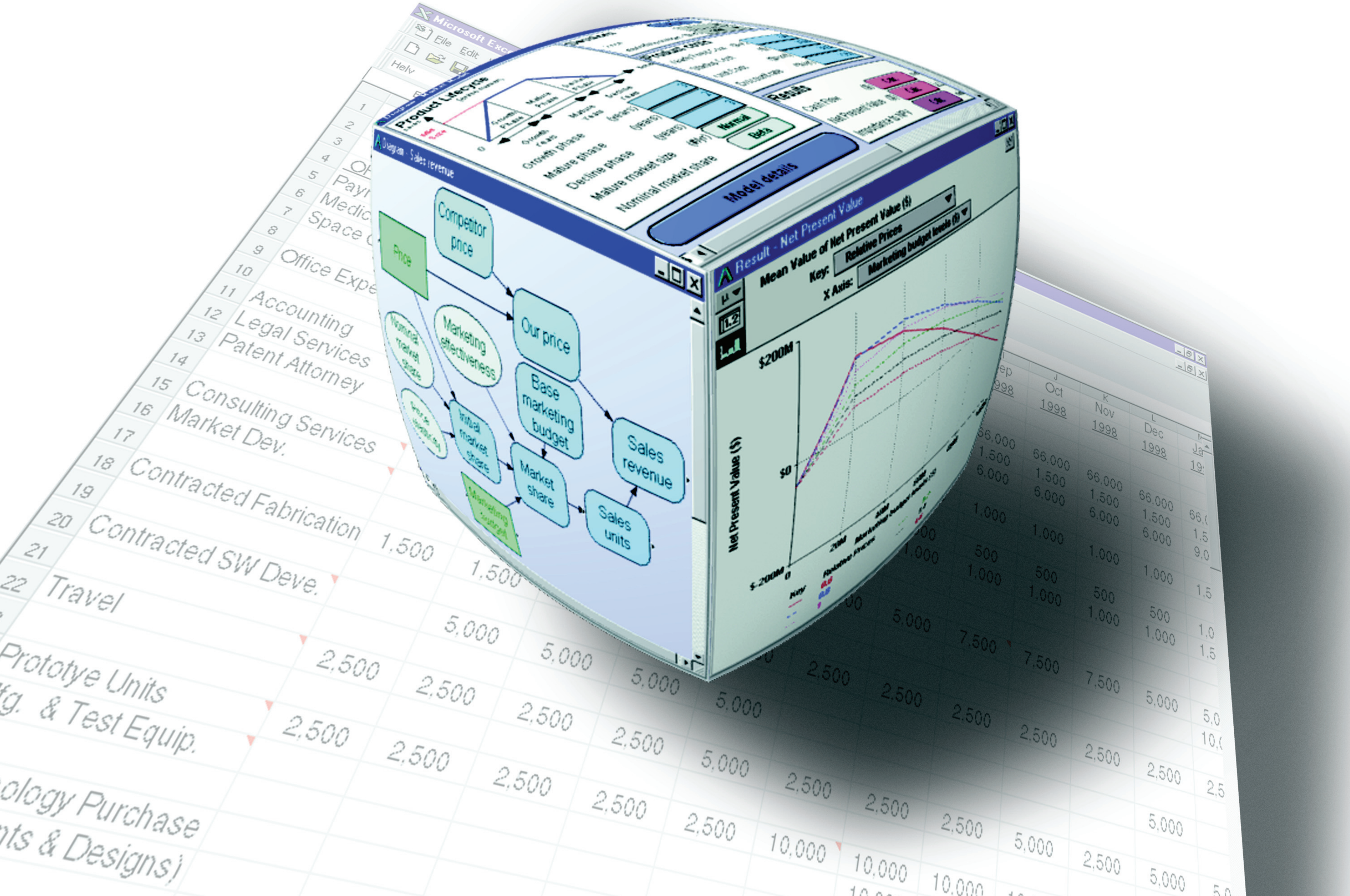
analytica®
Beyond the Spreadsheet

Optimizer Guide

Analytica 4.3

3 March, 2011



Chapter 1

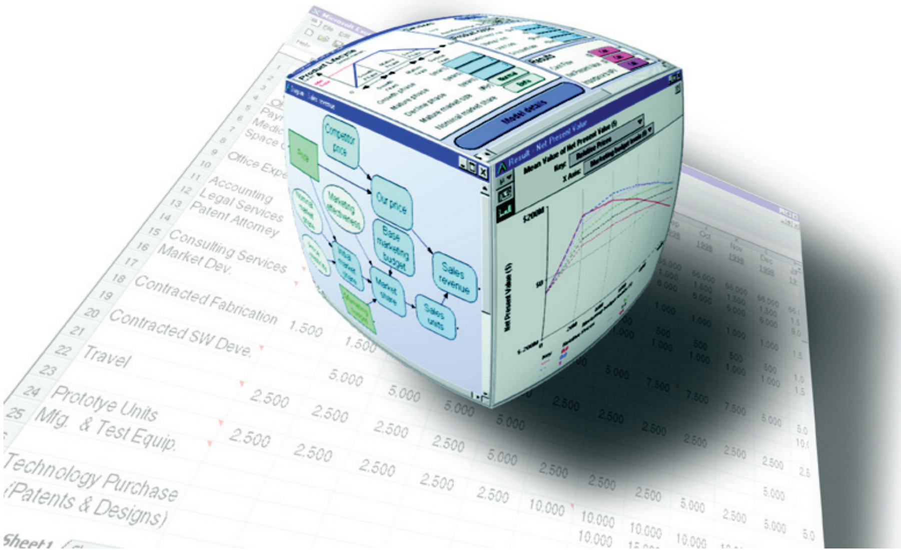
Lumina Decision Systems, Inc.
26010 Highland Way
Los Gatos, CA 95033
Phone: (650) 212-1212
Fax: (650) 240-2230
www.lumina.com

Copyright Notice

Introduction

This introduction explains:

- How to use this Optimizer Guide
- What is the Analytica Optimizer?
- How to obtain the Analytica Optimizer
- How to activate the Analytica Optimizer
- How to activate the Analytica Optimizer for ADE
- How to activate high-end add-on Optimizer engines
- What's new in Analytica Optimizer 4.3



How to use the Analytica Optimizer Guide

This Guide explains how to use the Analytica Optimizer. It provides a Quick Start Tutorial in Chapter 1 and an introduction to the basic concepts of optimization, including linear, quadratic, and nonlinear programming in Chapter 2. Special topics for NLPs are also covered in Chapter 4. But, it's not a complete textbook on optimization. You might find it useful, especially for more challenging applications, to consult one of the many good textbooks on optimization.

If you're new to Analytica

You will find it easier if you first learn the essentials of Analytica before learning the Analytica Optimizer described here. Start with the *Analytica Tutorial* to learn the basics of interacting with Analytica and its modeling language, especially Chapter 5, *Working with Arrays*. It's important to have a good understanding of Intelligent Arrays to make good use of the Optimizer.

If you're new to the Analytica Optimizer

We suggest you start with Chapter 1, *Quick Start*, an introductory tutorial that takes you through the key steps to create a simple optimization. Chapter 2, *Optimization characteristics* explains the general principles of optimization and the types of optimization, including Linear Programming (LP), Quadratic Programming (QP), and Non-Linear Programming (NLP). We also recommend Chapter 3, *Optimizing with Arrays* to master optimization with parametric analysis. Chapter 4 explains how to use optimization in models having dynamic influences and Monte Carlo-based uncertainties.

If you've used Analytica Optimizer 4.2 or earlier

This release 4.3 of Analytica introduces *Structured Optimization*, a new set of features that eliminates many difficult steps previously required for structuring optimizations in Analytica. For example, it can use a set of decision variables of varying dimensions, instead of requiring you to combine and flatten them manually. It introduces Constraints as a new object class. It can discover automatically whether your objective is linear, quadratic, or nonlinear, and apply the appropriate solver engine --and a whole lot more.

So if you've used the Optimizer before, we strongly recommend that you learn about these new features so you can take full advantage of them. *"What's new in Analytica Optimizer 4.3" on page 6* and then read Chapter 1, *Quick Start*, and Chapter 4, *Optimizing with Intelligent Arrays* (at least). Even though Analytica 4.3 still supports functions from earlier releases of Optimizer for backward compatibility, you will likely want to learn and use the new functions instead.

What is the Analytica Optimizer?

The Optimizer enhances Analytica with powerful functions to find optimal decisions and to solve equations. Most Optimizer models aim to find a decision strategy — values for Decision variables — to maximize or minimize a quantified objective subject to a set of equality or inequality constraints.   Some models merely seek a feasible solution that satisfies a set of constraints without regard to an objective.

Types of Optimization

A Linear Program (LP) requires the objective function and constraints to be linear functions of decision variables. Linear programs are solvable using straightforward algorithms that yield unique (global) maximizing or minimizing solutions. Although LP algorithms are well understood, large scale optimizations can be computationally complex.

When pairs of decision variables are multiplied together, including squared decision variables, quadratic terms result. A QP is a problem with quadratic terms in the objective and linear constraints. A generalization of a QP, in which one or more constraints contains quadratic terms is called a Quadratically Constrained Program (QCP). If the objective and constraint functions satisfy a mathematical property known as convexity, QP solutions are always unique (global). Non-convex formulations can result in “local” solutions that may or may not represent the global optimum.

NonLinear Programming (NLP) imposes no restrictions on the mathematical properties of objectives and constraints. A wide variety of computational algorithms can be applied to NonLinear Programs. Strategies include gradient tracking and genetic algorithms that allow potential solutions to compete within the computation.

With all classes of optimization, Analytica supports variables that are continuous, discrete (integer, Boolean, grouped), or mixture of continuous and discrete decision variables.

Analytica Optimizer 4.3 can analyze your objective function and constraints automatically to discover the type of Optimization and select the appropriate solver engine. See the section on the Type parameter of **DefineOptimization()** for more details on these types of Optimization.

Premium Solver Specifications

The standard edition of Analytica Optimizer uses the Premium Solver Platform licensed from Frontline Systems, Inc. Frontline developed the Optimizer/Solver in Microsoft Excel, and is the world leader in spreadsheet optimization. The Premium Solver is the leading add-on software for spreadsheet optimization, and incorporates state-of-the-art technologies. The LP and QP solver engines that come included with Analytica Optimizer handle up to 8000 variables and 8000 constraints in addition to bounds on he decision variables. Up to 2000 of these variables may be constrained to be integer-valued for Mixed Integer Programing (MIP). Up to 2000 decision variables of any kind are supported when quadratic constraints are present. The NLP solvers 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. Large scale add-on engines are also available at extra cost that eliminate the limit on variables and constraints entirely (see).

If your problems exceed these limits, or you need a solver that is even faster, you can add any of a number of high-end solvers, LP, QP, or NLP, that include some of the most powerful solver engines available anywhere. See “Installing Optimizer add-on engines” on page 5 and the **Engine** parameter of **DefineOptimization()** on page 76 for details.

Optimize with uncertain values and Intelligent Arrays

The Analytica Optimizer performs optimization under uncertainty to maximize expected values and minimize loss percentiles, as well as other statistical functions of objectives and constraints. Analytica 4.3 allows users to combine Intelligent Arrays with all classes of optimization. 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.

Compatibility with other Analytica editions

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

How do I obtain the Analytica Optimizer?

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

If your copy of Analytica is for release 4.2 or earlier, you need to upgrade to release 4.3 to obtain the newest Optimizer features as described in this manual. Substantial discounts are available if you have a maintenance agreement for Analytica 4.2 (included free for 12 months from purchase).

For more information:

- Visit the Lumina web site: http://www.lumina.com
- Call Lumina at 650-212-1212

Activating Analytica Optimizer

If you have purchased *an individual license* of Analytica Optimizer, you will be sent an *Activation Key*. This key allows you to retrieve a license specific to your computer and account from Lumina's activation server. The license itself is contained in a file, usually stored in the Analytica installation directory and usually named `Analytica.lic`. The term *activation* refers to the process of obtaining this file.

First time install

When you are installing Analytica for the first time, you will run the Analytica installer program, downloaded from `http://www.lumina.com/support/downloads/`. If you have purchased the 64-bit edition, download `Ana64Setup.exe`, otherwise download `AnaSetup.exe`. If you run the installer from your own end-user account (recommended), then enter the activation key when prompted, and the installer will automatically retrieve and install your license file over the internet (if you don't have an internet connection, see "Manual Activation" below). If a system administrator runs the installer from an account other than your own end-user account, then leave the activation key field blank, then after it is installed, log into your own account and follow the instructions for "Previous edition already installed".

Previous edition already installed

If you already have any edition of Analytica 4.3 installed (including Trial), your installation includes the Optimizer files and there is no need to download new software. To activate the Optimizer software while you are connected to the internet, you need to enter a new activation key with the Optimizer option. Follow these steps:

1. Start Analytica in the usual way, e.g., via the Windows Start menu, or by double-clicking an Analytica model file.
2. From Analytica's **Help** menu, select the **Update license...** option, to show the **Analytica Licensing Information** dialog box.
3. Enter the activation key into the **License ID** box, replacing the license name currently displayed. As soon as the key is correctly entered, an **Activate** button appears to the right. When you press **Activate**, your license is retrieved over the internet from an activation server. Your license name will then appear in the **License ID** box.
4. Click **OK**.
5. Exit and restart Analytica.

Manual Activation

Automatic activation will fail if you do not have a connection to the internet, or if your firewalls and proxy servers are configured to prevent Analytica from communicating with the activation server. In this case, you can obtain the license file through *manual activation*. You will need your activation key and the **Host ID** and **User ID** that are displayed at the bottom of the **Analytica Licensing Information** dialog from Step 2 above. Enter this information into the form at `http://www.lumina.com/support/activate-analytica/`, and the license file will be emailed to you.

Floating License

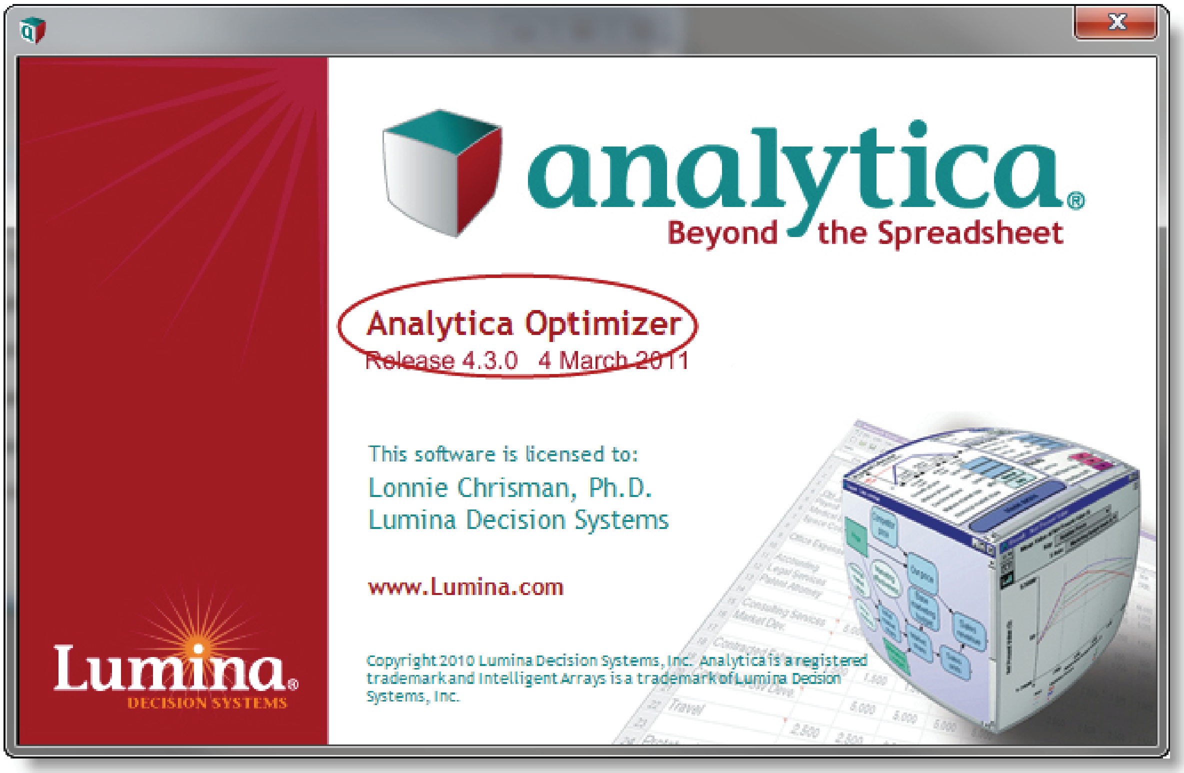
If your organization provides you with a floating license to Analytica Optimizer, then your IT department will provide you with the name (and possibly IP port) of the Reprise License Manager (RLM) server computer where your IT department will have already installed and activated a floating license.

Run the Analytica installer as described in “First time install” on page 4, but during the install, on the **License Information** page, select **Centrally managed license (RLM License Server)** and enter the server host name, or port@host, provided to you.

If you already have Analytica installed, you can also enter this information into the **Analytica License Information** by selecting **Update License...** on the **Help** menu.

Validating successful activation

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 Optimizer,” as shown below.



Analytica Optimizer

Release 4.3.0 4 March 2011

This software is licensed to:

Lonnie Chrisman, Ph.D.

Lumina Decision Systems

www.Lumina.com

Copyright 2010 Lumina Decision Systems, Inc. Analytica is a registered trademark and Intelligent Arrays is a trademark of Lumina Decision Systems, Inc.

Installing Optimizer add-on engines

You can add on other engines for solving optimization problems. Some engines provide superior performance on particular classes of optimization problems, and some engines handle larger numbers of variables or constraints. The following add-on engines are available:

- Large Scale LP
- Large Scale SQP
- Large Scale GRG
- Gurobi (LP/QP)

- XPRESS (LP/QP)
- MOSEK (cQCP/NLP)
- KNITRO (NLP)
- OptQuest (NSP)

When you purchase an add-on engine license, or obtain an add-on engine trial, Lumina will provide you with the following items:

- `EngineSetup.exe` (or `EngineSetup64.exe` for Analytica 64-bit users), the add-on engine installer from Frontline Systems.
- The password required by EngineSetup
- The file `SolverAddons.lic`
- A Lumina activation key.

Installing an add-on engine requires the following steps:

1. Install Analytica Optimizer first, if you have no already done so.
2. Run `EngineSetup.exe` or `EngineSetup64.exe`, the add-on engine installation program. Use `EngineSetup64.exe` when you are using Analytica 64-bit.
3. The `EngineSetup` program asks for a password and activation key. The password will be provided to you by email when you are sent the Engine Setup program. You can leave this activation key field blank.
4. Start up Analytica and select **Update License...** from the **Help** menu.
5. Enter the activation key provided into the **License ID** box and press the **Activate** button.

Validate the activation

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

```
Variable Engines := OptEngineInfo("All","TrialPeriod")
```

The names of available engines should appear with non-zero values.

What's new in Analytica Optimizer 4.3

Analytica Optimizer 4.3 eliminates most of the work in formulating a model for optimization while offering increased flexibility and power compared to release 4.2 earlier. This new framework is known as Structured Optimization. Some of these features are similar to those found in algebraic optimization languages. But in Analytica, they build on the strengths of Analytica's influence diagrams and Intelligent Arrays.

These are the highlights:

- It uses **the influence diagram structure** of your original model. You no longer need to reformulate your model for the Optimizer to combine and flatten the Decisions into a vector of scalar variables. All you need to do is specify which decision variables you want to optimize. Even if the decision variables are arrays with differing dimensions, it combines them automatically into a single Decision Vector to pass to the Solver without you having to think about it.

- A new function, **DefineOptimization()**, supports any type of optimization, replacing the original separate functions **LpDefine()**, **QpDefine()** and **NLPDefine()**[1].
- It adds ***Constraint*** as a new class of object for optimization, joining the existing objects: Variable, Decision, Objective, and Chance.
- It is designed to co-exist with parametric analysis and other non-optimized uses of the model.
- Users can encode Linear, Quadratic, and Non-Linear problems in the same way.
- Objectives and constraints are encoded using Analytica's usual expression syntax (i.e., variable definitions). Constraints are encoded as comparisons (<=, >=, =, or Ranges: lb <= x <= ub).
- You can have any number of separate decision nodes and constraint nodes. Each decision or constraint can be multidimensional, and the dimensionality of each decision or constraint can be different. No need to flatten, unflatten, or conconcatenate multidimensional decisions and constraints. It takes care of all that for you.
- In linear and quadratic cases, it determines the coefficients automatically by analyzing the Objective function. There is no need for you to separate out the coefficients. You just use standard Analytica expressions.
- Coefficients for linear and quadratic problems are processed using sparse matrices of coefficients, enabling the solution of very large (but sparse) LP, QP and QCP problems.
- It infers whether indexes are intrinsic to the problem or are extrinsic - i.e.should be array abstracted, with the optimization run separately for each value of the index. Or you can override the default inference, if you want to control the array abstraction.
- It greatly expands the richness and flexibility of the Domain attribute for Decision variables, letting you specify lower and upper bounds, Boolean or Integer type (grouped integer for Mixed Integer Programming), or even array-valued domains (where integer type, bounds, etc. vary along one or more indexes).
- It supports Decision variables with explicit Domains - e.g., a lists of labels. It formulates these as integer-valued NLPs, and automatically manages the mapping from domain value to the integer used internally by the solver.
- It supports **Piecewise-Linear** functions (i.e., models that use the **LinearInterp** function). This feature automatically converts non-linear curves to an approximate linear form so that suitable non-linear models can be solved using LP methods.
- It supports ***semi-continuous*** variables. These are variables whose value must fall within given bounds, or may be zero.
- Library functions are provided for copying the solution into the definition of the decision variables, and for restoring the original definitions. These can be utilized from button scripts.
- Optimization can be used in conjunction with Analytica’s uncertainty analysis and Dynamic simulation.
- The names of optimizer functions have been changed to begin with **Opt...**, whereas in release 4.2 and earlier these functions begin with **Lp...** For example,

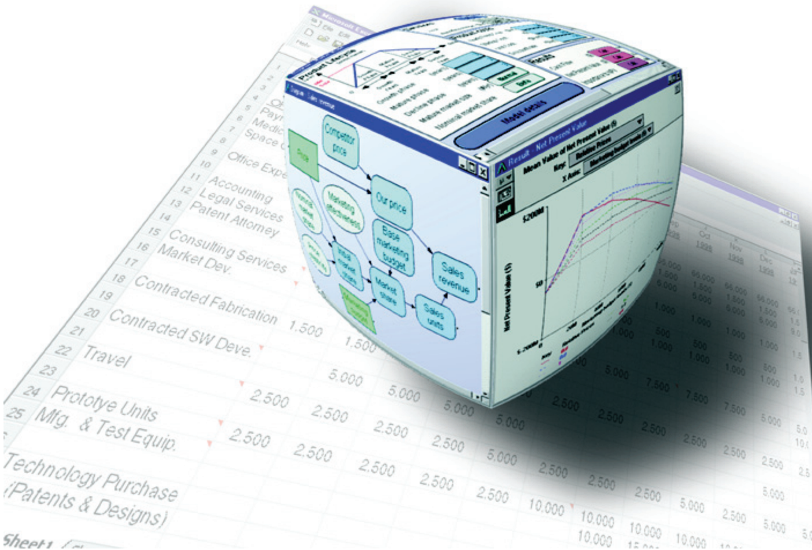
[1] It still supports these legacy functions for compatibility with older models. See the Analytica Wiki (wiki.lumina.com) for details

OptSolution() replaces the old function **LpSolution()**, **LpStatusText()** is now **OptStatusText()**, etc. The name change reflects the fact that most these functions apply to non-LP optimizations equally well. The older functions are now deprecated, but will continue to function as before.

Chapter 1 *Quick Start*

This chapter shows you how to:

- Set up a basic NLP optimization model using Decision, Objective, and Constraint Nodes
- Define the central Optimization node using the **DefineOptimization()** function
- Obtain solution output and status
- Specify domain types (i.e., integer, continuous, etc.) and bounds for decisions
- Combine parametric analysis with optimization
- Change initial guesses for non-convex solution spaces



Introduction to Structured Optimization

The Analytica Optimizer 4.3 release includes a significant new set of features, collectively called Structured Optimization, designed to simplify the optimization modeling process. In Structured Optimization format, LP, QP and NLP optimizations can all be modeled in similar ways.

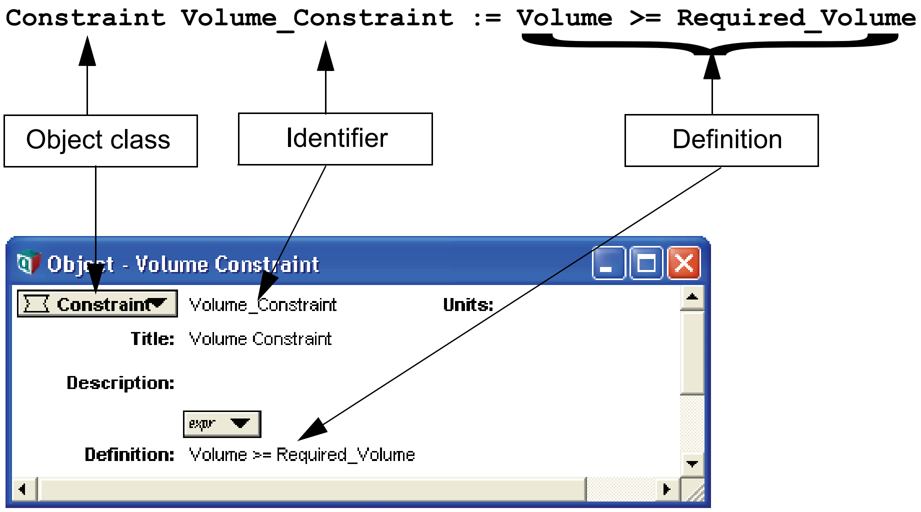
All types of optimization are specified using the **DefineOptimization()** function[1]. The **DefineOptimization()** function automatically analyzes your model to determine the type of optimization and selects the appropriate optimization engine, although you can still override this process if desired.

Another significant change associated with Structured Optimization is the introduction of a new object type, the Constraint. Constraint objects give users the ability to specify constraints, or arrays of constraints, in common expression format using equality or inequality operators. This intuitive interface allows users to easily integrate different types of constraints and to organize constraint arrays efficiently.

This chapter includes simple NLP examples to demonstrate the roles of Decision variables, Constraint objects, Objective variables, and Decision attributes in the Structured Optimization framework. The same basic structure applies to LP and QP optimizations as well.

Notation

Throughout this manual, we use a shorthand notation for displaying the definitions of Analytica objects. An object's class (e.g., Variable, Decision, Constraint, etc) and identifier is followed by `:=`, and then the definition is shown, e.g.:.



```
Constraint Volume_Constraint := Volume >= Required_Volume
```

Constraint Volume_Constraint Units:

Title: Volume Constraint

Description:

Definition: Volume >= Required_Volume

[1] **DefineOptimization()** supercedes the **LPDefine()**, **QPDefine()** and **NLPDefine()** function that were used to specify optimizations prior to Analytica 4.2. These functions remain available for backward compatibility, but are now deprecated.

The Optimum Can Example

The *Optimum Can* example determines the dimensions of a cylindrical object having minimum surface area for a given volume. Admittedly, this is not a very interesting optimization problem. In fact, the solution can be derived on paper using basic differential calculus. (Spoiler alert! The optimum can has height equal to twice the radius.) But the simplicity of the example allows us to focus on the workflow and object relationships using the new Structured Optimization framework in Analytica.

Decisions

In this example we will decide on the **Radius** and **Height** of a cylindrical vessel. We represent each of these as a Decision variable in the influence diagram. The values we define for these nodes will be used as initial guesses for optimization types that require an initial guess (NLP or non-convex QP). Otherwise the definitions of these inputs are not important. We use 1cm as an initial guess for both the radius and height.

```
Decision Radius := 1
Decision Height := 1
```

Constants

Constants have no special interpretation in optimization definitions. They can be used as usual for values that stay constant in the model. In this example we will use a Constant for the required volume which does not vary in the model.

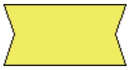
```
Constant Required_Volume := 1000
```

Variables

General variables are used for intermediate values as well as for the central **DefineOptimization()** function described below. We also use a variable to define `Volume` of the cylinder.

```
Variable Volume := pi*Radius^2*Height
```

Constraints



Constraints contain equalities or inequalities that restrict the range of optimized results. In this example, we use a constraint object to enforce the minimum volume requirement on our can.

```
Constraint Volume_Constraint := (Volume >= Required_Volume)
```

Objectives

Most optimizations have an objective value to maximize or minimize. (Some problems are only concerned with feasible solutions that meet constraints.) In this example we are minimizing the surface area of our can. We define surface area using an Objective variable. The can has round disks at the top and base with surface area (πR^2) and a tubular side with surface area ($2\pi RH$).

```
Objective Surface_area :=
      2 * (pi * Radius^2) + (2 * pi * Radius * Height)
```

The DefineOptimization() function

The **DefineOptimization()** function is the key component of all Structured Optimization models. It brings all other components together, specifying the optimization to be performed. This function is typically placed in a Variable object in the center of our influence diagram. Although this function includes many optional parameters, we will only use the core parameters in this example:

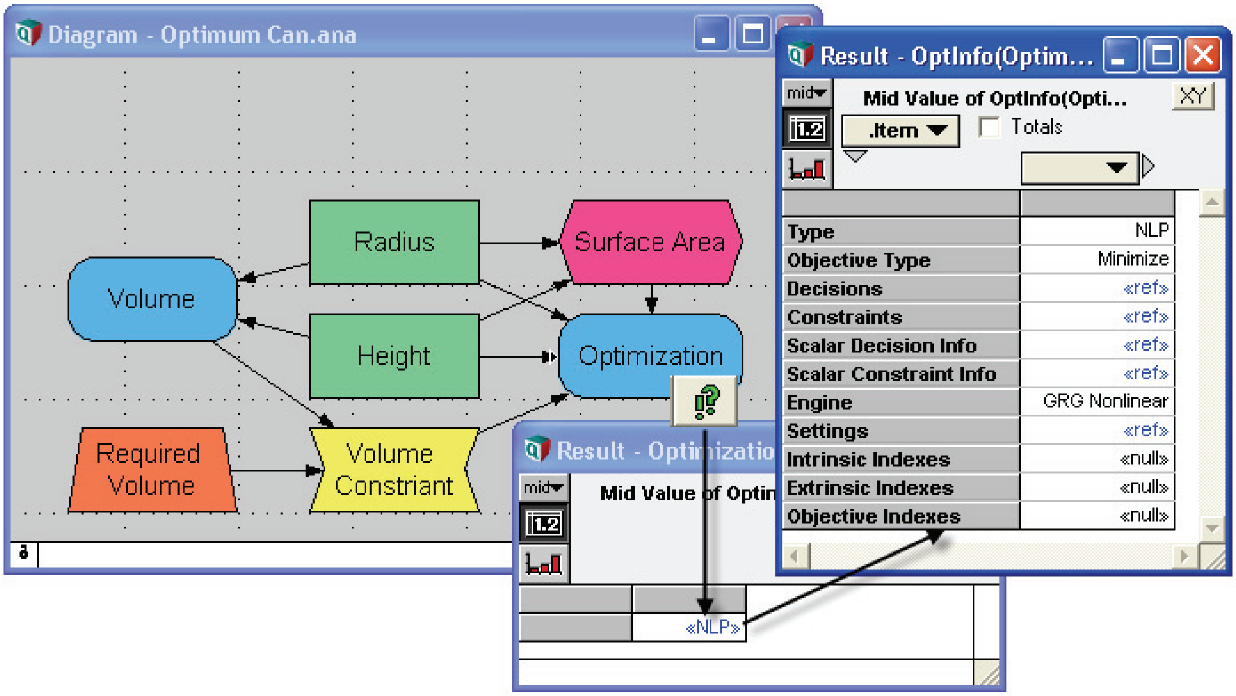
- **`Decision:`**
  Identifier for the decision node (or a list of identifiers separated by commas if there are multiple decisions). Specify **All** to include all decision nodes in the model or **All in *module*** to include all desired decisions within a designated module.
- **`Constraint:`**
  Identifier for the constraint node (or a list of identifiers separated by commas if there are multiple constraints). Specify **All** to include all constraint nodes in the model or **All in *module*** to include all desired constraints within a designated module. You can also specify inequality or equality expressions directly, or omit the parameter entirely in an unconstrained optimization problem.
- **`Maximize/Minimize:`**
  Use the words “Maximize” or “Minimize” depending on the type of problem. Follow this with an expression or with the identifier for the relevant objective node.

We define our Define Optimization node as:

```
Variable Opt :=
   DefineOptimization(
      Decisions: Radius, Height,
      Constraints: Volume_Constraint,
      Minimize: Surface_area)
```

Viewing the Optimization Object

The **DefineOptimization()** function evaluates to a special object that contains detailed information about the optimization. The object appears as a blue hyperlink that shows the type of optimization problem you have constructed. In this case we see it is **«NLP»**. You can double-click the optimization object to open a new window revealing internal details from the optimization engine. Clicking reference objects allows you to drill down to finer levels of detail. This information is also available by using the `OptInfo()` function (See “OptInfo(Opt, "Item", Decision, Constraint, asRef)” on page 81 for more details).



Type	NLP
Objective Type	Minimize
Decisions	«ref»
Constraints	«ref»
Scalar Decision Info	«ref»
Scalar Constraint Info	«ref»
Engine	GRG Nonlinear
Settings	«ref»
Intrinsic Indexes	«null»
Extrinsic Indexes	«null»
Objective Indexes	«null»

In this case we have allowed Analytica to automatically determine the type of problem. Alternatively, you can specify the problem type along with the desired engine and other settings by adding optional parameters to **DefineOptimization()**. See the “Optimizer Function Reference” chapter on page 72 for more details about the **Type** and **Engine** parameters of **DefineOptimization()**.

Obtaining the Solution

Now that we have specified an optimization, how do you compute and view the result? You may be tempted to re-evaluate the `Radius` and `Height` decision variables to see if their values have changed. But this is not how optimization works in Analytica. Input values always retain their original definitions. (In this case we simply used 1 as a dummy value for `Radius` and `Height`.) To obtain the solution, you need to create an output node defined with the **OptSolution()** function. This function usually uses two parameters:

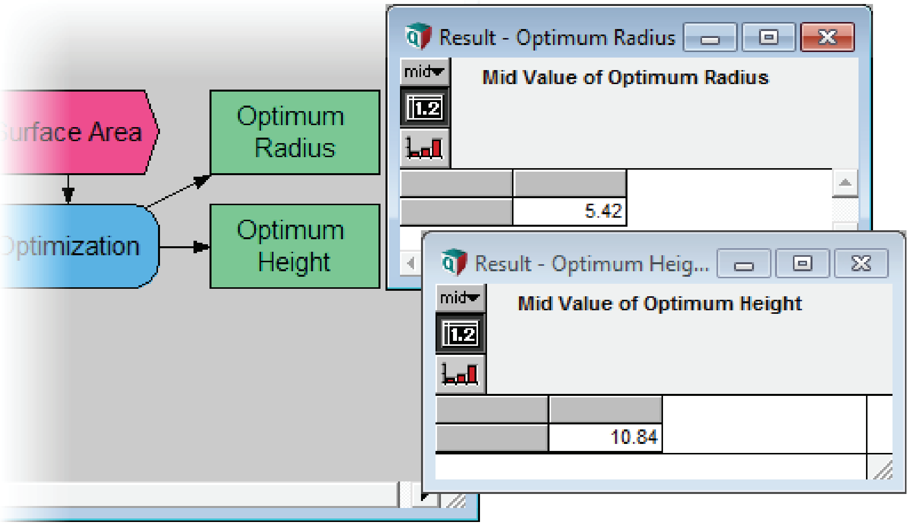
```
OptSolution(Opt,Decision)
```

- ***Opt***: Identifier for the node containing **DefineOptimization()**
- ***Decision***: Identifier for the counterpart Decision input node

```
Decision Opt_Radius  := OptSolution(Opt, Radius)
Decision Opt_Height  := OptSolution(Opt, Height)
```

The Decision parameter is optional. If it is omitted, the solution will include all decisions along a local index named `.DecisionVector`.

As expected optimum Height value is twice the Radius.



Obtaining the optimized Objective value

To conveniently evaluate the optimized objective value (the surface area of the solution can) you can use the **OptObjective()** function. The only parameter is the identifier for the Define Optimization node.

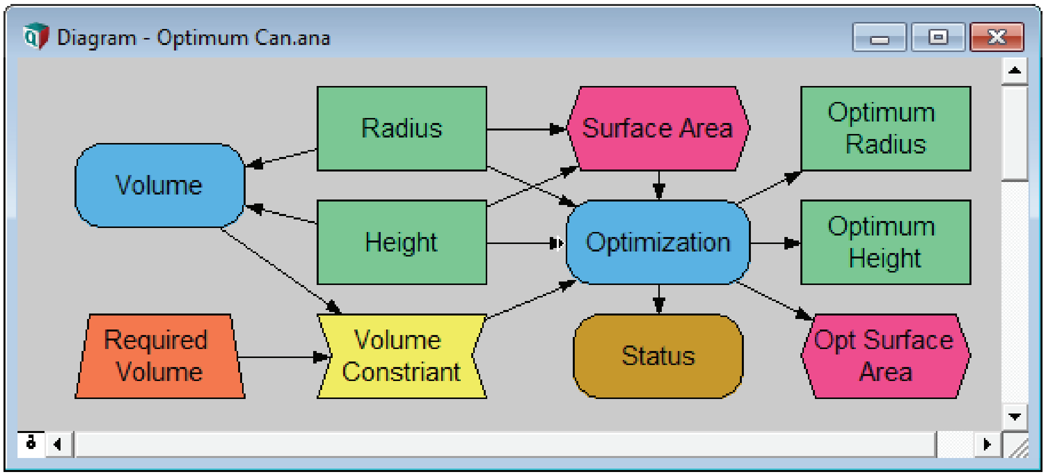
```
Objective Opt_Surface := OptObjective(Opt)
```

Viewing Optimization Status

To check the status of an optimization, it is convenient to use the **OptStatusText()** function. Enter the identifier for the node containing **DefineOptimization()**.

```
Variable Status := OptStatusText(Opt)
```

This will reveal a text string describing the status of the optimization result. Status messages differ according to problem characteristics and the engine being used. In general these messages indicate whether or not a feasible solution has been found and if so, whether or not the optimizer was able to converge to a bounded solution. In this example status is: “Optimal solution has been found.”



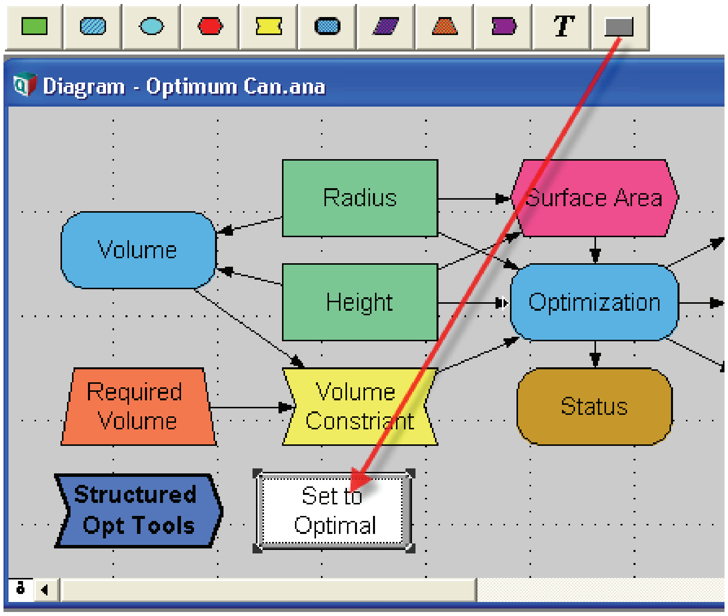
Copying Optimized Results to Definitions

In some cases, you may wish to copy the optimized decision values into the definition of the original decisions. With this, the result for variables downstream of the decisions will reflect their optimal values as well.

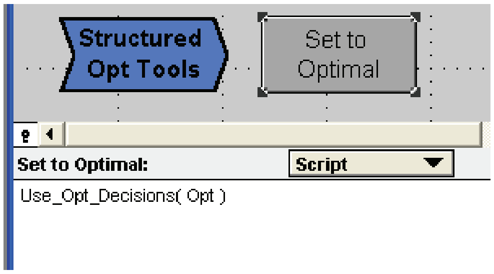
You can configure your model to copy optimized results into the original decisions by adding two buttons to your model. The first button solves for the optimal solution and copy the optimal values. The second button restores the original (non-optimized) definition. Functions provided in the `Structured Optimization Tool.ana` library take care of the details.

To configure these buttons:

1. With the diagram in focus, select **Add Library...** from the **File** menu.
2. Select `Structured Optimization Tools.ana` and press **Open**. Select **Embed**, **OK**.
3. Drag a button from the tool bar, title it “Set to Optimal”:



4. Press to edit the button’s **Script** attribute. Enter: `Use_opt_decisions( opt )`



Set to Optimal: Script

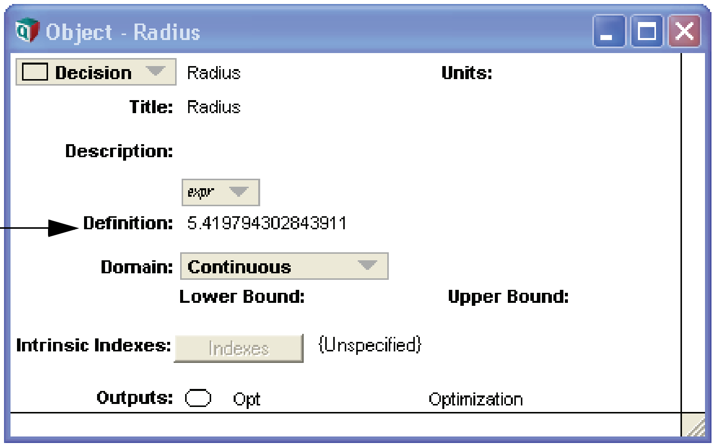
Use_Opt_Decisions(Opt)

5. Drag a second button to the diagram, name it “Restore Defintions” and set its Script attribute to:

```
Restore_Decision_Defs( opt )
```

Now we're ready to try them out.

6. Press ☝ to enter browse mode. Press the **Set to Optimal** button.
7. Open the object window for **`Radius`**:



Object - Radius

Decision: Radius

Units:

Title: Radius

Description:

expr

Definition: 5.419794302843911

Domain: Continuous

Lower Bound:

Upper Bound:

Intrinsic Indexes: Indexes {Unspecified}

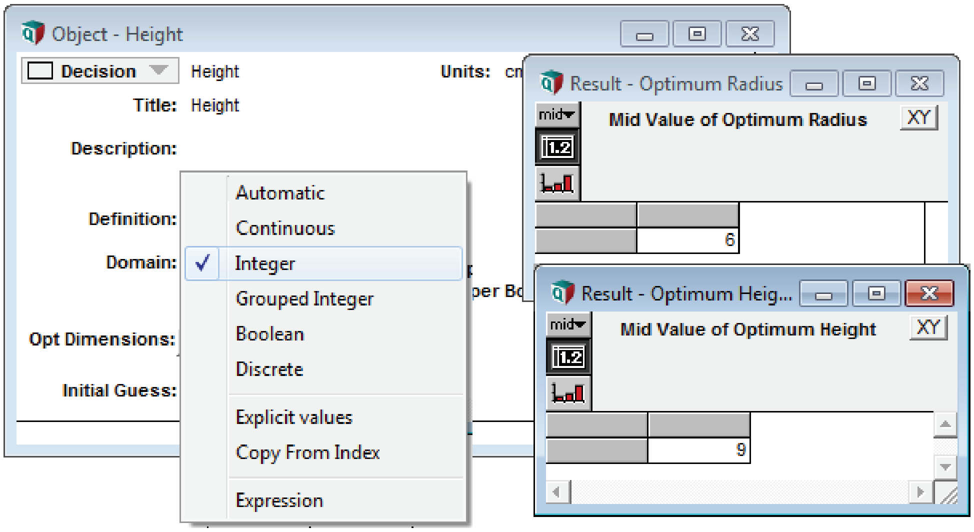
Outputs: Opt Optimization

The definition has been filled in with the optimal value

Changing variable types (Domain)

Double-click either **`Radius`** or **`Height`** to open the **Object window** for the node. You will notice pull-down menu for **Domain**. This attribute specifies the variable type. It is always visible for decision nodes if you are using the Optimizer edition.

Suppose the factory requires **`Radius`** and **`Height`** to be integer values in centimeters for tooling purposes (or maybe just because they don't like decimals). Change the Domains of **`Radius`** and **`Height`** to "**Integer**" and re-evaluate the solution:



Object - Height

Decision: Height

Units: cm

Title: Height

Description:

Definition:

Domain: ✓ Integer

Opt Dimensions:

Initial Guess:

Result - Optimum Radius

Mid Value of Optimum Radius

6

Result - Optimum Heig...

Mid Value of Optimum Height

9

The new solution finds the integer values that come closest to meeting the optimization criteria.

See the Attribute Reference Chapter for descriptions of all available domains.

Setting bounds on decision values

Suppose the cans must not exceed a 5cm radius in order to meet National Association for the Advancement of People with Small Hands (NAAPSH) guidelines. One way to set this limit would be to add another constraint. But since this restriction applies directly to one of the decision variables, it is easier to simply set an upper bound on the variable directly.

Double-click the **Radius** variable and enter **5** as the upper bound. The updated solution will describe a thinner can: 5cm in Radius and 13cm in Height.

Bounds and Domains

Some Domain types are not compatible with bounds. If one of these domains is selected (i.e. **Boolean**), bounds attributes will not be visible.

Bounds and Feasible Solutions

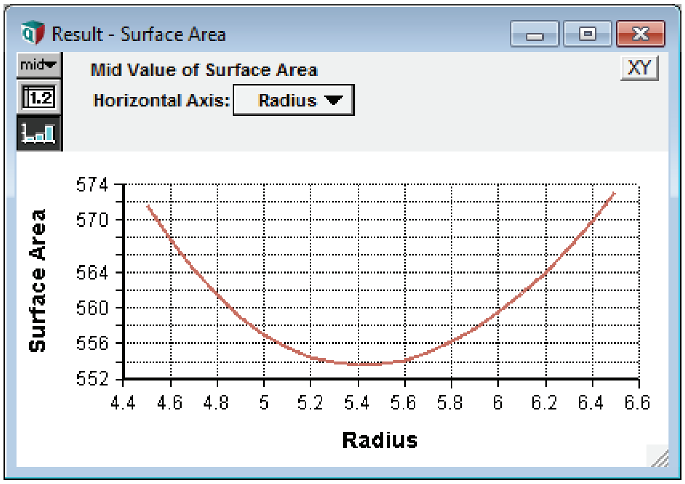
It is possible to have no feasible solution within the designated bounds. For example if you restrict **Radius** to 5cm while restricting **Height** to 10cm, it will be impossible to produce a can that meets the minimum volume constraint. The `OptStatusText()` function indicates whether or not a feasible solution has been found.

Using Parametric Analysis with Optimization

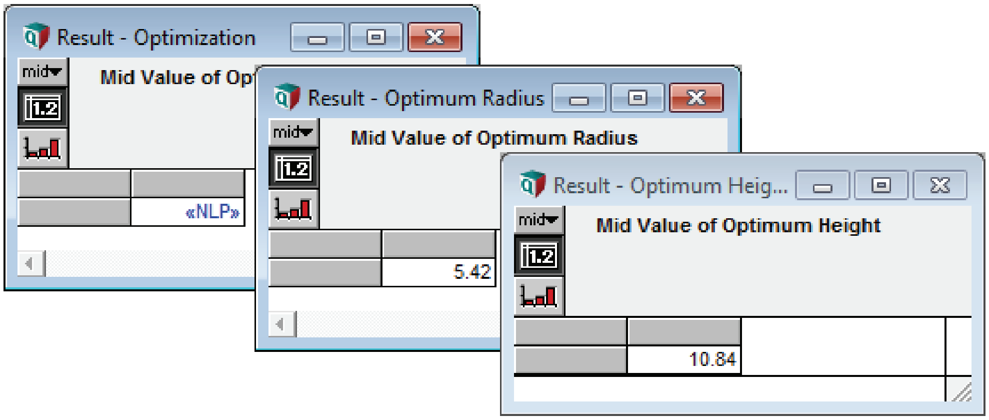
Before adding optimization to existing models, it is often useful to perform a parametric analysis to see how variations in decision inputs affect the objective value. If you have done this, your Decision and Objective variables will include parametric indexes. To demonstrate this in the *Optimum Can* example, we can define the `Radius` to be a sequence of values that vary parametrically. We then re-define `Height` such that the volume of the cylinder remains constant as radius varies:

```
Variable Radius := Sequence(4.5, 6.5, 0.1)
Variable Height := Required_volume/(pi*Radius^2)
```

Now you can evaluate the objective **Surface_Area** to see how it is affected by **Radius**.



An optimization requires a scalar-valued objective. An array-valued objective usually implies an array of optimizations, each optimizing an individual element of the objective array. But parametric indexes are an exception to this rule! If the Objective is an array over parametric indexes, the indexes are ignored by the optimization. So even though we have an array valued Objective in this example, there is still only one optimization run.



Parametric analysis is a good way to gain insight into your model. The Structured Optimization framework is designed so that it will not be confused by this.

The Initial Guess attribute

LP and convex QP problems do not rely on initial guesses and always yield globally optimal solutions. But in NLP and non-convex QP problems it is not always possible to guarantee that a solution found by the optimizer is a global optimum. It might be merely a “local” optimum within the solution space. Optimization methods for these problems use an initial guess from which to start the search for a solution. The particular solution the optimizer returns may depend on the starting point.

Normally, Analytica uses the defined value of the Decision variables as the initial guess. In the *Optimum Can* example, we initially defined `Radius` and `Height` as 1. If a decision variable is defined using a parametric index, Analytica uses the first element of the parametric array as the initial guess.

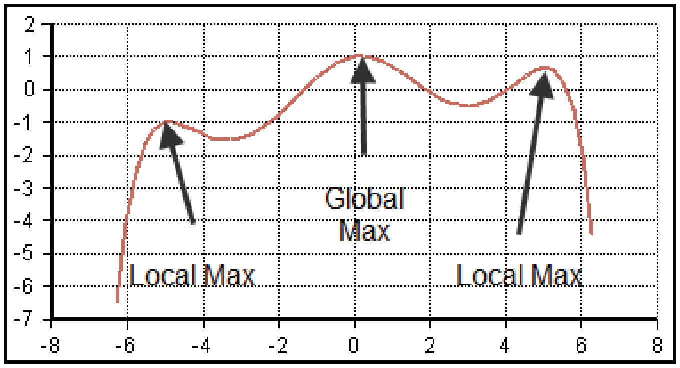
You can change the initial guess without re-defining the decision variable using the **Initial Guess** attribute in the Decision node. We can demonstrate this using the *Polynomial NLP.ana* example where the objective is a non-convex curve with local maxima.

The Initial Guess attribute is hidden by default. To make it visible in Decision nodes:

- Select **Attributes...** from the Object menu.
- Toggle a check box next to **Initial Guess**.

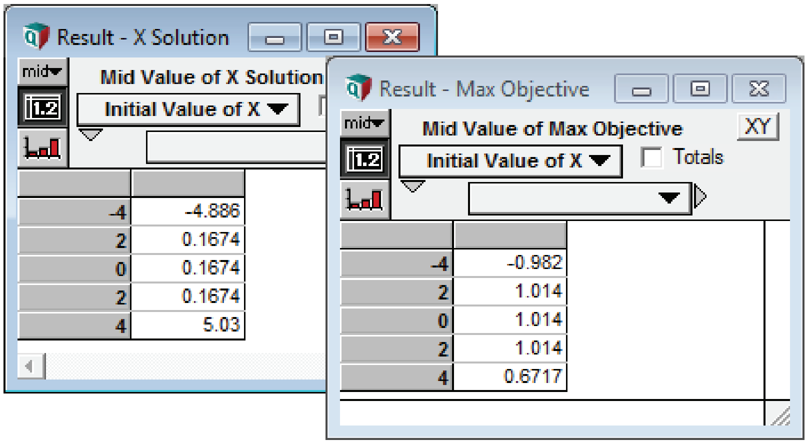
The attribute will now be visible in the Object windows of all Decision variables.

The polynomial curve in this model is designed to have several critical points.



```
Decision X := 0  (or any value at all)
Initial Guess of X := [-4, 2, 0, 2, 4]
Objective Polynomial :=
    1+X/6-X^2/2+X^4/24-X^6/720+X^8/40320-X^10/3628800
Variable Opt := DefineOptimization(
    Decision: X,
    Maximize: Polynomial)
Variable X_solution := OptSolution(Opt,X)
Objective Max_Objective := OptObjective(Opt)
```

The array of initial guesses will cause Analytica to abstract over the index and perform multiple optimizations.



Result - X Solution

Mid Value of X Solution

Initial Value of X

Initial Value of X	
-4	-4.886
2	0.1674
0	0.1674
2	0.1674
4	5.03

Result - Max Objective

Mid Value of Max Objective

Initial Value of X ☐ Totals

Initial Value of X	
-4	-0.982
2	1.014
0	1.014
2	1.014
4	0.6717

We see that the result depends on the initial guess for this non-convex NLP.

If the array of guesses were entered as a *definition* for the decision variable instead of as an *initial guess* attribute, Analytica would interpret it as a parametric index and apply only one initial guess. (See subsection above.) Therefore it is necessary to use the Initial Guess parameter if you want to perform multiple optimizations using an array of guesses.

Note: *Finding multiple different local extrema in this fashion can be a useful way to locate multiple solutions of interest. One often needs to combine unmodeled factors with insight and results obtained from a model, so these other solutions may turn out to be more interesting for reasons that you have not modeled.*

Note: *When your interest is in finding just the global optima, there are additional methods for dealing with the problem of local optima. The* ***Multistart*** *and topological search options can be utilized with gradient-based methods (see "Coping with local optima" on page 104). The "Evolutionary" and "OptQuest" engines use population-based search methods that are more robust to local optima.*

Chapter Summary

The *Optimum Can* example demonstrates the basic workflow of Structured Optimization in Analytica. It includes input Decision variables, a Constraint object, an Objective variable, intermediate Variables and the central **DefineOptimization()** function.

The **DefineOptimization()** function recognizes the non-linear characteristics of the Optimum Can model and classifies it as an NLP. The function evaluates as a special object containing details about the optimization.

The Domain attributes in Decisions allow you set variable type and bounds.

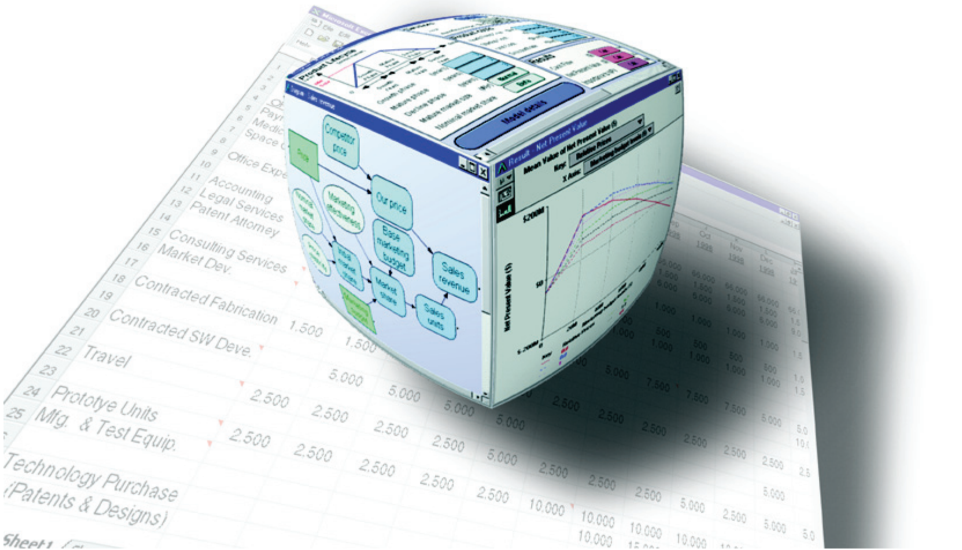
Structured Optimization is compatible with a decision variable defined as a parametrically varying sequence.

If the Initial Guess attribute is kept hidden or left blank, Analytica will use the defined value of the decision variable as an initial guess. Users can override this value or enter an array of initial guesses by using the Initial Guess attribute in Decision nodes. This attribute is hidden by default but can be made visible when necessary.

Chapter 2 *Optimization characteristics*

This chapter shows you:

- The different types of optimization problems
- How to recognize different types of optimization problems
- How to recognize problems that have continuous, discrete, or mixed variables
- How to optimize when solving simultaneous equations

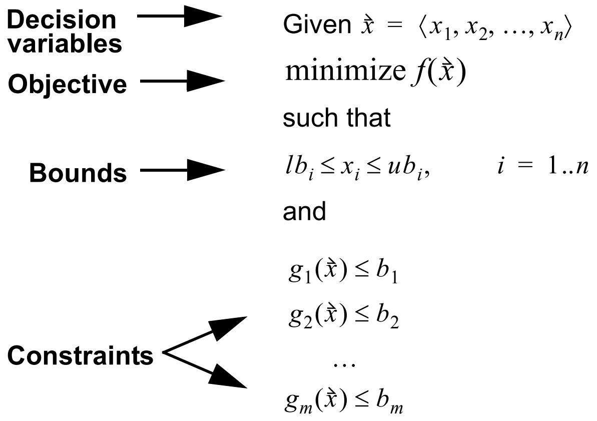


Introduction

Although the material in this chapter is not specific to Analytica, it should give users a foundation of knowledge sufficient to understand the basic characteristics of an optimization problem and to understand the mathematical characteristics that define different optimization types.

Parts of an optimization problem: General Description

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



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

minimize $f(\vec{x})$

such that

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

and

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

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

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

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

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

Identifying the type of optimization

A critical issue in formulating an optimization problem is determining whether it is linear, quadratic, or nonlinear. Although Analytica 4.3 can automatically determine class of optimization, it is important for the user should have a basic understanding of what these classes mean.

For a ***linear program (LP)***, the objective must be a linear function of the decision variables. For a ***quadratic program (QP)***, the objective is a quadratic function and the constraints must be linear functions of the decision variables. For a ***quadratically constrained program (QCP)***, the objective and constraints must all be linear or quadratic functions of decision variables. The problem is a ***nonlinear program (NLP)*** if the objective or any of the constraints are nonquadratic in any of the decision variables.

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

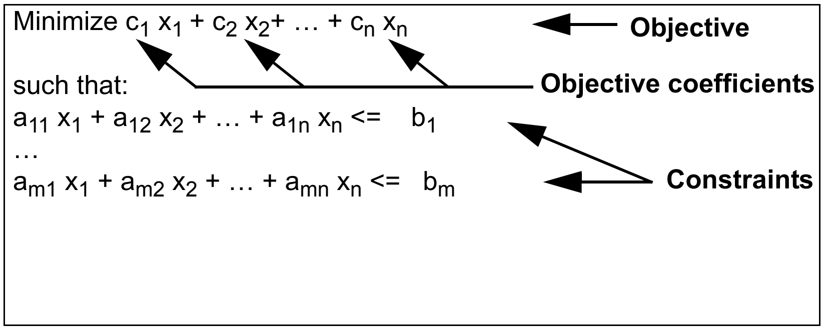
There are often several ways to formulate the same problem. Linear and quadratic formulations are faster and more flexible, so it is worth careful thought to see if it is possible to reformulate a nonlinear optimization into a linear or quadratic optimization. Often a simple transformation, combination,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

Specific Optimization Characteristics

The general description of optimization problems on page 16 applied to all problem types. The following sections offer more specific mathematical descriptions for each type of optimization:

Parts of a Linear Program (LP)

A linear optimization problem has the following standard formulation.



Minimize $c_1\, x_1 + c_2\, x_2 + \ldots + c_n\, x_n$

such that:

$a_{11}\, x_1 + a_{12}\, x_2 + \ldots + a_{1n}\, x_n <= \quad b_1$

$\ldots$

$a_{m1}\, x_1 + a_{m2}\, x_2 + \ldots + a_{mn}\, x_n <= \quad b_m$

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

Linear programs generally solve quickly, although large models can be computationally complex. Solutions to linear problems always represent a global maximum or minimum. This means that you always be assured of finding the absolute optimum solution.

Parts of a Quadratic Program (QP)

A quadratic program has the following standard formulation:

Minimize $\vec{c}\cdot\vec{x}+\vec{x}^TQ\vec{x}$ ← **Objective function**

such that:

$$\vec{a}_1\cdot\vec{x}\le b_1$$

...

$$\vec{a}_m\cdot\vec{x}\le b_m$$

Constraints

The objective and constraint left-hand sides are written here in matrix notation. The constraints are the same as they were in the LP formulation, and differ here only by the fact that they appear here in matrix notation. The term $\vec{c}\cdot\vec{x}$ is the linear part of the objective, and the $\vec{x}^TQ\vec{x}$ is the quadratic part of the objective,which is specified by the $n\times n$ matrix Q. $\vec{x}^T$ denotes the vector transpose of the decision variables.

Parts of a Quadratically Constrained Program (QCP)

The general form for a quadratically constrained quadratic program is:

Minimize $\vec{c}\cdot\vec{x}+\vec{x}^TQ\vec{x}$ ← **Objective function**

such that:

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

...

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

Constraints

This formulation augments the pure quadratic program by adding quadratic terms to the constraints. It may be the case that most constraints are linear, so that the Q_i matrices for those constraints are not present (or 0), but if even a single constraint is quadratic, the problem is classified as a QCP..

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

Parts of a Non-Linear Program (NLP)

The basic formulation for a nonlinear optimization is

$$\text{minimize } \underbrace{f(\vec{x})}_{\textbf{objective function}} \text{ such that } \begin{array}{l} g_1(\vec{x}) \le b_1 \\ g_2(\vec{x}) \le b_2 \\ \dots \\ g_m(\vec{x}) \le b_m \end{array} \longleftarrow \textbf{constraints}$$

where $\vec{x}$ is a vector denoting the *n*-dimensional candidate solution.

A nonlinear 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, and 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 whether a solution is globally optimal or not. For these reasons, it is better to reformulate nonlinear problems as linear or quadratic when possible.

When a model is linear, quadratic, or quadratically constrained, **DefineOptimization()** analyzes the model and all the definitions contained within, and determines the coefficients for the objective and each constraint. Solver engines are able to apply special algorithms to these coefficient matrices, and avoid repeated evaluations and finite differencing of your model. However, once your problem is non-linear, the solver engines must repeatedly re-evaluate your model, often multiple times around a single search point in order to estimate gradients. All the engine can infer about the search space are the results of these evaluations. Hence, a solver of a non-linear problem has less information to work with, and thus has a much more difficult task to perform.

Continuous, integer, and mixed-integer programs

Each decision variable can be specified as ***continuous***, meaning it is a real number (between bounds if specified), as ***semi-continuous*** (a real number with bounds, or zero), as ***integer***, meaning a whole number, as ***binary*** or ***Boolean,*** meaning its values can be True (1) or False (0), as a member of an integer ***group***, where each member of the group must have a different integer value, or as one of a finite set of ***explicit discrete*** categorical labels. Optimization problems are classified as ***continuous***, meaning the decision variables are all continuous; ***integer***, meaning they are all integer, binary, or group variables; or ***mixed-integer*** if they are a mixture of continuous, semi-continuous, integer, binary, group, or discrete 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.

Solving simultaneous equations

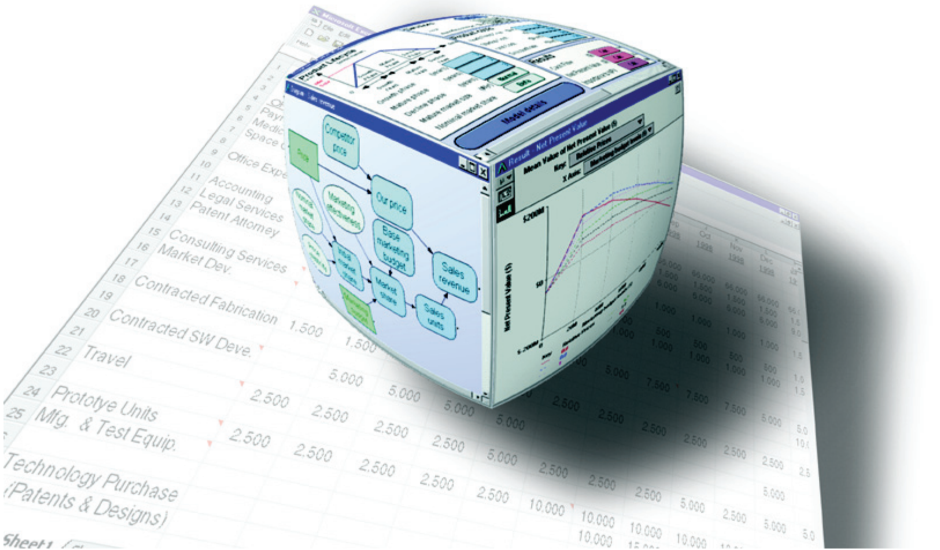
The Optimizer first attempts to find a feasible solution. If found, it then attempts to optimize within the set of feasible solutions. Thus, solving a set of simultaneous equations is a special case of the optimization problem, where each constraint has a sense of =, the objective is irrelevant (unless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

Chapter 3

Optimizing with Arrays

This chapter shows you how to:

- Optimize using Decision and Constraint arrays
- Identify indexes that are intrinsic to the optimization
- Enter intrinsic arrays in Intrinsic Indexes lists
- Use Analytica's Intelligent Array logic to set up multiple optimizations
- Refine a model by changing dimensions of the decision array



Arrays in optimization models and Array abstraction

Arrays in Optimization Problems

The textbook description of LP, QP and NLP formulations in the previous chapter depicted the set of decision variables as a one-dimensional vector, along with a linear vector of constraints. However, array-valued variables and array-valued constraints arise naturally in many optimization problems, and it is often more natural and convenient to formulate specific decision variables as multi-dimensional arrays, with dimensionality differing from decision to decision.

Structured optimization allows you to easily define array-valued decision variables, using the dimensionality that is natural to your problem. Additionally, since the variables appearing in a constraint expression may themselves be array-valued when computed, a single inequality expression may expand to be a multi-dimensional array of scalar constraints. Internally, structured optimization takes care of flattening and concatenating all these decision and constraint arrays for solution by the underlying solver engine so that you don't have to worry about it.

Array Abstraction

As experienced users know, array abstraction is an important feature of Analytica. Array abstraction allows an index to be added to the input of a computation, such that the original computation is carried out repeatedly for each new input value without having to alter the original computation. Array abstraction in Analytica is a univeral concept that derives its power from the fact that it applies at every level, from simple expressions all the way up to entire models.

Optimization models are no different. If we have a submodel that solves an optimization problem, we can vary the set of inputs across a new index and repeat the optimization repeatedly for a set of scenario combinations. Thus, array abstraction gives rise to arrays of optimization problems.

A necessary distinction comes about

Analytica with its Structured Optimization feature therefore provides both the ability to incorporate arrays *within* an optimization, as well as array abstract to obtain arrays *of* optimizations. This leads to a distinction, in which we can describe indexes as being either ***intrinsic*** or ***extrinsic***. Intrinsic indexes lead to arrays within an optimization problem, while extrinsic indexes lead to arrays of optimizations.

An *intrinsic* index is what other optimization environments would simply call an index. It makes all of its elements available to the optimizer during a single optimization run.

An *extrinsic* index is available for array abstraction in Analytica. If you start with model that performs a single optimization, and then add a new dimension to one of the input variables, Analytica will generally treat the new index as extrinsic and abstract over it. You will now have an array of optimizations corresponding to each element of the extrinsic index.

The inherent support for both types of arrays is a key differentiator of Analytica's Structured Optimization. In this chapter we explain how to control array handling in Analytica and work through a detailed example that handles indexes in both ways.

Tip Readers who are not already familiar with the basic concepts of array abstraction will benefit from reading Tutorial Chapter 5 and User Guide Chapter 12.

Narrative Examples

Here are some narrative examples to help clarify what it means for an optimization to handle an index intrinsically or extrinsically:

Breakfast of Champions

Suppose you have a model that decides on the best breakfast for you to eat every day of the wrestling season. Your objective is to win as many matches as possible. The model has an index of Days since there will be a breakfast for each day.

Extrinsic Decision Index

If the Days index is treated extrinsically, the model will perform a separate optimization for each day. If you have a match on a particular day, the model will probably advise you to eat a hearty breakfast to fuel your victory. On rest days, there is no match to win so the model won't care what you eat. Each optimization is completely oblivious of the existence of any day other than the one it is considering. In this scenario you will probably win your early matches but you also risk gaining weight and getting bumped up to a heavier class later in the season. The total number of matches you win throughout the season will not necessarily be optimized. If fact, it would be impossible for your model to optimize any objective that spans more than a single day.

Intrinsic Decision Index

If the Days index in intrinsic, the model will perform a single optimization that outputs your entire breakfast schedule for the season. The intrinsic character of the index is clear when you consider that each element of the array can influence outcomes across the entire index of Days.

Beer Distribution

Intrinsic Constraint Index

This chapter includes a detailed LP example in which there are multiple breweries shipping to multiple regions. Each brewery has a maximum capacity limit, and each region has a minimum demand to be met. In the example, we express capacity constraints for all breweries as one-dimensional array in the Supply Constraint node, using **Brewery** as the index for the array. We have a similar array of Demand constraints using **Region** as the index. The key to the problem is that all constraints must hold simultaneously for the solution to work. Therefore, Brewery and Region are intrinsic indexes. If they were extrinsic instead, we would have an array of optimizations; one for each combination of Brewery and Region. Each optimization would consider a scenario where only a single combination of constraints would apply. For example, “What if the Fairfield brewery is limited to 600k cases of beer, and the Western region must receive at least 500k, but no other supply or demand constraints apply?” Answering a series of these strange questions would not be useful to us.

Extrinsic Index:

To demonstrate the role of an extrinsic index in the Beer Distribution example, we imagine that regional beer demand will depend on the winner of the World Series. The index titled World Series Winner includes two mutually exclusive scenarios for which we want to perform separate optimizations. This is an example of an extrinsic index available for array abstraction. While searching for the optimum beer distribution solution for a particular scenario, it is not necessary for the optimizer to know that the other scenario exists.

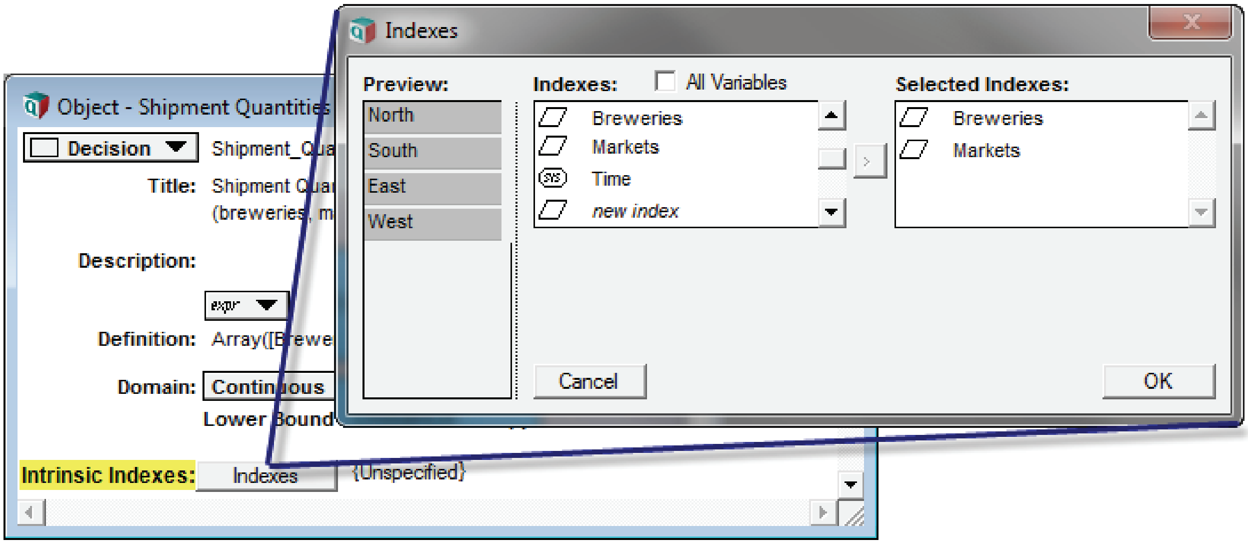
Tip

If you are not sure about the status of an index in an optimization, ask “Does the optimizer need to consider every element of this index simultaneously in order to find the correct solution?” If the answer is YES, the index is intrinsic to the optimization.

Setting the Intrinsic Index Attribute

Decision and Constraint nodes contain a special attribute called **Intrinsic Indexes**. Populating this list will ensure that Analytica Optimizer interprets the array as intended. The process is similar to the way you identify indexes when defining a table.

To see how this works, create a new Decision or Constraint node and double-click it to open the Object Window. Press the **Indexes** button next to **Intrinsic Indexes**. This opens the index selection window. Select from the indexes list on the left and press the transfer button to move them to the **Selected Indexes** window on the right. Click **OK**.



Analytica inferd Intrinsic Indexes

You can leave the **Intrinsic Indexes** attribute *unspecified* in either decisions and constraints, in which case **DefineOptimization()** infers the intrinsic indexes from the problem formulation. Whenever your intention is different from what these heuristics infer, you will need to specify the intrinsic indexes explicitly. An indexes that appear only in Constraints, but not in any Decision, is inherently ambiguous, and usually results in a meaningful (but different) optimization problem whether it is intrinsic or extrinsic. Analytica will assume such indexes are extrinsic, so if you want them to be intrinsic you must specify them explicitly in the Constraint's **intrinsic indexes** attribute.

Tip It is good practice to always populate **Intrinsic Index** lists in all Decisions and Constraints used in an optimization problem.

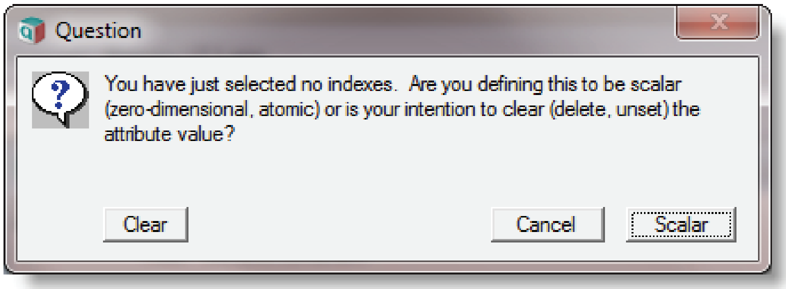
Extrinsic Indexes

Whenever the optimizer does not need to consider the index as a whole --as when considering only a single index element-- the index is *extrinsic* to the optimization. If a solution array has an extrinsic dimension it displays the results of multiple independent optimization runs along the extrinsic dimension. This is simply the concept of array abstraction applied to optimizations.

Extrinsic by omission Extrinsic indexes are never explicitly specified — an index is extrinsic when it is present but not intrinsic. The intrinsic indexes attribute for a Decision variable is inclusive and exhaustive, which means that every index listed is always included as an intrinsic index, and only those listed are intrinsic indexes of the decision. The intrinsic indexes attribute of a constraint is inclusive but not exhaustive. Any index listed there will become an intrinsic index for the constraint, but if other indexes are present that are already intrinsic elsewhere in the optimization, they too will be intrinsic to the constraint. The non-exhaustive nature of the intrinsic index specification allows optimization models to array abstract consistently.

What if all dimensions of an array are extrinsic?

In some cases you may want to explicitly tell the optimizer that all indexes of the Decision are extrinsic. This means that you intend for the optimizer to treat the Decision as a *scalar* value. To do this, Open the **Intrinsic Indexes** attribute and click **OK** without selecting any indexes. Analytica displays a dialog box asking you to confirm whether you want to designate the array as **Scalar** or leave the list **Unspecified**.



You have just selected no indexes. Are you defining this to be scalar (zero-dimensional, atomic) or is your intention to clear (delete, unset) the attribute value?

Example 1: Beer Distribution LP, Base Case

Model Description

We start with an adaptation of a classic Linear Programming (LP) example: A large brewing company operates five breweries nationwide. Each brewery distributes product among four regions. Routes include all combinations of Breweries and Market regions. The challenge is to find the shipping pattern that minimizes distribution cost while a) meeting demand in each region, and b) observing production limits at each brewery. We assume that distribution cost per case of beer is proportional to the distance between the brewery and the region.

Setting up the Model

To explore and follow this example in Analytica, find the **Beer Distribution LP 1.ana** model in the Optimizer Examples folder.

Brewery and Market (indexes)

There are five **Breweries** and four **Market** regions:

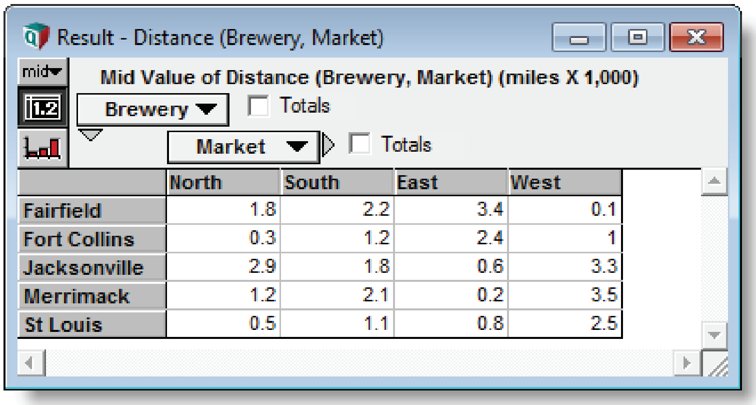
```
Index Brewery :=
   ['Fairfield','Fort Collins','Jacksonville','Merrimack','St Louis']

Index Market := ['North','South','East','West']
```

Distance (input array)

Distance between breweries and markets is represented as a table dimensioned by **Brewery** and **Market**. Units are thousands of miles:

```
Variable Distance :=
   Table(Brewery,Market)(1.8,2.2,3.4,0.1,0.3,1.2,2.4,1,2.9,1.8,0.6,3.3,
   1.2,2.1,0.2,3.5,0.5,1.1,0.8,2.5)
```



Result - Distance (Brewery, Market)

Mid Value of Distance (Brewery, Market) (miles X 1,000)

	North	South	East	West
Fairfield	1.8	2.2	3.4	0.1
Fort Collins	0.3	1.2	2.4	1
Jacksonville	2.9	1.8	0.6	3.3
Merrimack	1.2	2.1	0.2	3.5
St Louis	0.5	1.1	0.8	2.5

Freight Price (input scalar value)

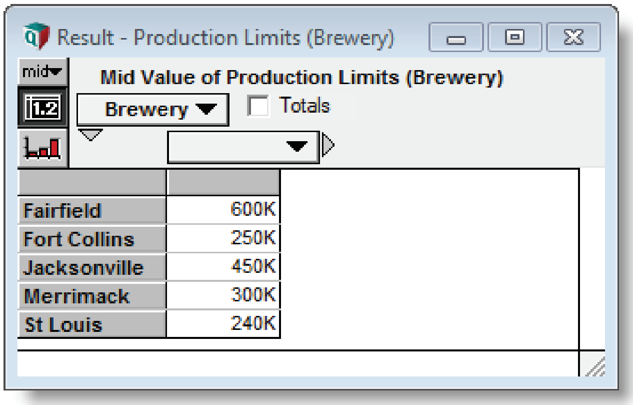
Freight Price is a simple scalar value of $5 per 1,000 miles per case of beer.

```
Variable Freight_price := 5
```

Production Limits (input array)

Production Limits indicate maximum capacity for each brewery in cases of beer:

```
Variable Production_limits :=
Table(Brewery)(600K,250K,450K,300K,240K)
```



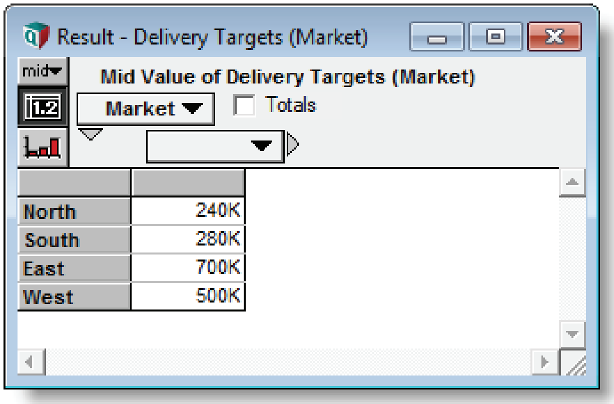
Result - Production Limits (Brewery)
mid
Mid Value of Production Limits (Brewery)
Brewery
□ Totals

Fairfield	600K
Fort Collins	250K
Jacksonville	450K
Merrimack	300K
St Louis	240K

Delivery Targets (input array)

Delivery Targets indicate the minimum quota that each Market region must receive:

```
Variable Delivery_target := Table(Market)(240K,280K,700K,500K)
```



Result - Delivery Targets (Market)
mid
Mid Value of Delivery Targets (Market)
Market
□ Totals

North	240K
South	280K
East	700K
West	500K

Distribution Cost (intermediate array)

Distribution Cost per Case is Freight Price multiplied by Distance:

```
Variable Dist_per_case := Freight_price * Distance
```

Shipment Quantities (input decision array)

The input decision array represents **Shipment Quantities** from each brewery to each region. It is a two-dimensional array indexed by **Brewery** and **Market**. The non-optimized values are not used anywhere else in the model so we can simply insert 1 as a dummy value across the array.
(Initial guesses do not apply since this is a Linear Program.)

Units are cases of beer.

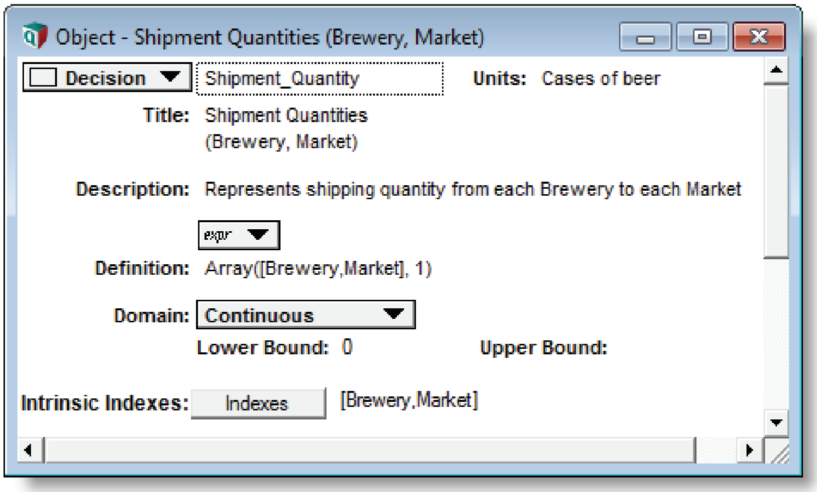
Staying Positive

We set the Lower Bound attribute to zero to disallow negative shipping values, along with the undesirable implication of turning perfectly good beer back into barley and hops.

Set Brewery and Market as intrinsic indexes

Finally and perhaps most importantly, we consider the index status for this array. The optimized solution should be in the same form as this table. It should also be integrated such that every array element has simultaneous influence over all other elements. **Brewery** and **Market** are both *intrinsic* indexes in this case.

```
Decision Shipment_Quantity := Array([Brewery, Market],1)
Shipment_Quantity attribute Lower Bound := 0
Shipment_Quantity attribute Intrinsic Indexes := [Brewery, Market]
```



Shipment_Quantity

Units: Cases of beer

Title: Shipment Quantities (Brewery, Market)

Description: Represents shipping quantity from each Brewery to each Market

Definition: Array([Brewery,Market], 1)

Domain: Continuous

Lower Bound: 0

Upper Bound:

Intrinsic Indexes: [Brewery,Market]

Total Distribution (Objective)

Our goal is to minimize the total distribution cost. For each route, distribution cost is the **Distribution Cost per Case** multiplied by the **Shipment Quantity** for each route. The final objective is the sum for all routes:

```
Objective Total_dist_cost :=
   Sum(Shipment_Quantity * Dist_per_case, Brewery, Market)
```

Summing over **Brewery** and **Market** makes the objective a scalar value. This is a necessary condition for the optimization.

Tip

Every minimizing or maximizing optimization is based on a scalar objective value. In other words, intrinsic dimensions are not allowed in the Objective. If the Objective is not scalar, Analytica assumes the dimensions are extrinsic and performs independent optimizations for each scalar element in the Objective array.

(For some optimization problems, the challenge is simply to find a solution that satisfies all constraints. This type of optimization has no objective at all.)

Supply Constraint

The **Supply Constraint** ensures that shipment quantities from a single brewery to all available markets do not exceed the production limit for the brewery.

The **Production Limit** array is already dimensioned by **Brewery**. Since this array is an input to **Supply Constraint**, we expect **Brewery** to be a dimension of **Supply Constraint** as well, even though we don't explicitly mention the index in the defining expression. This is the basic principle of array abstraction in Analytica.

The **Supply Constraint** array actually defines five constraints; one for each **Brewery**.

Set Brewery as an intrinsic index

Is **Brewery** an intrinsic index for the **Supply Constraint** array? The answer is YES because we need to enforce the respective Supply Constraints on all breweries simultaneously. Make sure to include **Brewery** in the Intrinsic Indexes list for the **Supply Constraint** node.

```
Constraint Supply_constraint :=
    Sum(Shipment_Quantity, Market)<=Production_Limits
```

```
Supply_constraint attribute Intrinsic Indexes := [Brewery]
```

Is **Market** an intrinsic index for the **Supply Constraint**? The answer is NO because **Market** is not a dimension of the array at all. The Sum() function eliminates it, leaving the array with only one dimension.

Demand Constraint

The **Demand Constraint** ensures that each market region receives a total quantity from all breweries greater than or equal to the market demand. The input array **Delivery Targets** is dimensioned by **Market**, and therefore the **Demand Constraint** will also have this dimension.

The **Demand Constraint** array defines four constraints; one for each **Market**.

Set Market as an intrinsic index

Market is an intrinsic index for the Demand Constraint because the solution should meet quotas for all markets. Make sure to include **Market** in the Intrinsic Indexes list for the **Demand Constraint** node.

```
Constraint Demand_constraint :=
    Sum(Shipment_Quantity, Brewery) >= Delivery_Target
Demand_constraint attribute Intrinsic Indexes := [Market]
```

The Sum() function eliminates **Brewery** from the array.

Optimization Node

Remembering required attributes of the DefineOptimization() function as described in Chapter 1, we need to identify Decisions, Constraints, and the Objective to be minimized/maximized.

```
Variable Optimization := DefineOptimization
    (Decision:Shipment_Quantity,
    Constraints:Supply_Constraint, Demand_Constraint,
    Minimize: Total_Dist_Cost)
```

Solution Node

We obtain the solution using the OptSolution() function. The first parameter identifies the optimization node. The second parameter (optional) identifies a specific Decision for which the solution should be represented.

```
Decision Optimized_Solution :=
    OptSolution(Optimization, Shipment_Quantity)
```

Optimized Objective

The OptObjective() function calculates the minimized or maximized quantity. In this case, it represents the total distribution cost.

```
Variable Optimized_Objective := OptObjective(Optimization)
```

Status

Status text will confirm that the optimizer has found a solution

```
Variable Status := OptStatusText(Optimization)
```

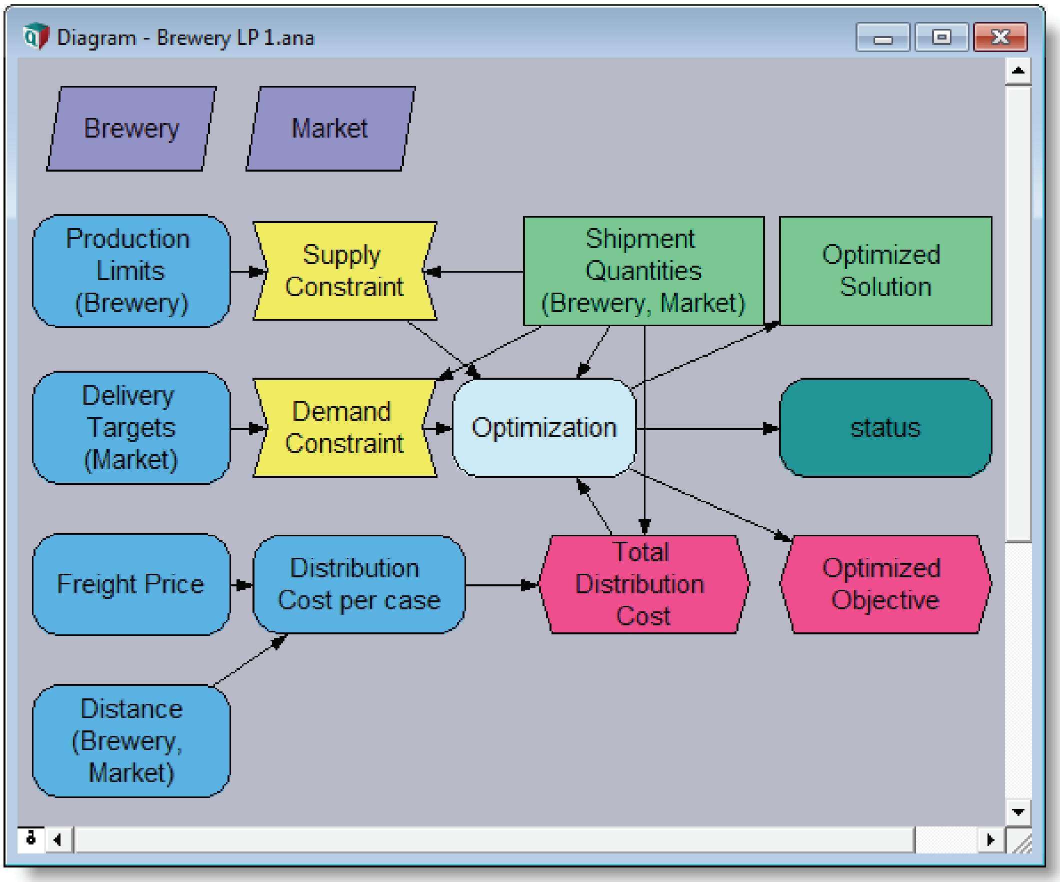


Diagram - Brewery LP 1.ana
Brewery
Market
Production Limits (Brewery)
Supply Constraint
Shipment Quantities (Brewery, Market)
Optimized Solution
Delivery Targets (Market)
Demand Constraint
Optimization
status
Freight Price
Distribution Cost per case
Total Distribution Cost
Optimized Objective
Distance (Brewery, Market)

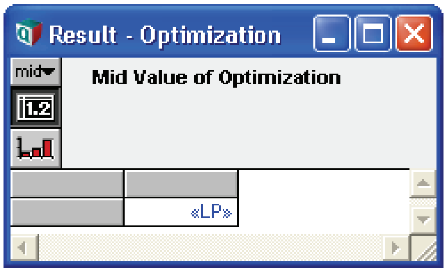
Verifying the Optimization Setup

Evaluate the Optimization node

It is always a good idea to evaluate the Optimization node to confirm that the optimization is interpreting your problem as intended. The DefineOptimization() function evaluates as a special symbol, or array of symbols, indicating the type of optimization problem presented. In this case, **«LP»** confirms that this is a Linear Program as expected. Furthermore, the result shows only a single **«LP»** symbol. This confirms that the solution is the result of a single optimization run.

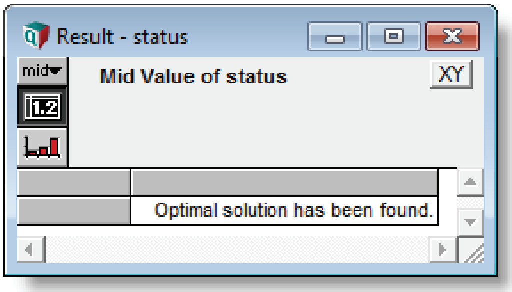
Evaluate Status

The **OptStatusText()** function confirms that the optimizer has found a feasible solution satisfying all constraints.


Result - Optimization

Mid Value of Optimization

«LP»


Result - status

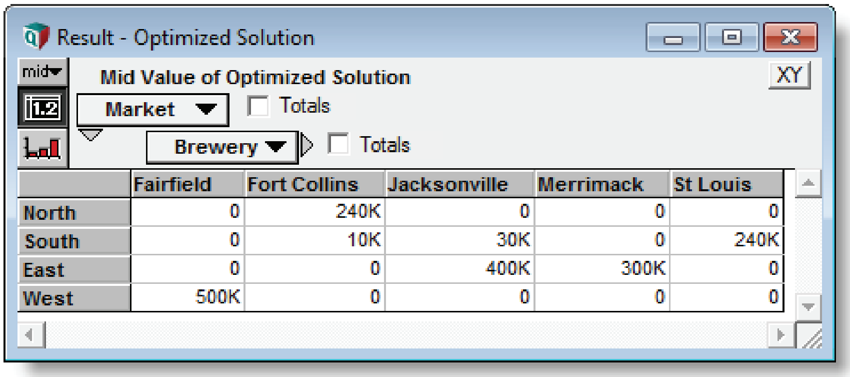
Mid Value of status

Optimal solution has been found.

Checking the Result

Optimized Solution as an Array

Evaluate the Optimized Solution array to check the final result.



	Fairfield	Fort Collins	Jacksonville	Merrimack	St Louis
North	0	240K	0	0	0
South	0	10K	30K	0	240K
East	0	0	400K	300K	0
West	500K	0	0	0	0

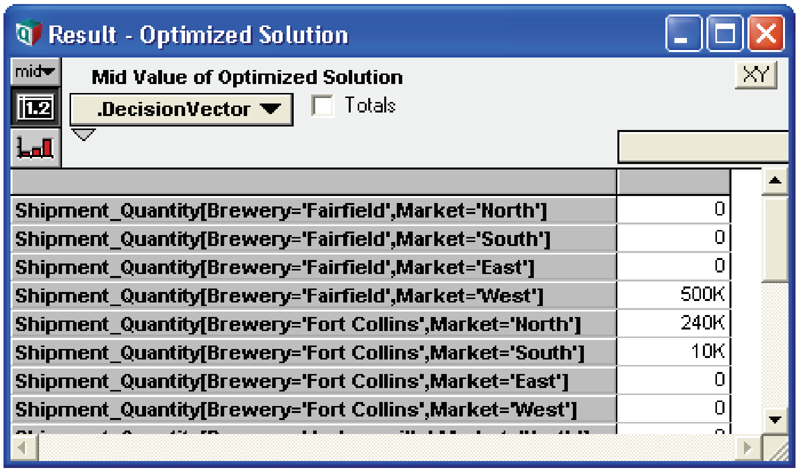
In this example we used the optional second parameter of the OptSolution() function to reference the **Shipment Quantities** array. This formats the output to match the same dimensions as the Decision input array. It is a two-dimensional table indexed by **Market** and **Brewery**.

Optimized Solution as a List

If you omit the second parameter of OptSolution(), Analytica creates a local index called **.DecisionVector** and presents the result as a one-dimensional array. You can experiment with this output format by removing the second parameter from the OptSolution() function.

```
Variable Optimized_solution := OptSolution(Optimization)
```

.



Shipment_Quantity[Brewery='Fairfield',Market='North']	0
Shipment_Quantity[Brewery='Fairfield',Market='South']	0
Shipment_Quantity[Brewery='Fairfield',Market='East']	0
Shipment_Quantity[Brewery='Fairfield',Market='West']	500K
Shipment_Quantity[Brewery='Fort Collins',Market='North']	240K
Shipment_Quantity[Brewery='Fort Collins',Market='South']	10K
Shipment_Quantity[Brewery='Fort Collins',Market='East']	0
Shipment_Quantity[Brewery='Fort Collins',Market='West']	0

Summary: Basic Beer Distribution LP

The basic beer distribution example demonstrates a simple Linear Program with a two-dimensional decision array. There are two one-dimensional Constraint arrays, each over a different index. The model has only two indexes overall: **Brewery** and **Market**. Both indexes are intrinsic to the optimization. The example does not include any extrinsic indexes.

Example 2: Beer Distribution with Added Scenario

Model Description Let's assume that beer demand in Western and Southern markets will depend on the winner of the 2010 World Series. Notwithstanding those inclined to drown their sorrows, we assume that people in the winning market will drink more, while those in the losing market will drink less.

To show this in the model, we add a new index titled **World Series Winner** and add this new dimension to the **Delivery Targets** array. Then we edit the values in the array to reflect the new assumptions. No other adjustments are necessary.

Combining Optimization with Intelligent Arrays

Now that the model is clear on which indexes are intrinsic, you are free to add extrinsic indexes and rely on Analytica's Intelligent Array abstraction features just as you would in any other situation. In this example we add a new dimension that propagates from optimization inputs all the way to the solution array, driving multiple optimization runs along the way. This demonstrates how easy it is to combine optimization with Analytica's Intelligent Arrays.

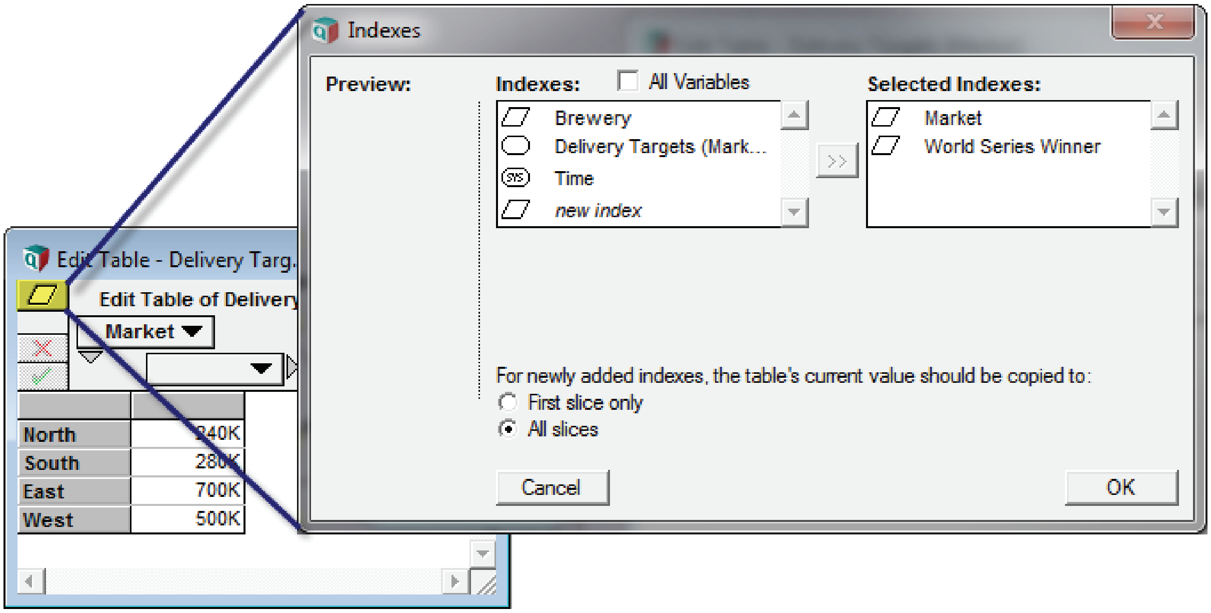
Setting Up the Model

To explore and follow this example in Analytica, find the **Beer Distribution LP 2.ana** model in the Optimizer Examples folder. In this section we start with the model from Example 1, detailing changes only.

World Series Winner Add a new index for the **World Series Winner**

```
Index World_Series_Winner :=
   ['Giants win World Series', 'Rangers win World Series']
```

Edit Delivery Targets Array To add this new index to the **Delivery Targets** array, open the edit table and press the Index selection button in the upper left corner. Add **World Series Winner** to the list of indexes for this array.



Indexes

Preview:

Indexes: □ All Variables

- Brewery
- Delivery Targets (Mark...
- Time
- *new index*

Selected Indexes:

- Market
- World Series Winner

For newly added indexes, the table's current value should be copied to:

○ First slice only
● All slices

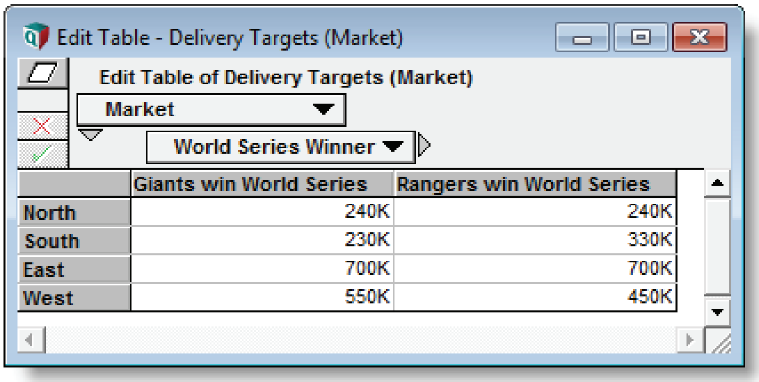
Edit Table - Delivery Targ...

Edit Table of Delivery

Market

North	240K
South	280K
East	700K
West	500K

Edit the two-dimensional table to match the values shown here:



Edit Table - Delivery Targets (Market)

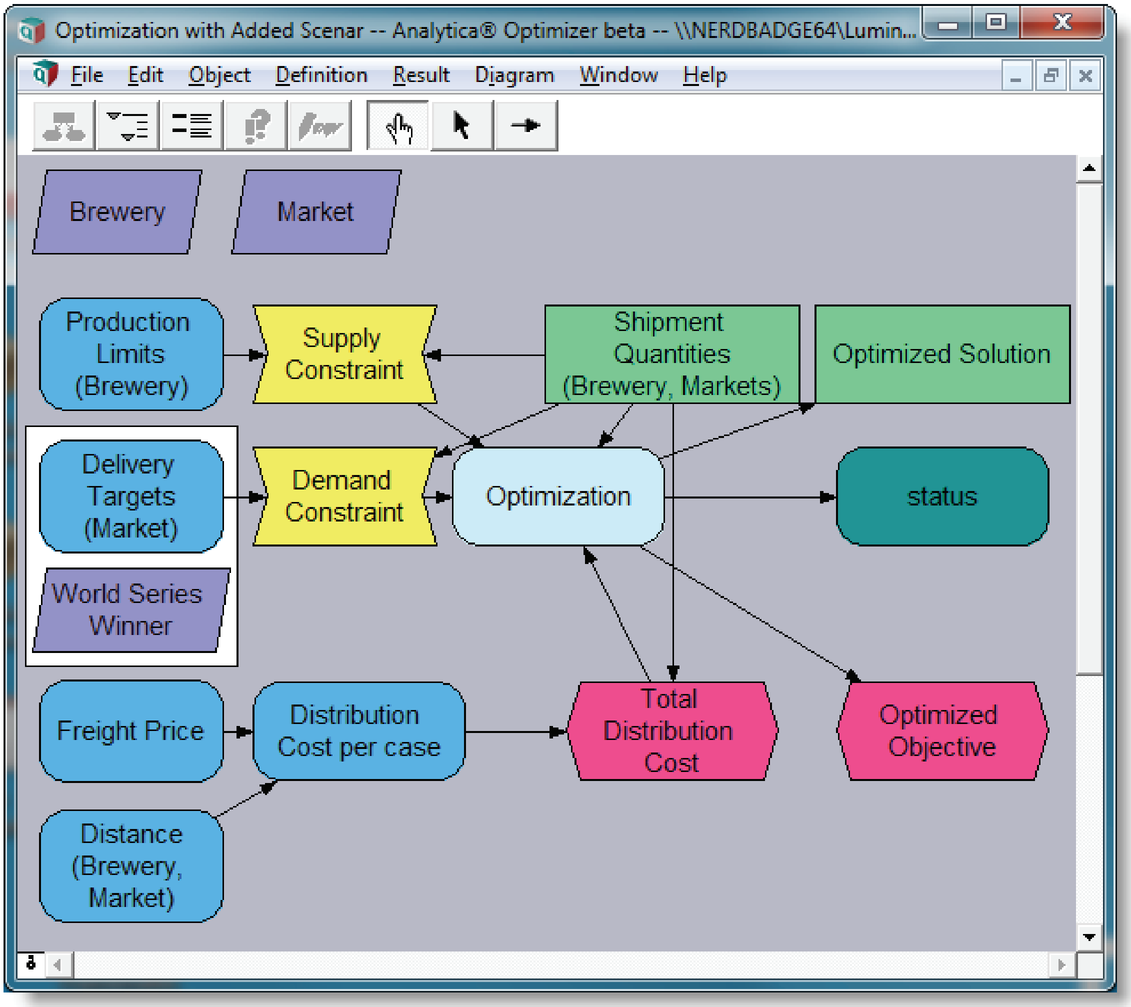
Edit Table of Delivery Targets (Market)

Market

World Series Winner

	Giants win World Series	Rangers win World Series
North	240K	240K
South	230K	330K
East	700K	700K
West	550K	450K

```
Variable Delivery_Targets := Table(Market,World_Series_Winner)
    (240K, 240K ,230K, 330K ,700K, 700K ,550K, 450K)
```

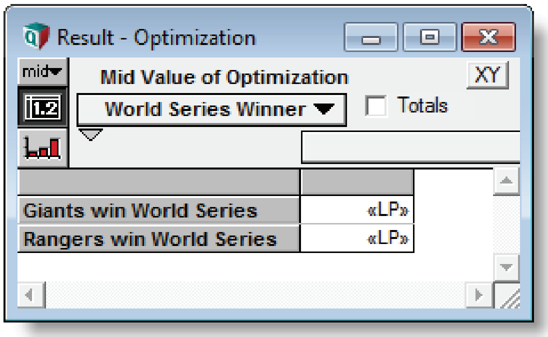

Optimization with Added Scenar -- Analytica® Optimizer beta -- \\NERDBADGE64\Lumin...
File
Edit
Object
Definition
Result
Diagram
Window
Help
Brewery
Market
Production Limits (Brewery)
Supply Constraint
Shipment Quantities (Brewery, Markets)
Optimized Solution
Delivery Targets (Market)
Demand Constraint
Optimization
status
World Series Winner
Freight Price
Distribution Cost per case
Total Distribution Cost
Optimized Objective
Distance (Brewery, Market)

Verifying the Optimization Setup

Evaluate the Optimization node and check status

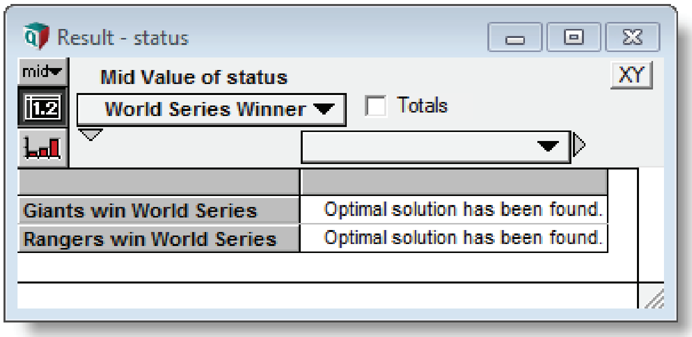
The DefineOptimization() function now evaluates to an array, telling you that the optimizer is performing independent optimizations for each World Series scenario.

The status text reports the status of each optimization individually.



Mid Value of Optimization

Giants win World Series	«LP»
Rangers win World Series	«LP»

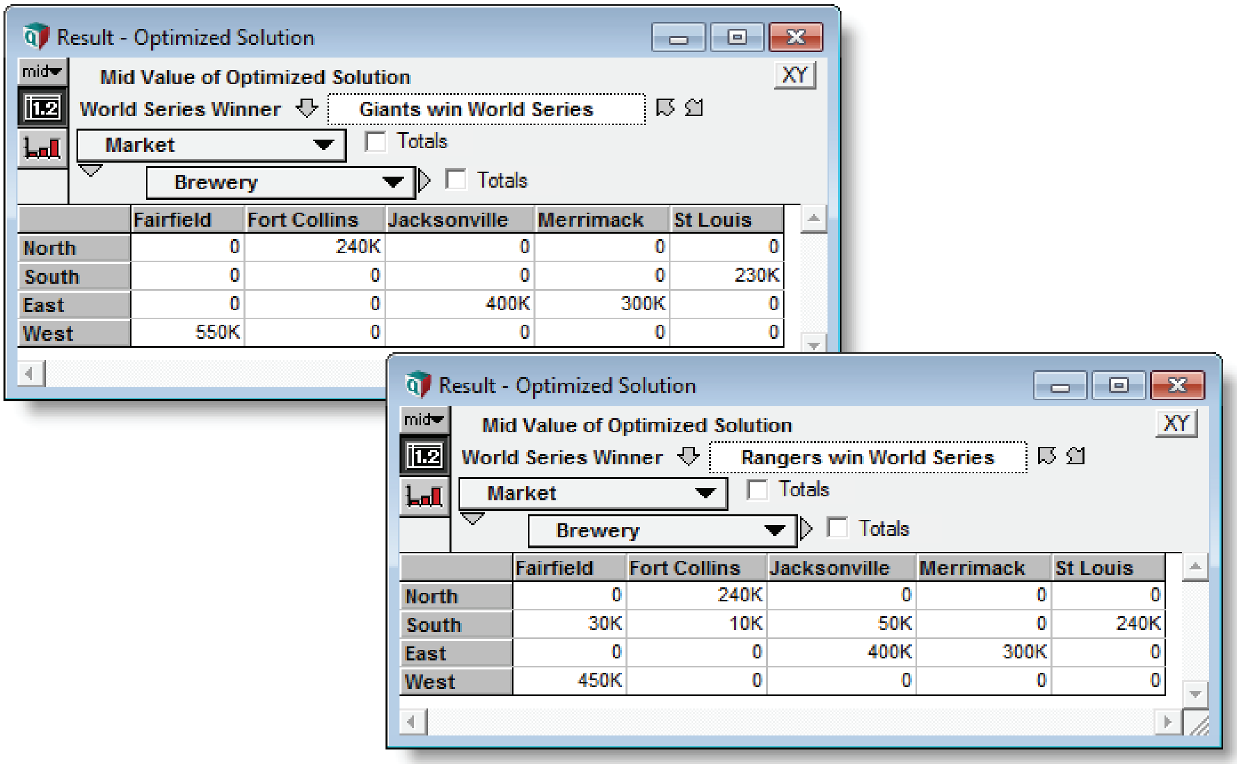


Mid Value of status

Giants win World Series	Optimal solution has been found.
Rangers win World Series	Optimal solution has been found.

Checking the result

Evaluate the Optimized Solution array to check the solution.



Mid Value of Optimized Solution

World Series Winner: Giants win World Series

	Fairfield	Fort Collins	Jacksonville	Merrimack	St Louis
North	0	240K	0	0	0
South	0	0	0	0	230K
East	0	0	400K	300K	0
West	550K	0	0	0	0

Mid Value of Optimized Solution

World Series Winner: Rangers win World Series

	Fairfield	Fort Collins	Jacksonville	Merrimack	St Louis
North	0	240K	0	0	0
South	30K	10K	50K	0	240K
East	0	0	400K	300K	0
West	450K	0	0	0	0

The result is a three-dimensional array including both intrinsic and extrinsic dimensions.You can pivot the output to any desired configuration.

Summary: Beer Distribution LP with Added Scenario

In this example, we started with the basic model and added a new dimension to the **Delivery Targets** input array. According to the basic rules of array abstraction, Analytica adds the new dimension to all arrays influenced by **Delivery Targets**. These include **Demand Constraint**, **Optimization**, **Optimized Solution**, **Optimized Objective** and **Status** values. Adding an extrinsic dimension to the Optimization node drives separate optimization runs for each element of the extrinsic array.

The solution output combines intrinsic and extrinsic dimensions, representing them all in a combined array. In this example, the dimensions of the solution array are **Brewery**, **Market**, and **World Series Winner**.

Example 3: Beer Distribution with Limited Routes

Model Description

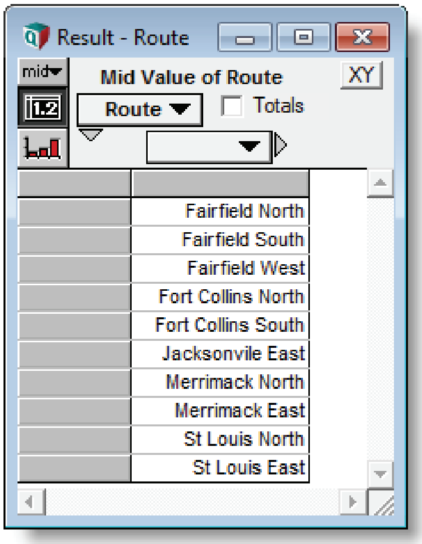
In the previous examples, we allowed all five Breweries to ship to all four Markets, presenting twenty possible routes. In this example we confine the shipments to ten chosen routes and disallow the rest. Such restrictions can be cumbersome in some optimizing environments. In Analytica, it is a simple matter of changing the dimension of the decision array.

Setting Up the Model

To explore and follow this example in Analytica, find the **Beer Distribution LP 3.ana** model in the Optimizer Examples folder. In this section we start with the model from Example 2, detailing changes only.

Create a Route Index

First we create an index of approved routes. These are specific combinations of **Brewery** and **Market**.



Mid Value of Route

Fairfield North
Fairfield South
Fairfield West
Fort Collins North
Fort Collins South
Jacksonvile East
Merrimack North
Merrimack East
St Louis North
St Louis East

```
Index Route :=
    ['Fairfield North','Fairfield South','Fairfield West',
    'Fort Collins North','Fort Collins South',
    'Jacksonvile East',
    'Merrimack North','Merrimack East',
    'St Louis North','St Louis East']
```

Re-define the Decision input array

In previous examples we defined the Decision input as a two-dimensional array. In this example we re-define it to be a one-dimensional array along the index of approved routes. Once again, we can insert a dummy value of 1 for the non-optimized quantities since they are not used anywhere else in the model.

Edit the Intrinsic Indexes list

Make sure to also edit the Intrinsic Indexes list. Remove **Brewery** and **Market**, and add **Route** to the list.

```
Variable Shipment_Quantity := Array(Route, 1)
Shipment_Quantity attribute Intrinsic Indexes := [Route]
```

Re-define the Objective and its inputs

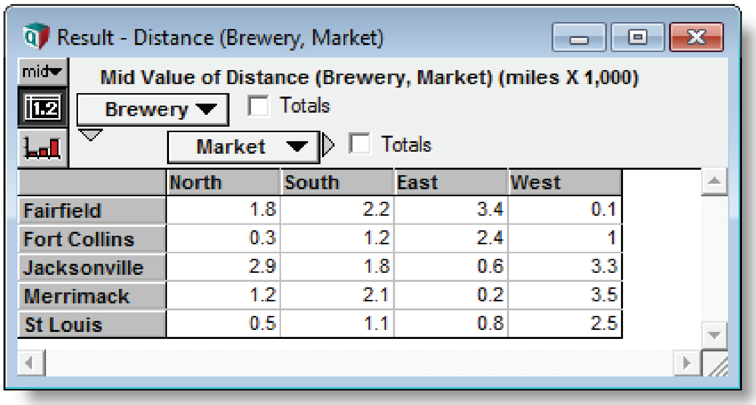
The Objective representing **Total Distribution Cost** should now sum over the **Route** index instead of **Brewery** and **Market**.

```
Objective Total_Dist_Cost := Sum(Shipment_Quantity * Dist_per_case,
   Route)
```

All inputs to **Total Distribution Cost** should be also dimensioned by **Route**. We re-define the **Distance** array using corresponding values from the original table.

```
Variable Distance :=
   Table(Route)(1.8,2.2,0.1,0.3,1.2,0.6,1.2,0.2,0.5,0.8)
```

Original Distance Array



Result - Distance (Brewery, Market)

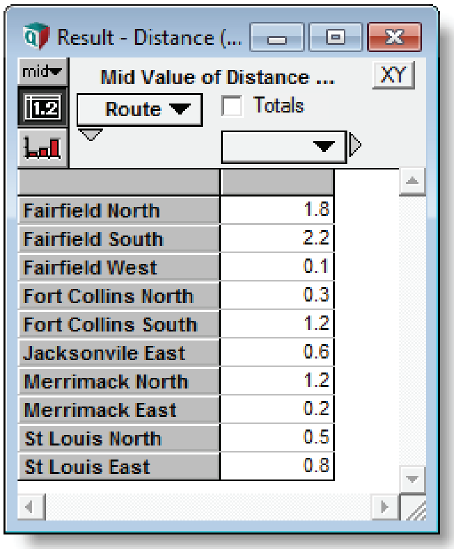
Mid Value of Distance (Brewery, Market) (miles X 1,000)

Brewery ▼ □ Totals

Market ▼ □ Totals

	North	South	East	West
Fairfield	1.8	2.2	3.4	0.1
Fort Collins	0.3	1.2	2.4	1
Jacksonville	2.9	1.8	0.6	3.3
Merrimack	1.2	2.1	0.2	3.5
St Louis	0.5	1.1	0.8	2.5

Re-defined Distance Array



Result - Distance (...

Mid Value of Distance ...

XY

Route ▼ □ Totals

Fairfield North	1.8
Fairfield South	2.2
Fairfield West	0.1
Fort Collins North	0.3
Fort Collins South	1.2
Jacksonvile East	0.6
Merrimack North	1.2
Merrimack East	0.2
St Louis North	0.5
St Louis East	0.8

Creating Index Maps

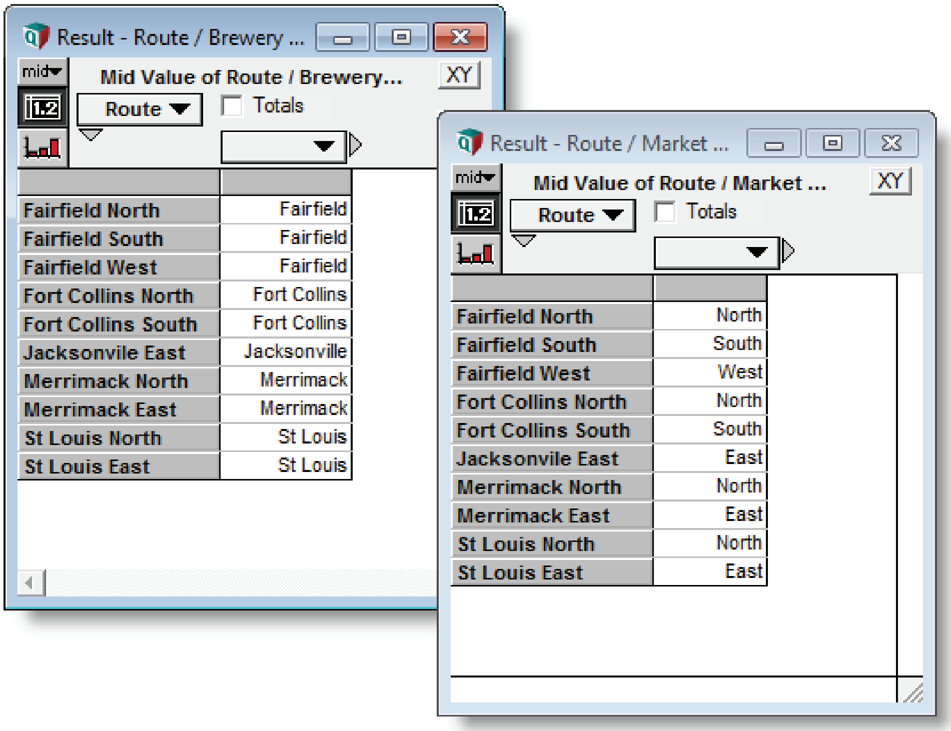
Constraint arrays do not change indexes in the new model. **Production Limits** still apply to **Breweries,** and **Delivery Targets** still apply to **Markets**. But the **Shipment Quantities** array no longer contains these indexes. To solve this dilemma, we start by making index maps relating the **Route** index to **Brewery** and **Market**. An index map is a one-dimensional array that associates a coarse index with a fine index. The map uses the fine index as its dimension and lists corresponding elements of the coarse index.

Create a new variable titled **Route / Brewery map**.

```
Variable Route_Brewery_map := Table(Route)
    ('Fairfield', 'Fairfield', 'Fairfield',
    'Fort Collins', 'Fort Collins',
    'Jacksonville',
    'Merrimack', 'Merrimack',
    'St Louis', 'St Louis' )
```

Create a new variable titled **Route / Market map**.

```
Variable Route_Market_map :=Table(Route)
    ('North','South','West','North','South',
    'East','North','East','North','East')
```



Result - Route / Brewery ...
mid
Mid Value of Route / Brewery...
XY
Route
□ Totals

Fairfield North	Fairfield
Fairfield South	Fairfield
Fairfield West	Fairfield
Fort Collins North	Fort Collins
Fort Collins South	Fort Collins
Jacksonvile East	Jacksonville
Merrimack North	Merrimack
Merrimack East	Merrimack
St Louis North	St Louis
St Louis East	St Louis

Result - Route / Market ...
mid
Mid Value of Route / Market ...
XY
Route
□ Totals

Fairfield North	North
Fairfield South	South
Fairfield West	West
Fort Collins North	North
Fort Collins South	South
Jacksonvile East	East
Merrimack North	North
Merrimack East	East
St Louis North	North
St Louis East	East

Using Aggregate()

The Aggregate() function can use these maps to create aggregated sums. This allows us to convert **Shipment Quantities** from the fine index (**Route**) to the coarse target indexes (**Brewery** or **Market**).

The basic syntax of the Aggregate() Function is:

```
Aggregate(input_array, map, fine_index, target_index)
```

Aggregate Shipment Quantities

Now we can create suitable inputs for the Constraint nodes.

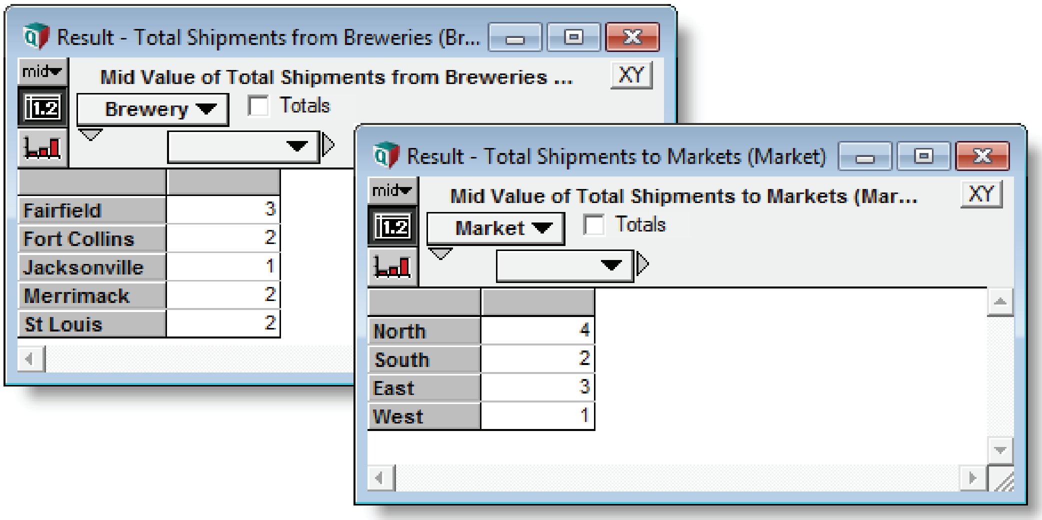
Create a new variable titled **Total Shipments from Breweries**

```
Variable Ship_from_Brewery :=
   Aggregate(Shipment_Quantity, Route_Brewery_map, Route, Brewery)
```

Create a new variable titled **Total Shipments to Markets**

```
Variable Ship_to_Market :=
   Aggregate(Shipment_Quantity, Route_Market_map, Route, Market)
```

You can evaluate the aggregated arrays to verify that they represent **Shipment Quantities** aggregated by **Brewery** and **Market**. Input Quantities are 1 for each **Route**, so these aggregated sums represent the number of routes associated with each **Brewery** or **Market**.


Result - Total Shipments from Breweries (Br...
mid
Mid Value of Total Shipments from Breweries ...
XY
Brewery
□ Totals
Result - Total Shipments to Markets (Market)
Mid Value of Total Shipments to Markets (Mar...
Market

Fairfield	3
Fort Collins	2
Jacksonville	1
Merrimack	2
St Louis	2

North	4
South	2
East	3
West	1

Edit Constraint Inputs

Edit the **Supply Constraint** to use **Total Shipments from Breweries** as an input. **Brewery** is still an intrinsic index for this node.

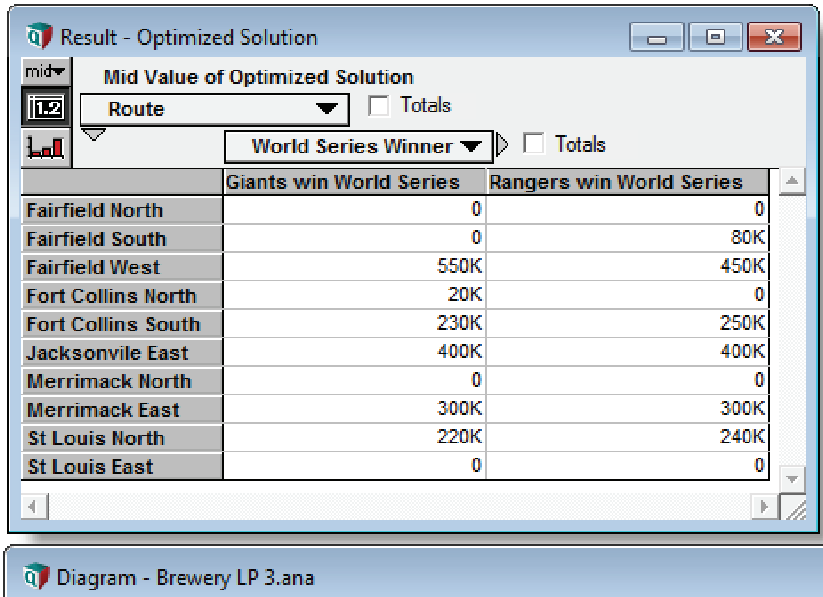
```
Constraint Supply_Constraint := Ship_from_brewery<=Production_Limits
Supply Constraint attribute Intrinsic Indexes := [Brewery]
```

Edit the **Demand Constraint** to use **Total Shipments to Markets** as an input. **Market** is still an intrinsic index for this node.

```
Constraint Demand_Constraint := Ship_to_market >= Delivery_Target
Demand Constraint attribute Intrinsic Indexes := [Market]
```

Checking the Result

Evaluate the **Optimized Solution** array to see the result for the limited route example. This is now a two-dimensional array indexed by the intrinsic **Route** index and the extrinsic **World Series Winner** index.



Result - Optimized Solution

mid

Mid Value of Optimized Solution

Route ☐ Totals

World Series Winner ☐ Totals

	Giants win World Series	Rangers win World Series
Fairfield North	0	0
Fairfield South	0	80K
Fairfield West	550K	450K
Fort Collins North	20K	0
Fort Collins South	230K	250K
Jacksonvile East	400K	400K
Merrimack North	0	0
Merrimack East	300K	300K
St Louis North	220K	240K
St Louis East	0	0

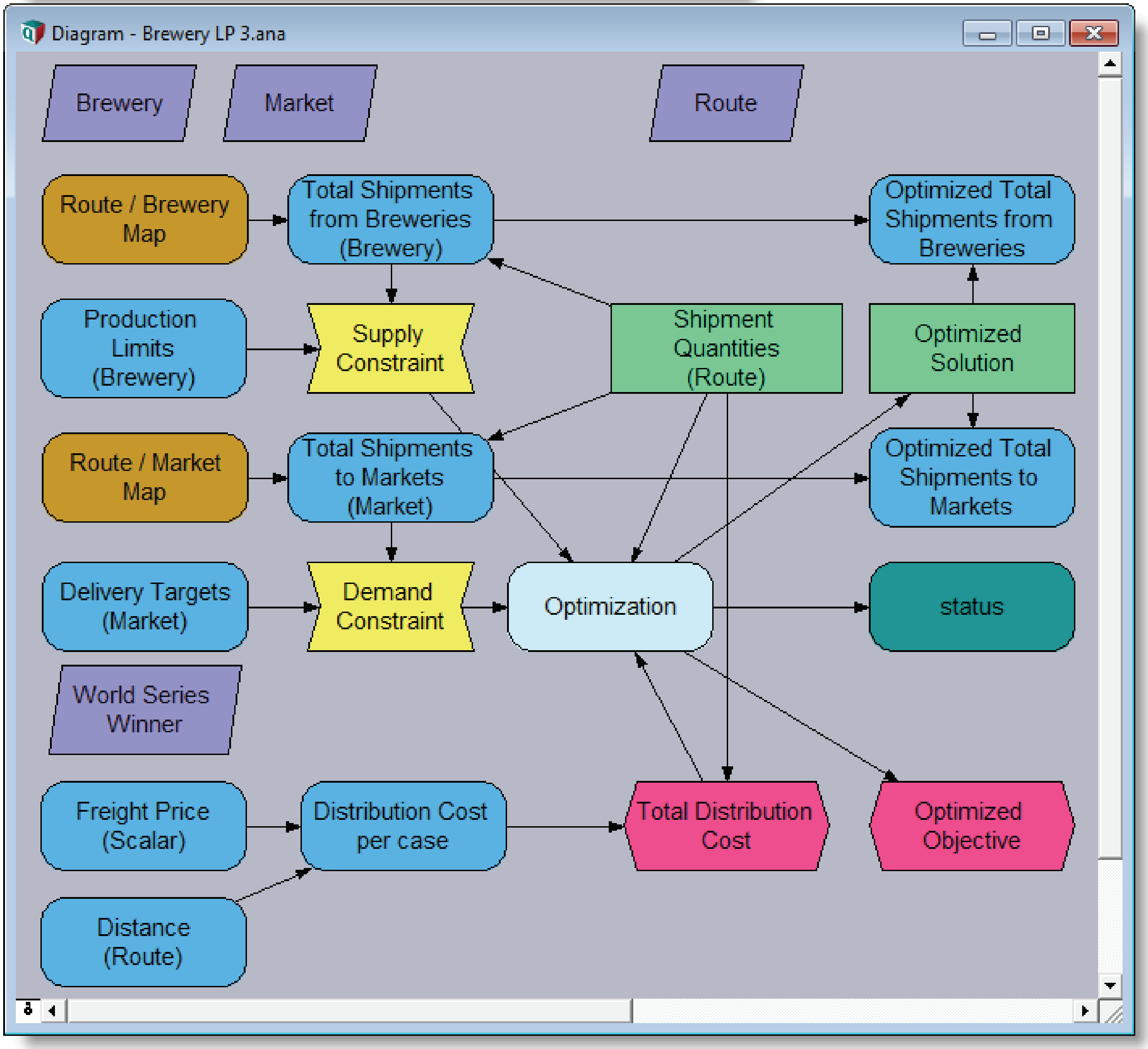


Diagram - Brewery LP 3.ana
Brewery
Market
Route
Route / Brewery Map
Total Shipments from Breweries (Brewery)
Optimized Total Shipments from Breweries
Production Limits (Brewery)
Supply Constraint
Shipment Quantities (Route)
Optimized Solution
Route / Market Map
Total Shipments to Markets (Market)
Optimized Total Shipments to Markets
Delivery Targets (Market)
Demand Constraint
Optimization
status
World Series Winner
Freight Price (Scalar)
Distribution Cost per case
Total Distribution Cost
Optimized Objective
Distance (Route)

Chapter Summary

This chapter presented an overview of how to use arrays in optimization problems. We used three examples to demonstrate the differences between *intrinsic* and *extrinsic* indexes in optimization arrays.

The Intrinsic Index attribute in Decision and Constraint arrays controls how the optimizer interprets arrays. The optimizer incorporates intrinsic arrays into the optimization while leaving extrinsic arrays eligible for array abstration outside the optimization.

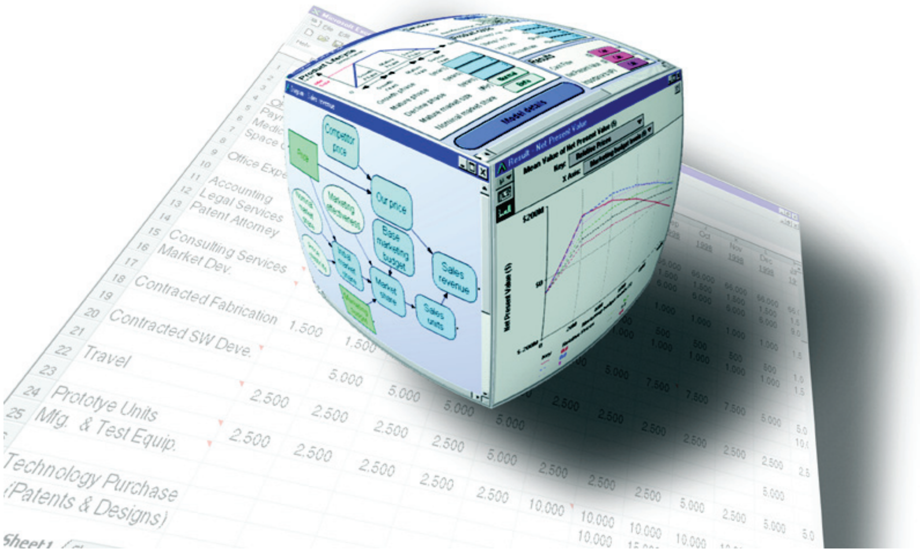
> In the first example, the input Decision and Constraint arrays had intrinsic dimensions. A single optimization run incorporated all elements of the intrinsic indexes simultaneously.
>
> In the second example we added an *extrinsic* dimension to the model. Analytica propagated this new dimension through the model according to the usual principles of array abstraction. As a result, the optimizer performed independent optimization runs for each element of the new index. The Optimized Solution array displayed the combined results as a single array having both intrinsic and extrinsic dimensions.
>
> In the third example, we restricted the solution space by substituting a new index for the input Decision and Objective arrays. We aggregated decision quantities using index maps in order to apply constraints.

Chapter 4

Key Concepts: The Airline NLP example

This chapter shows you how to:

- Apply parametric variations to Variable and Decision nodes in optimizations
- Use a parametric array of Initial Guesses
- Combine uncertainty with optimization using
  - Fractile or Average Stochastic method (FAST)
  - Multiple Optimizations of Separate Samples (MOSS) method
- Optimize using reduced objectives
- Use Time as an *intrinsic* or *extrinsic* index
- Understand special characteristics of NLPs
- Improve efficiency of NLPs using the SetContext parameter of DefineOptimization()
- Plane an Optimization inside a Dynamic Loop



Concepts covered in the Airline NLP example

The Airline NLP demonstrates a set of key concepts that Analytica Optimizer modelers should be familiar with. Although it includes some topics that apply only to NLP models, each module includes content that is relevant to all optimization types.

Before reading this chapter, you should already be familiar with the basic parameters of **DefineOptimization()** and **OptSolution()** functions (Chapter 1 on page 5-7), and roles of intrinsic and extrinsic indexes in optimization (Chapter 3 of this guide).

Additionally, Modules 3 and 4 of the Airline NLP example assume familiarity with Monte Carlo simulation and Probability Distributions (User Guide Chapter 16). Module 7 assumes familiarity with the **Dynamic()** function (User Guide Chapter 18)

Topics relevant to all optimization types (LP, QP, and NLP) are:

- **Module 1**: Setting up basic Airline NLP example
- **Module 2**: Parametric Analysis
- Combining uncertainty with optimization:
    - **Module 3**: Optimizing on Fractiles or Averages Stochastically (FAST)
    - **Module 4:** Multiple Optimizations of Separate Samples (MOSS) method
- **Module 5:** Abstracted objectives; example of Time as an *extrinsic* index
- **Module 6:** Intrinsic decision arrays; example of Time as an *intrinsic* index

Embedded topics relevant only to Non-Linear Problems (NLPs) are:

- Improving efficiency using *context variables* (**Modules 4 and 5**)
- **Module 7:** Embedding an NLP inside a dynamic loop

NLP Characteristics

The Airline model is an example of a Non-Linear Program (NLP), the most general formulation for an optimization. In this type of 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, and 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 whether a solution is globally optimal or not. For these reasons, it is better to reformulate nonlinear problems as linear or quadratic when possible.

Searching rather than solving

Algorithms for solving Linear Programs (LP) and Quadratic programs (QP) operate directly on arrays of coefficients for linear and quadratic terms. Once Analytica deduces these coefficients from your model, the problem is solved by the underlying engine without further interaction with your Analytica model. But NLP optimizations work differently. While an NLP is being solved, your model is being repeatedly evaluated at each search point. The search begins with an *Initial Guess* for the decision variables. Once the initial guess is established, the optimizer will calculate the corresponding objective quantity according to the structure of the model. For smooth models, the optimizer uses gradients and Jacobians (determined through finite differencing) to decide which decision values to use for the next guess. The iterative process continues until the gradient is within minimums (signaling a local or global optimum) or until time or iteration limits are reached.

Initial Guesses in NLPs

Since NLP methods cannot guarantee a global optimum, the solution may depend on the initial guess for decisions. Normally, Analytica evaluates the defined quantities in decision nodes and uses these values for initial guesses. You can optionally override the defined value using the Initial Guess attribute (see “The Initial Guess attribute” on page 11 and “Initial Guess” on page 68 for details about overriding initial guesses).

Improving efficiency by identifying *Context Variables*

As the Optimizer repeatedly assigns different values to the decision variables, it requires Analytica to repeatedly evaluate the objective function and all intermediate variables. In special cases where the optimization contains extrinsic indexes, the intermediate variables may be arrays that contain more slices of data than the optimization needs. This is a situation where Analytica’s efficiency with arrays can actually slow down NLP solution searches. The **DefineOptimization()** function has an optional parameter to remedy this: **SetContext**. Understanding which variables to designate as context variables is key to making sure that NLPs are running as fast as possible in Analytica. We develop this important concept in Module 4 of the example.

Airline NLP Module 1: Base Case

You can navigate to the Airline NLP example starting in the directory in which Analytica is installed.

```
Example Models/Optimizer Examples/Airline NLP.ana
```

The goal is to determine optimum ticket fare and the number of planes to operate such that the airline can achieve the highest profit possible.

Decisions are **Fare** and the **Number of Planes**. Since this is an NLP, the values we use to define these decisions will be used as initial guesses in the optimization. **Fare** will be a continuous variable with lower and upper bounds. **Number of Planes** will be an Integer value with lower and upper bounds.

```
Decision Fare := 200
Domain of Fare := Continuous(100,400)
Decision Number_of_Planes := 3
Domain of Number_of_Planes := Integer(1,5)
```

First we assume a base level of demand in passenger-trips per year assuming a base fare of $200 per trip.

```
Variable Base_Demand := 400k
```

Actual demand will vary by price according to market demand elasticity.

```
Variable Elasticity := 3
Variable Demand := Base_Demand*(Fare/200)^(-Elacticity)
```

Each plane holds 200 passengers. We assume each plane makes two trips per day, 360 days per year.

```
Variable Seats_per_plane := 200
Variable Annual_Capacity := Number_of_Planes*Seats_per_Plane*360*2
```

Annual seats sold is limited either by capacity or by passenger demand.

```
Variable Seats_Sold := Min([Demand, Annual_Capacity])
```

We assume annual fixed cost per plane and variable cost per passenger.

```
Variable Fixed_cost := 15M
Variable Var_cost := 30
```

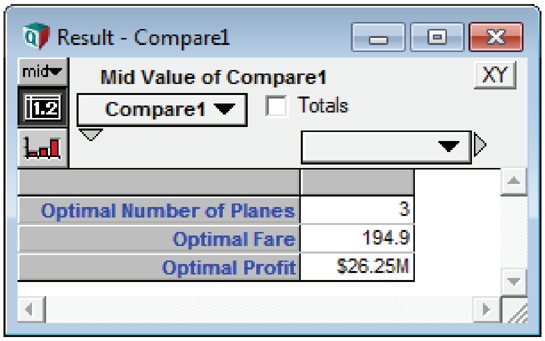
The objective, **Profit**, is the difference between revenue and cost

```
Objective Profit := Seats_sold * (Fare - Var_cost)
   - Number_of_Planes * Fixed_cost
```

Finally we create the optimization node and solution quantities.

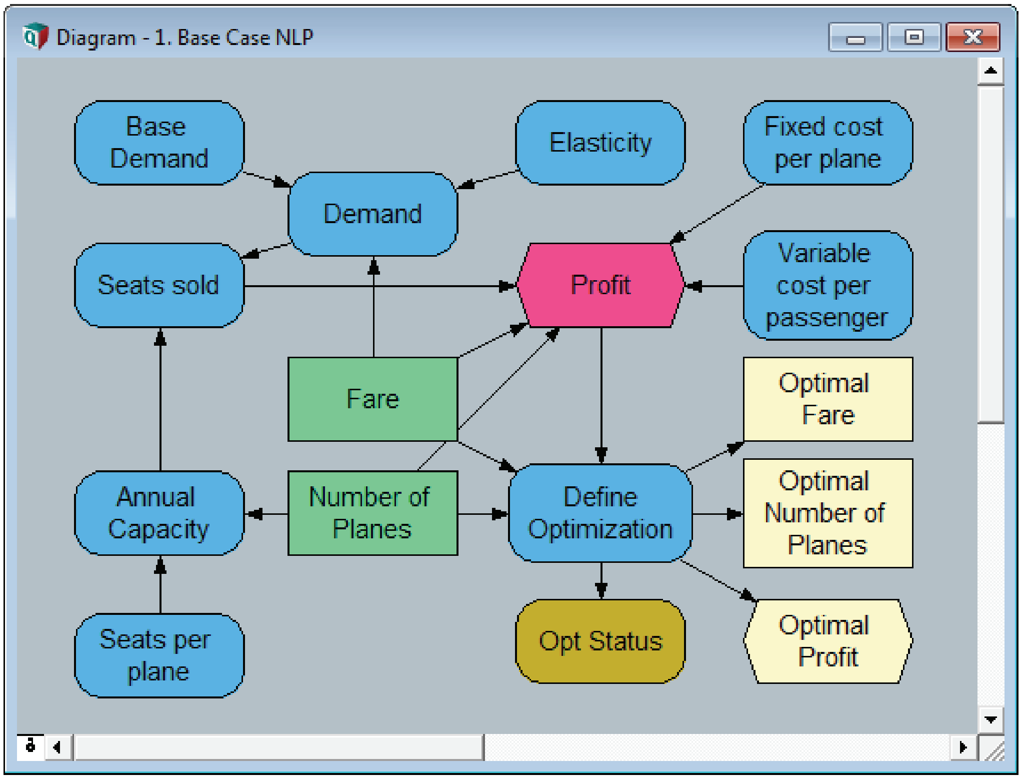
```
Variable Opt := DefineOptimization(
   Decisions: Number_of_Planes, Fare
   Maximize: Profit)
Decision Optimal_Fare := OptSolution(Opt, Fare)
Decision Optimal_Planes := OptSolution(Opt, Number_of_Planes)
Objective Optimal_Profit := OptObjective(Opt)
Variable Opt_Status := OptStatusText(Opt)
```

.



Optimal Number of Planes	3
Optimal Fare	194.9
Optimal Profit	$26.25M

The solution yields an optimal fare of $195 with three planes in service



Using parametric analysis: Airline NLP Module 2

It is often useful to vary the input values of a model to see how the changes affect downstream variables. In optimization problems, we are primarily concerned with the objective value. Analytica's Structured Optimization also makes it easy for you to see how parametric variations affect objectives and optimization results.

Analytica Optimizer treats parametric variations on Variables and Decisions differently.

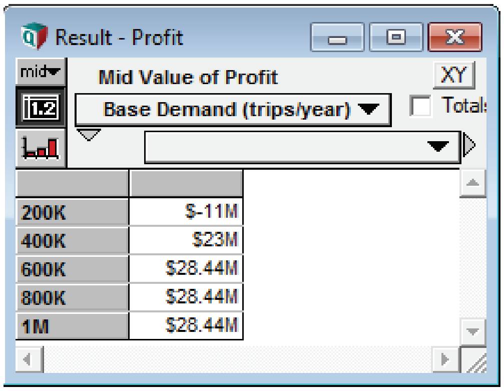
Parametric variations in input Variables

Applying Parametric variations to input Variables causes Analytica to abstract over the parametric index. This allows you to see how parametric variations affect the Objective value as well as the optimization result.

To see this, start with the Module 1 Base Case and apply parametric variations to **Base Demand**.

```
Variable Base_Demand := [200k,400k,600k,800k,1M]
```

Evaluate **Profit** to see how this variation affects the pre-optimized objective value. (This value applies the original Decision inputs: **Fare** = 200, **Number of Planes** = 3)



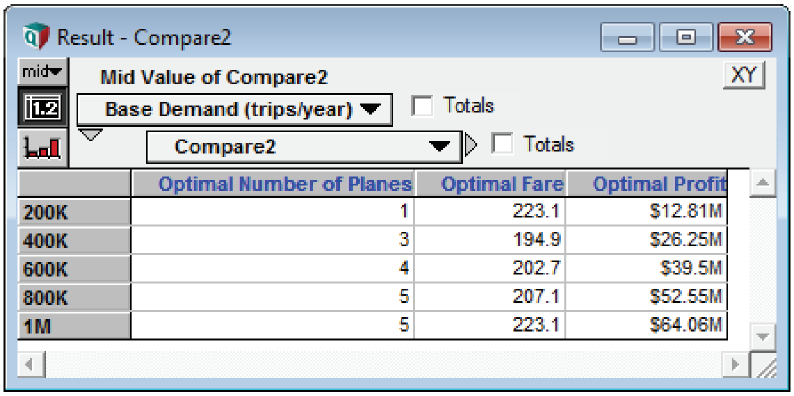
Mid Value of Profit

Base Demand (trips/year)

200K	$-11M
400K	$23M
600K	$28.44M
800K	$28.44M
1M	$28.44M

We see that profit levels saturate when planes reach full capacity. The transition happens somewhere between 400k and 600k values for Base Capacity, assuming original Decision values.

Now evaluate the solution variables:



Mid Value of Compare2

Base Demand (trips/year)

Compare2

	Optimal Number of Planes	Optimal Fare	Optimal Profit
200K	1	223.1	$12.81M
400K	3	194.9	$26.25M
600K	4	202.7	$39.5M
800K	5	207.1	$52.55M
1M	5	223.1	$64.06M

Analytica abstracts the parametric index on **Base Demand** so that we have separate optimization runs for each element. Indexes abstracted over optimizations are called *extrinsic*. Here we see that the airline can make a profit in all Base Demand scenarios after making the right adjustments to **Fare** and **Number of Planes**. Higher demand always leads to higher profit. For the lowest Base Demand scenario, a successful strategy is to raise the fare and save cost by operating only one plane.

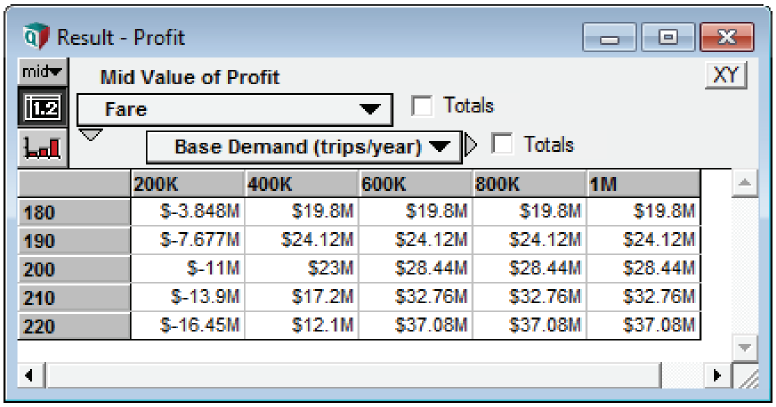
Parametric variations in Decisions

Applying parametric variations to Decisions will NOT cause array abstraction on the optimization. But you can still see how parametric variations on Decisions affect the Objective.

Apply parametric variation to Fare:

```
Decision Fare := [180,190,200,210,220]
```

Evaluate **Profit** to see how this variation affects the pre-optimized objective value. (This value applies the parametric inputs for **Fare**, and the original Decision value for **Number of Planes**: 3)



Mid Value of Profit

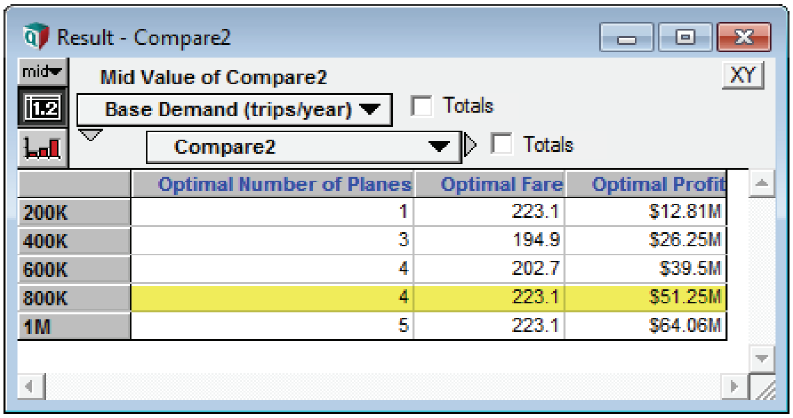
Fare ▼ □ Totals

Base Demand (trips/year) ▼ □ Totals

	200K	400K	600K	800K	1M
180	$-3.848M	$19.8M	$19.8M	$19.8M	$19.8M
190	$-7.677M	$24.12M	$24.12M	$24.12M	$24.12M
200	$-11M	$23M	$28.44M	$28.44M	$28.44M
210	$-13.9M	$17.2M	$32.76M	$32.76M	$32.76M
220	$-16.45M	$12.1M	$37.08M	$37.08M	$37.08M

The new Objective array shows the result of both variations. When Base Demand is low, a lower fare is best. When Base Demand is 600k or above, the higher fares are favored.

Because **Fare** is a Decision, its parametric index is ignored by the optimization. We expect the optimization results to have the same dimensions as before:



Mid Value of Compare2

Base Demand (trips/year) ▼ □ Totals

Compare2 ▼ □ Totals

	Optimal Number of Planes	Optimal Fare	Optimal Profit
200K	1	223.1	$12.81M
400K	3	194.9	$26.25M
600K	4	202.7	$39.5M
800K	4	223.1	$51.25M
1M	5	223.1	$64.06M

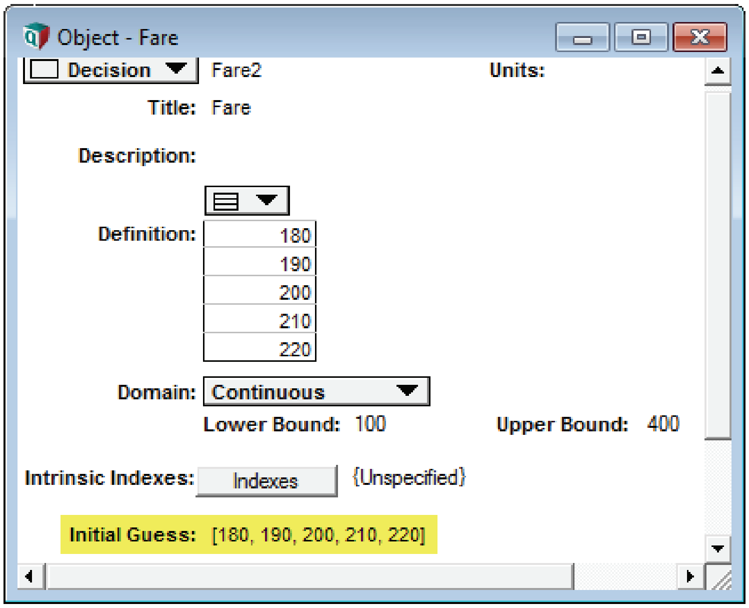
The dimensions are the same. The result is similar but not identical. Notice that the solution for the 800k Base Demand scenario has changed since we applied parametric variation to Fare. Why did this occur?

This demonstrates an unavoidable characteristic of NLPs: Due to the existence of locally optimum solutions, the solution may depend on the initial guess. Analytica uses the first element of a parametric series as the initial guess. By adding a parametric index on Fare, we effectively changed the initial guess from $200 to $180.

If you enter the parametric variation on Fare as an Initial Guess attribute of the Decision node, Analytica will abstract over the index and run separate optimizations for each initial guess value.

If the Initial Guess attribute is not already visible, select Attributes from the Object menu and toggle a check box next to Initial Guess. The attribute will now be visible in the Object window.

```
Initial Guess of Fare := [180,190,200,210,220]
```



Object - Fare

Decision Fare2 Units:

Title: Fare

Description:

Definition:

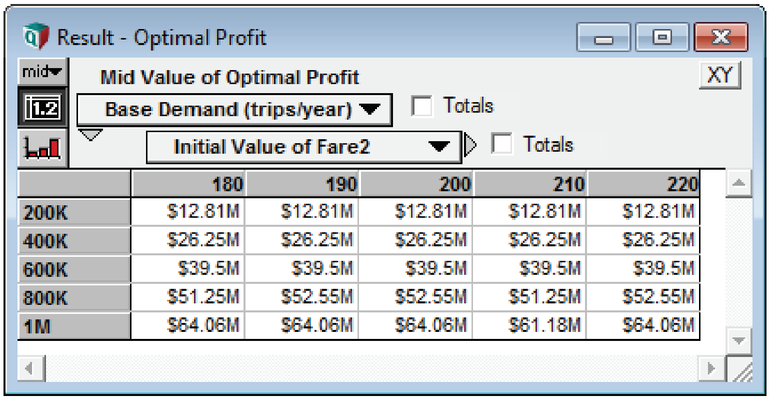
180
190
200
210
220

Domain: Continuous

Lower Bound: 100 Upper Bound: 400

Intrinsic Indexes: Indexes {Unspecified}

Initial Guess: [180, 190, 200, 210, 220]



Result - Optimal Profit

Mid Value of Optimal Profit

Base Demand (trips/year) □ Totals

Initial Value of Fare2 □ Totals

	180	190	200	210	220
200K	$12.81M	$12.81M	$12.81M	$12.81M	$12.81M
400K	$26.25M	$26.25M	$26.25M	$26.25M	$26.25M
600K	$39.5M	$39.5M	$39.5M	$39.5M	$39.5M
800K	$51.25M	$52.55M	$52.55M	$51.25M	$52.55M
1M	$64.06M	$64.06M	$64.06M	$61.18M	$64.06M

There are only slight differences in the optimal Profit achieved. The original of guess of **Fare** = 200 is never inferior in this case.

The tendency toward locally optimal solutions is a characteristic of the optimization engine alone. Analytica is designed to make all aspects of modeling as transparent as possible. This is especially important for optimization. In a spreadsheet environment using the same engine, the same optimization characteristics would apply but they would be more difficult to explore. Premium NLP engines available on Analytica include sophisticated algorithms to increase the likelihood of finding globally optimum solutions.

Optimizing with Uncertainty

Analytica uses Monte Carlo simulation to analyze uncertain quantities. Probability distributions are represented as sample arrays that include the system index `Run`. Each element of the `Run` index represents a sample of possible values for uncertain quantities, according to the chosen probability distribution. As an array dimension, `Run` propagates through the model in the same way as any other index. How can uncertain quantities be combined with optimization and how should we interpret results of uncertain optimizations?

There are two basic approaches to combining uncertainty with optimization in Analytica. The appropriate approach depends on whether the decision maker must commit before or after the uncertainty is resolved. These lead to a Monte Carlo simulation within a single optimization (the Stochastic approach) or an optimization within each Monte Carlo scenario (the *preposterior* approach). For convenience we label these FAST and MOSS, respectively.

1. Optimize on Fractiles or Averages Stochastically (FAST)

Reduce the `Run` index in the Objective and Constraints by using summary statistics. In the Airline example, Module 3, we maximize the expected value of the objective: `Mean(Profit)`. This collapses the `Run` dimension so that there is a scalar objective, and hence only one optimization. Fractile statistics are also commonly used instead of expected value. You can summarize the Objective as the median (50% fractile) or any other fractile value using the **GetFract()** function. For example, a constraint might require the 5% fractile for Profit to be greater than a certain value.
In any case, it is important to reduce the `Run` index in the Objective as well as all Constraints when performing Stochastic Optimization.

2. Multiple Optimizations of Separate Samples (MOSS)

Treat `Run` as an extrinsic index, abstracting it over the optimization. The result is a Monte Carlo sample of solutions. Each solution represents optimum decisions corresponding to the input values of an individual Monte Carlo sample. The MOSS method establishes the range of decision values that should be considered given the uncertainty. But it assumes that the actual decision will be made *a posteriori*, or after the uncertainty is resolved.

Even though Analytica's Intelligent Arrays make setting up MOSS optimizations much easier than in other environments, the method remains computationally intensive on all platforms. The sheer number of optimizations (equal to the sample size) may prevent the computational stone from rolling along as fast as you would like it to. It is always important to choose the correct sample size when using the MOSS method. For NLPs, you should also avoid unnecessary computation by identifying *context variables.* (See next section).

Module 3: Stochastic Optimization (FAST)

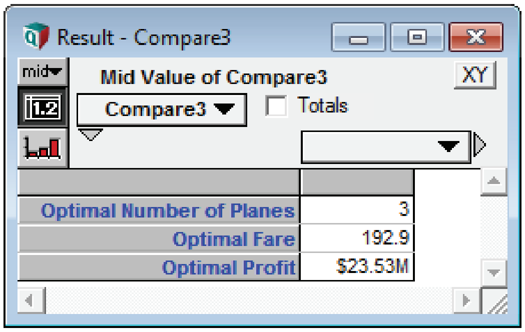
Starting with the Module 1 Base Case, suppose that **Base Demand** and **Elasticity** are uncertain quantities described by triangular distributions:

```
Chance Base_Demand := Triangular(300k, 400k, 500k)
Chance Elasticity := Triangular(300k, 400k, 500k)
```

Create a new Objective for the expected value of **Profit**. Use this as the objective for the optimization.

```
Objective Mean_Profit := Mean(Profit)
Variable Opt := DefineOptimization(
   Decisions: Number_of_Planes, Fare,
   Maximize: Mean_Profit)
```

Analytica allows you to adjust the sample size (the size of the `Run` index). For new models the default is 100. For the Airline NLP example, we only use 10. Select **Uncertainty Options** from the **Result** menu and enter 10 for the sample size.



Mid Value of Compare3

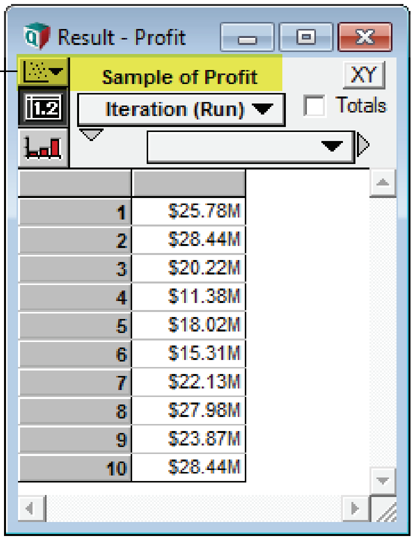
Optimal Number of Planes	3
Optimal Fare	192.9
Optimal Profit	$23.53M

This result represents the decision values that yield the highest expected value for **Profit**. It would be appropriate to implement these values if the uncertainty cannot be resolved before the decision is made, and if maximizing expected value of Profit is your goal. The values differ only slightly from the Base Case.

Module 4: Multiple Optimizations of Separate Samples (MOSS)

If you evaluate the `Profit` Objective from Module 3 (not the mean of `Profit`) and view in Sample mode, you'll see that it's an array dimensioned by `Run`.

Select Sample View



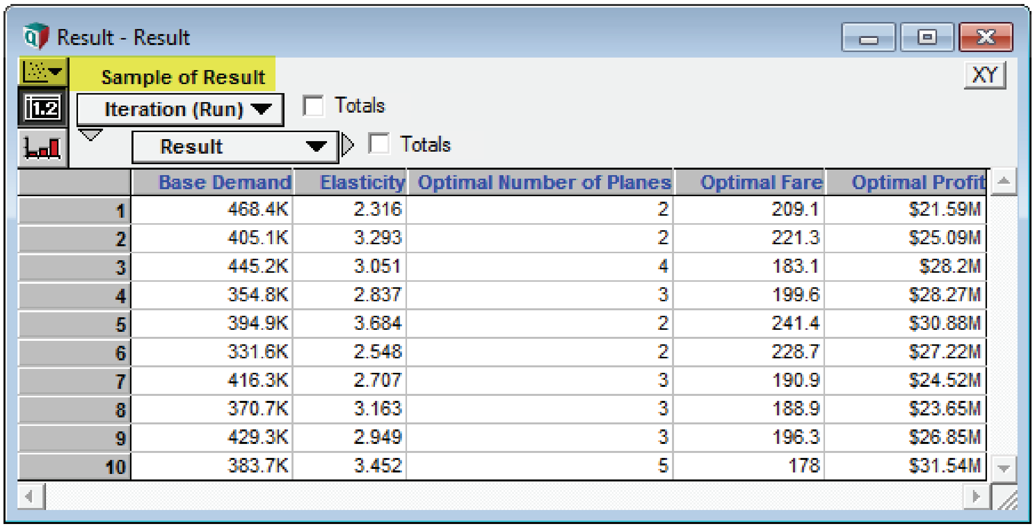
1	$25.78M
2	$28.44M
3	$20.22M
4	$11.38M
5	$18.02M
6	$15.31M
7	$22.13M
8	$27.98M
9	$23.87M
10	$28.44M

Naturally, the `Run` index was propagated to `Profit` from the `Base Demand` and `Elasticity` inputs. As long as an index does not originate from a Decision, it is eligible for array abstraction. To get a Monte Carlo sample of optimizations, we just need to make sure the dimension gets propagated all the way to the **DefineOptimization()** node. In Module 3, the **Mean()** function reduced the `Run` index before it could get that far.

Delete the Mean_Profit node and switch back to Profit as the optimization objective.

```
Variable Opt := DefineOptimization(
    Decisions: Number_of_Planes, Fare,
    Maximize: Profit)
```

If at first, the result does not look like it includes the `Run` index, remember that you must view it in Sample mode.



	Base Demand	Elasticity	Optimal Number of Planes	Optimal Fare	Optimal Profit
1	468.4K	2.316	2	209.1	$21.59M
2	405.1K	3.293	2	221.3	$25.09M
3	445.2K	3.051	4	183.1	$28.2M
4	354.8K	2.837	3	199.6	$28.27M
5	394.9K	3.684	2	241.4	$30.88M
6	331.6K	2.548	2	228.7	$27.22M
7	416.3K	2.707	3	190.9	$24.52M
8	370.7K	3.163	3	188.9	$23.65M
9	429.3K	2.949	3	196.3	$26.85M
10	383.7K	3.452	5	178	$31.54M

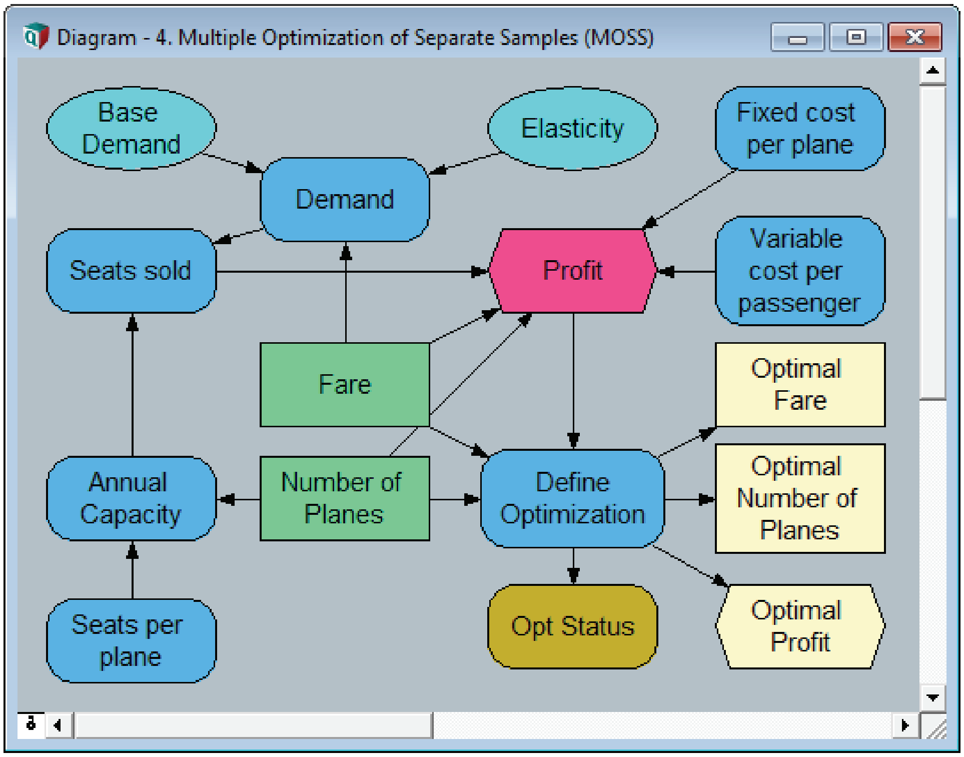
The MOSS approach does not determine what the best decision would be under circumstances of uncertainty. It gives us a range of decisions that would be optimal in situations where the corresponding **Base Demand** and **Elasticity** are already known with certainty. This allows you answer questions that would not be addressable using Stochastic Optimization. For example, "What is the *current expected value of Profit*, given that the **Number of Planes** and **Fare** decisions will be determined after **Base Demand** and **Elasticity** uncertainties are resolved?" The answer to this question is simply the average of the **Optimal Profit** samples (although it would be a good idea to use a larger sample size to avoid inherent sampling error.)

Improving Computational Efficiency of NLPs:

Module 4 of the Airline NLP was the first example in which we used an extrinsic index (**Run**) with an NLP. Since this was a very simple example with only ten samples, the result appeared pretty much instantly. Improving computational efficiency in this case would be important only to the world's most ambitious competitive coffee drinking champion. But real world NLPs can be very demanding on processing cycles. The dirty secret of the Module 4 example described above is that it made about ten times as many calculations as it needed to. Since Module 4 is intended to be a proxy for larger models, we need to fix this problem and make it run even faster! (This section is sponsored by Starbucks.)

Remember that NLP search algorithms are iterative. The optimizer repeatedly sets new Decision values and re-evaluates every downstream variable, all the way to the Objective value.

.

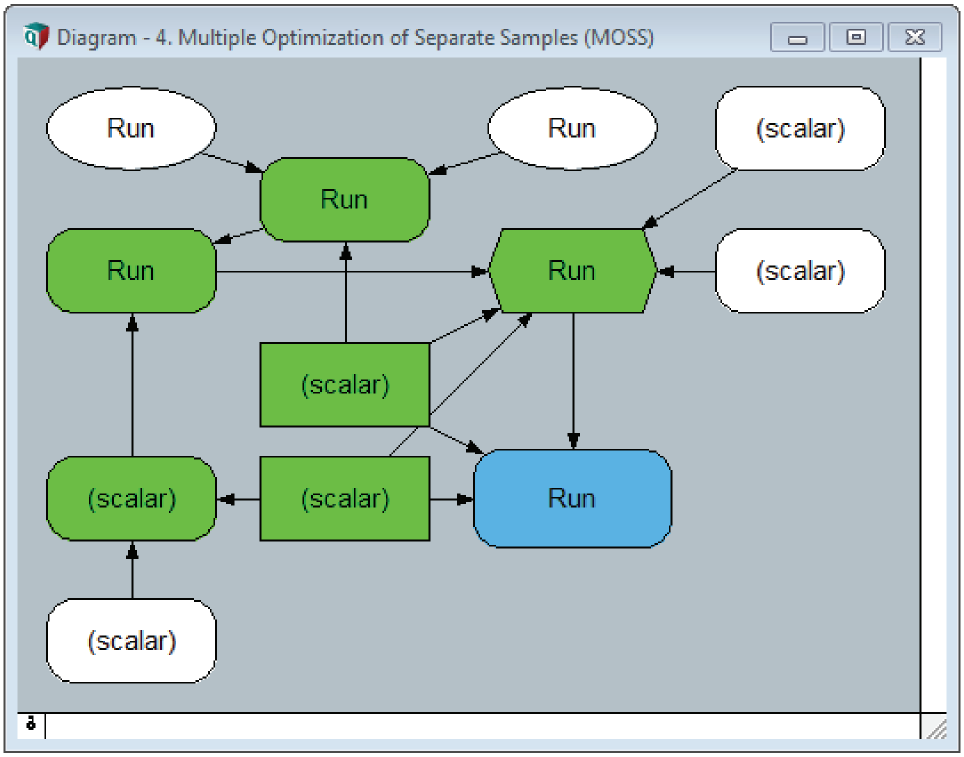


The influence diagram makes it easy to identify variables that are evaluated repeatedly during the search. These include **Annual_Capacity**, **Seats_Sold**, **Demand,** and of course the Objective: **Profit**

If any of these variables contains an extrinsic index (such as **Run** in this example) Analytica will compute the entire array for each iteration. This includes slices for all elements of **Run** whether they are needed or not. But **Run** is an extrinsic index in this version of the Airline model. This means that each element of Run has its own optimization and vice versa. Within a given optimization, the optimizer is interested in only one **Run** element even though entire arrays are being evaluated repeatedly. The superfluous slices are discarded by the optimizer, and the next iteration starts.

The optional **SetContext** parameter of **DefineOptimization()** allows you to identify nodes for which only a single element of the extrinsic index applies to a given optimization. This avoids the inefficiency described above.

:



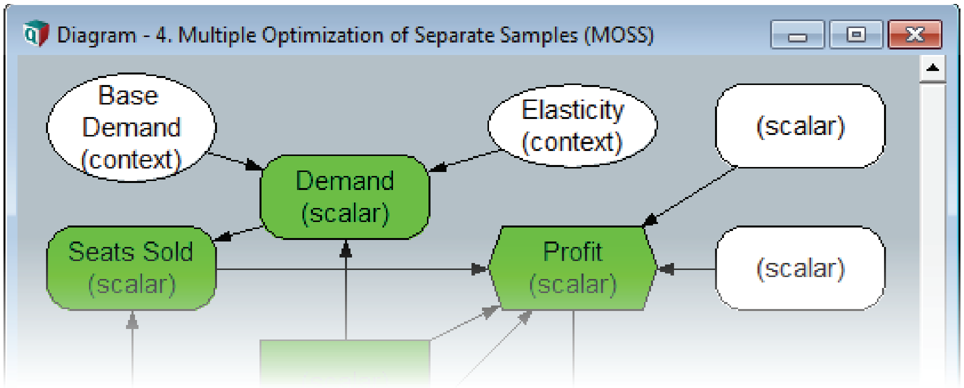
To identify the best context variables, let’s look at the same influence diagram in a different light. Nodes have been re-labeled here to show the extrinsic indexes present or “scalar” if none. Iterated quantities (i.e. quantities downstream of Decisions) are colored green.

There are a few basic principles of context setting:

- A context variable will not propagate extrinsic indexes to downstream variables during the optimization.
- You should avoid using variables that are downstream of Decisions. They are repeatedly evaluated and will therefore be only partially effective at improving efficiency.
- Context variables should be as close to the optimization as possible without being downstream of Decisions.
- The set of context variables should include only as many as necessary to prevent propagation of extrinsic indexes to iterated quantities.

In this example, Setting context on **Demand** would eliminate the **Run** index from **Seats Sold** and **Profit.** But **Demand** is downstream of a Decision and is therefore NOT the most suitable candidate.

.



Base Demand and **Elasticity** are right choices. They are only evaluated once, and together they can eliminate **Run** from the rest of the model during optimization.

The diagram above shows how the chosen context variables prevent the extrinsic index from being propagated to iterated variables. Iterated variables end up being scalar in terms of their extrinsic dimensions in the context of a single optimization. Outside the optimization, the dimensions of these arrays don't change.

The following definition will dramatically improve the performance of the Airline NLP Module 4 example:

```
Variable Opt := DefineOptimization(
   Decisions: Number_of_Planes, Fare,
   Maximize: Profit,
   SetContext: [Base_Demand, Elasticity])
```

Note: *In this example,* ***Run*** *was merely an example of an extrinsic index. The* ***SetContext*** *parameter is important for* ***all NLPs that use extrinsic indexes*** *whether they contain uncertain quantities or not.*

Module 5: Time as an *Extrinsic* index

Let's assume that Base Demand increases by 10% per year. How would that affect our decisions over time. In this example you Analytica will abstract over the system index **Time** instead of **Run**. The principle is exactly the same as in Module 4.

First we define the system Time index to represent years from 2011 to 2015. To edit this special index, select Edit Time from the Definition menu and enter the following definition:

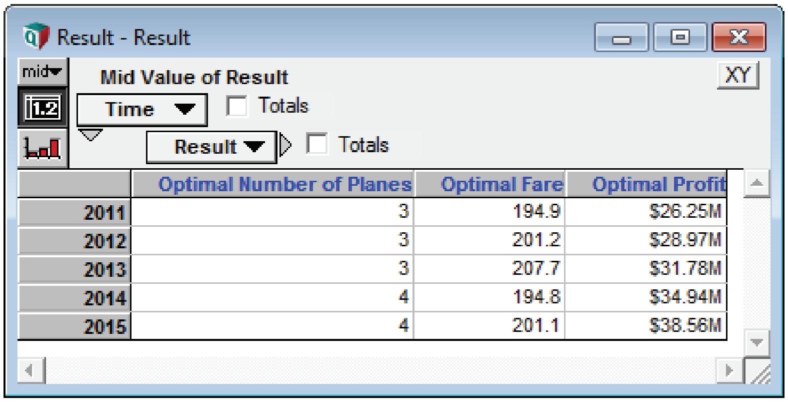
```
Index Time := 2011..1015
```

Starting with the Base Case setup, create a new node titled **Base Demand Growth Rate**. Re-define **Base Demand** to that is grows by this rate exponentially over time.

```
Variable Growth_Rate := 10%
Variable Base_Demand := 400k*(1+Growth_Rate)^(@Time-1)
```

This definition uses a positional reference for the **Time** index. The first year corresponds to position 1 (making the exponent zero). Thus, **Base Demand** for the first year will be 400k, and this value increases 10% every year.

Time is now an index of **Base Demand**. The extrinsic index is propagated to the Objective. Array abstraction results in separate optimizations for each year.



	Optimal Number of Planes	Optimal Fare	Optimal Profit
2011	3	194.9	$26.25M
2012	3	201.2	$28.97M
2013	3	207.7	$31.78M
2014	4	194.8	$34.94M
2015	4	201.1	$38.56M

Profit rises in each year as demand increases. A new plane is added in 2014.

Wait! Did you forget something? This in an NLP with an extrinsic index. If you remembered to set a context variable you win a free Grande Analyti-latte! **Base_Demand** is the sensible choice.

```
Variable Opt := DefineOptimization(
    Decisions: Number_of_Planes, Fare,
    Maximize: Profit,
    SetContext: Base_Demand)
```

Identifying the Source of an Extrinsic Index

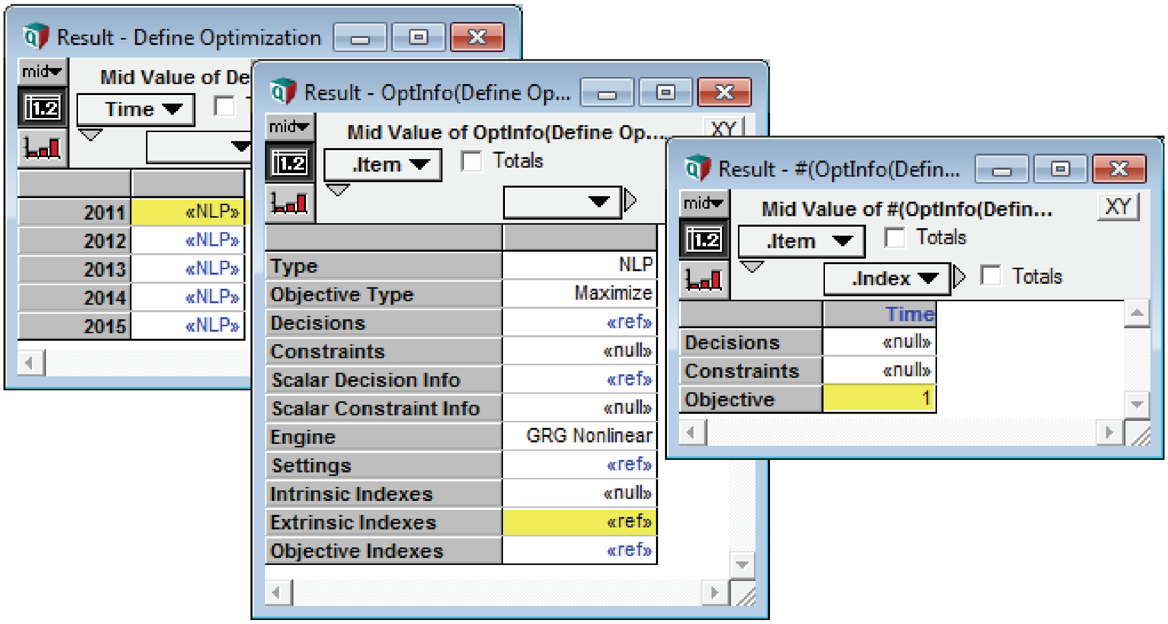
Due to Analytica's efficiency in array abstraction, you may occasionally find that you have inadvertently propagated an extrinsic index through your optimization model. If it is a very complex model, this can have an unpleasant effect on memory loads and computation times. It can also be difficult to trace the source of the error.

The **OptInfo()** function in Analytica Optimizer 4.3 includes "Extrinsic Indexes" as a convenient new item designed for this purpose. Even more conveniently, you don't even have to use the **OptInfo()** function to see this result. All OptInfo items are accessible by double-clicking the encoding object displayed when you evaluate **DefineOptimization()**.

Evaluate the Optimization node and set the view to Sample mode.

Double-click on the optimization encoding object: «NLP»

In the next view, double click the reference object next to **Extrinsic Indexes**.


Result - Define Optimization
mid
Mid Value of De
Time
Result - OptInfo(Define Op...
Mid Value of OptInfo(Define Op...
XY
.Item
Totals
Result - #(OptInfo(Defin...
Mid Value of #(OptInfo(Defin...
.Index

2011	«NLP»
2012	«NLP»
2013	«NLP»
2014	«NLP»
2015	«NLP»

Type	NLP
Objective Type	Maximize
Decisions	«ref»
Constraints	«null»
Scalar Decision Info	«ref»
Scalar Constraint Info	«null»
Engine	GRG Nonlinear
Settings	«ref»
Intrinsic Indexes	«null»
Extrinsic Indexes	«ref»
Objective Indexes	«ref»

	Time
Decisions	«null»
Constraints	«null»
Objective	1

In a complex model, you may have many Decisions and Constraints. The **OptInfo()** view would identify the specific arrays that contain various extrinsic indexes. In this example the **Time** index is traced to the Objective.

The final window here would also displayed by `OptInfo(Opt, "Extrinsic Indexes")`. See the Function Reference chapter on page 81 for a list of all **OptInfo()** items.

Module 6: Time as an *Intrinsic* Index

If there are interactions between decisions in different years, you might want to find the decisions in each year that collectively maximize the Net Present Value (NPV) or another objective that aggregates over time. In this example, the NPV() function will reduce the time dimension so that the Objective will be a scalar value again. But there is an array of decisions over time that all contribute to this value. In this case, there is one optimization that yields an collective array of decisions of decisions over time. `Time` becomes an *intrinsic* decision index.

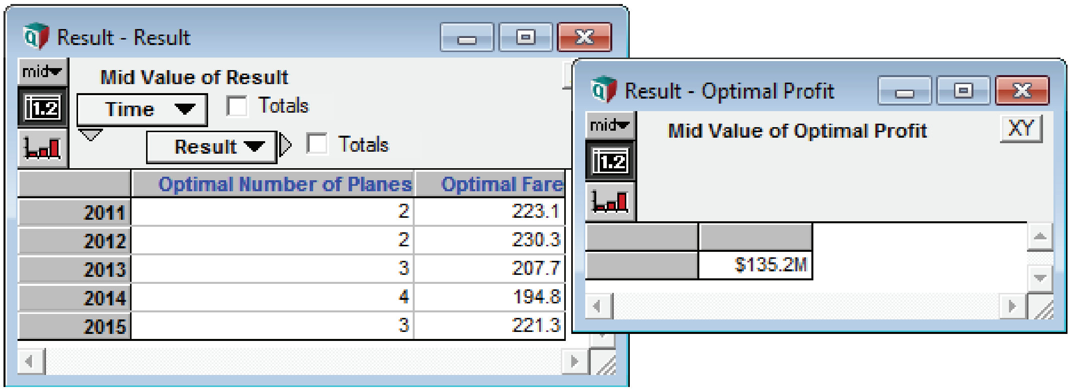
Create a new Objective node for NPV of Profit. Assume a discount rate of 5%.

Use the new objective in the optimization.
(Context variables are not necessary because there are no extrinsic indexes here.)

```
Variable Discount_Rate := 5%
Objective NPV_Profit := NPV(Discount_Rate, Profit, Time)
Variable Opt := DefineOptimization(
    Decisions: Number_of_Planes, Fare,
    Maximize: NPV_Profit)
```

Redefine the Decisions as arrays. We can use the **Array()** function to extend the original initial values over time. To avoid ambiguities about how the `Time` index should be handled, be sure to list it as an intrinsic index in the Decision node attributes:

```
Decision Fare := Array(Time, 200)
Intrinsic_Indexes of Fare := [Time]
Decision Number_of_Planes := Array(Time, 3))
Intrinsic Indexes of Number_of_Planes := [Time]
```



	Optimal Number of Planes	Optimal Fare
2011	2	223.1
2012	2	230.3
2013	3	207.7
2014	4	194.8
2015	3	221.3

$135.2M

Now a single optimization yields an array of decisions over time. They collectively maximize a single objective value: the NPV of profit over all years.

The computational time requirements for NLPs typically increase superlinearly with the number of decision variables, so this approach can become time-consuming if you have many decision variables and time periods. In general, it takes longer than Module 5 where we optimized separately for each year. The intrinsic time approach only makes sense if there is interaction between time periods that might favor a less than optimal objective value in a particular period for the benefit of the whole time series. In the absence of this type of interaction, it is better to run discrete optimizations for each period. This principle can be generalized to any decision index.

Module 7: Embedding an NLP in a Dynamic Loop

The solution in Module 6 above advises the airline to decrease the number of planes in service between 2014 and 2015. What if the airline has a lease agreement that does not allow them to return planes? In this case we would like to impose a constraint that requires the Number of Planes to increase or stay the same from year to year.

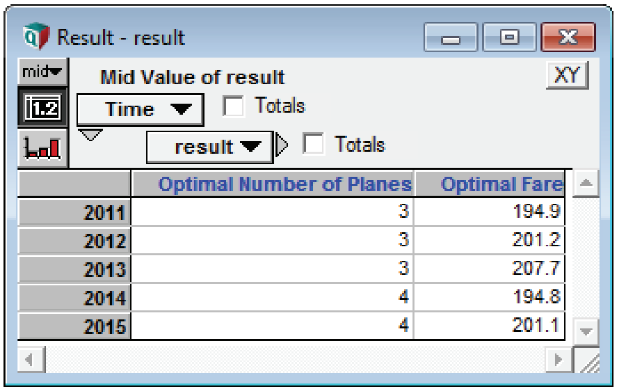
The lower bound of the constraint depends on `Number of Planes` in the previous year. The constraint is based on a dynamic quantity.

You can embed an NLP inside a dynamic loop to impose dynamic constraints. There is only one restriction:

- Bounds on Decision variables cannot be recursively dependent on Time (or other index on which the Dynamic loop is based). To impose recursively time-dependent bounds, they must be entered as constraints instead of Decision bounds attributes.

Let's try it! Starting with Module 6, create a new variable titled `Previous Number of Planes`. (We will have to use a dynamic definition since we are referring to a previous time step.) Then create the constraint based on the new variable.

```
Variable Previous_Planes := Dynamic(0, Number_of_Planes[Time-1])
Constraint No_Plane_Decrease := Number_of_Planes >= Previous_Planes
Variable Opt := DefineOptimization(
   Decisions: Number_of_Planes, Fare,
   Constraints: No_Plane_Decrease
   Maximize: NPV_Profit)
```



	Optimal Number of Planes	Optimal Fare
2011	3	194.9
2012	3	201.2
2013	3	207.7
2014	4	194.8
2015	4	201.1

The dynamic constraint is satisfied.

Controlling Engine Selection and Settings

Optimization engines can be controlled by changing certain internal settings. These settings are accessible through the optional **SettingName** and **SettingValue** parameters of **DefineOptimization()**. It is sometimes useful to experiment with different settings if the engine is not performing as expected on a particular type of problem. See “Specifying settings” on page 92 for more details.

You may also want to try different engines using the optional **Engine** parameter of **DefineOptimization()**. See “DefineOptimization()” on page 72.

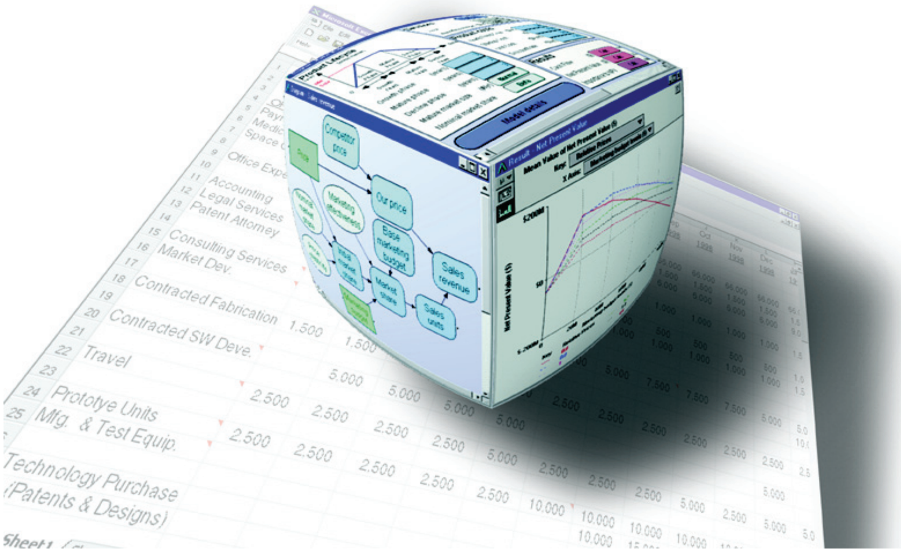
Chapter 5

Logistic Regression Function Reference

This chapter describes the Analytica logistic regression functions:

- **Logistic_regression( )**
- **Probit_regression()**
- **Poisson_regression()**

These functions are found in the `Generalized Regression.ana` library, and require Analytica Optimizer to use.



Logistic regression functions

The `Generalized Regression.ana` library contains functions that you can use to estimate the probability (or probability distribution) of a dependent (output) variable as a function of known values for independent (input) variables. This is similar to linear regression, which predicts the value of a dependent variable as a function of known values for independent variables. Logistic regression is the best known example generalized regression, so even though the term logistic regression technically refers to one specific form of generalized regression (with prob and poisson regression being other instances), it is also not uncommon to hear the term logistic regression functions used synonymously with generalized regression, as we have done with the title of this chapter.

To use the functions described in this chapter, you must have Analytica Optimizer and you must add the `Generalized Regression.ana` library to your model using the **Add Library** option from the **File** menu.

Logistic_regression(y, b, i, k)

Logistic regression is a technique for predicting a Bernoulli (i.e., 0,1-valued) random variable from a set of continuous dependent variables. See the Wikipedia article on logistic regression (http://en.wikipedia.org/wiki/Logistic_regression) for a simple description. Another generalized logistic model that can be used for this purpose is the **Probit_regression()** model. These differ in functional form, with the logistic regression using a **logit** function to link the linear predictor to the predicted probability, while the probit model uses a cumulative normal for the same.

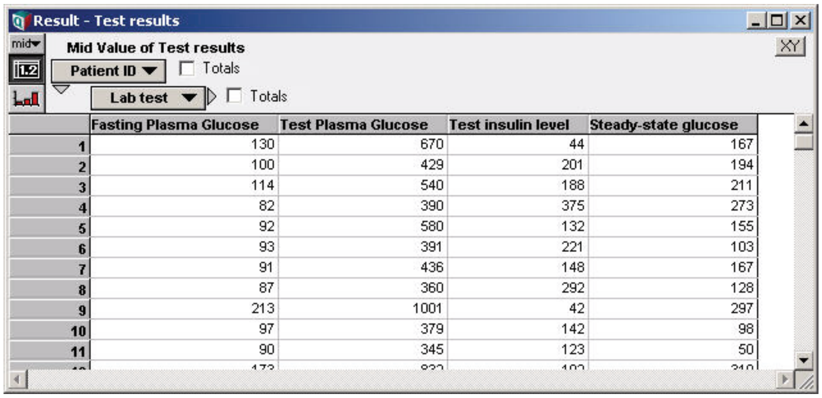
The **Logistic_regression()** function returns the best-fit coefficients, **c**, for a model of this form given a data set basis **b**, with each sample classified as **y_i**, having a classification of 0 or 1.

$$1n\left(\frac{p_i}{1-p_i}\right) = \sum_k c_k b_{i,k}$$

The syntax is the same as for the **Regression()** function. The basis can be of a generalized linear form, that is, each term in the basis can be an arbitrary nonlinear function of your data; however, the **logit** of the prediction is a linear combination of these.

When you have used the **Logistic_regression()** function to compute the coefficients for your model, the predictive model that results returns the probability that a given data point is classified as 1.

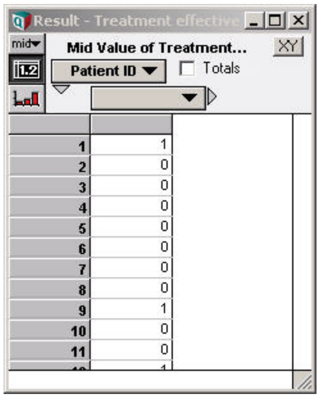
Example Suppose you want to predict the probability that a particular treatment for diabetes is effective given several lab test results. Data is collected for patients who have undergone the treatment, as follows, where the variable `Test_results` contains lab test data and `Treatment_effective` is set to `0` or `1` depending on whether the treatment was effective or not for that patient.



□ Totals

□ Totals

	Fasting Plasma Glucose	Test Plasma Glucose	Test insulin level	Steady-state glucose
1	130	670	44	167
2	100	429	201	194
3	114	540	188	211
4	82	390	375	273
5	92	580	132	155
6	93	391	221	103
7	91	436	148	167
8	87	360	292	128
9	213	1001	42	297
10	97	379	142	98
11	90	345	123	50



□ Totals

1	1
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	1
10	0
11	0

Using the data directly as the regression basis, the logistic regression coefficients are computed using this.

```
Variable c := Logistic_regression( Treatment_effective,
    Test_results, Patient_ID, Lab_test )
```

We can obtain the predicted probability for each patient in this testing set this.

```
Variable Prob_Effective :=
    InvLogit( Sum( c*Test_results,Lab_Test ))
```

If we have lab tests for a new patient, say `New_Patient_Tests`, in the form of a vector indexed by `Lab_Test`, we can predict the probability that treatment will be effective this.

```
InvLogit( Sum( c*New_patient_tests, Lab_test ) )
```

Probit_regression(y, b, i, k)

A probit model relates a continuous vector of dependent measurements to the probability of a binomial (i.e., 0,1-valued) outcome. In econometrics, this model is sometimes called the ***Harvard model***. The **Probit_regression()** function infers the coefficients of the model from a data set, where each point in the training set is classified as 0 or 1.

Probit regression is very similar to **Logistic_regression()**. Both are used to fit a binomial outcome based on a vector of continuous dependent quantities. They differ in their use of the **link** function.

Given a set of data points, indexed by **i**, with each point classified as **0,1** in the **Y** parameter, and a set of basis terms, **b**, containing the dependent variables (where the vector of dependent variables is indexed by **k**), the **Probit_regression()** function finds and returns the set of coefficients for the probit model where Φ is the inverse cumulative normal distribution function.

$$p_i = \Phi\left(\sum_k c_k b_k\right)$$

The basis, **b**, is a function of the dependent variables in your data. Each element along **k** of the basis vector can be an arbitrary, even nonlinear, combination of the data in your data set. However, the number of terms in the basis should be kept small relative to the number of data point in your data set.

Example Probit regression can be applied to the same prediction problem example shown above for logistic regression. The probit coefficients are obtained using this.

```
Variable c2 := Prob_regression( Treatment_effective, Test_results,
    Patient_ID, Lab_test )
```

The predicted probability for a new patient (with lab tests given by **`New_patient_tests`**) is given by this.

```
CumNormal( Sum( c2*New_patient_tests, Lab_test ) )
```

Library `Generalized Regression.ana`

Poisson_regression(y, b, i, k)

A Poisson regression model is used to predict the number of events that occur, **y**, from a vector independent data, **b**, indexed by **k**. The **Poisson_regression()** function computes the coefficients, c, from a set of data points, (**b**, **y**), both indexed by **i**, such that the expected number of events is predicted by this formula.

$$E(Y) = \exp\left(\sum_k c_k b_k\right)$$

The random component in the prediction is assumed to be Poisson-distributed, so that given a new data point **b**, the distribution for that point is shown below.

```
Poisson(sum(c*B,K)
```

If your dependent variable is continuous, with normally distributed error, use **Regression** or **RegressionDist**[2]. If your dependent variable is binomially distributed (i.e., 0,1-valued), use **Logistic_Regression()** or **Probit_Regression()**. If your dependent variable models a count, such as the number of events that occur, use **Poisson_Regression()**.

Note: *The distribution here accounts for data variation only, and does not include error in the coefficients c, as the **RegressionDist()** function does, for example.*

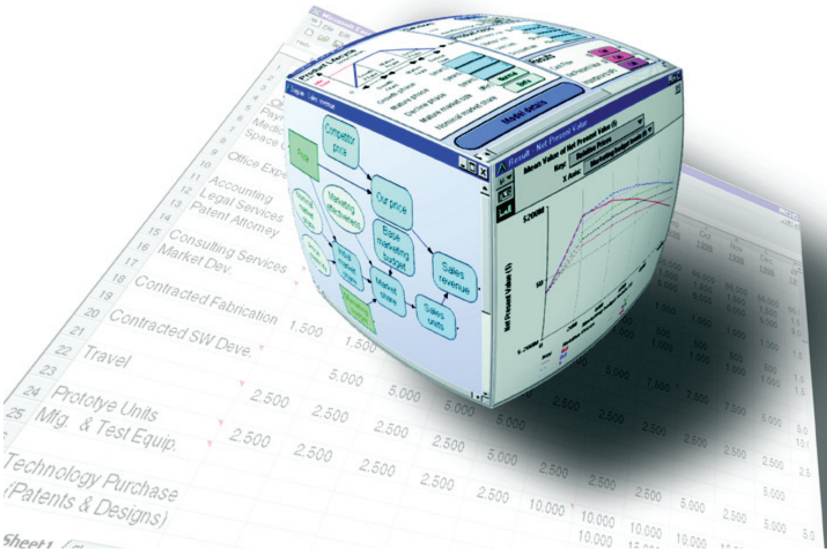
Library `Generalized Regression.ana`

[2] To use **RegressionDist**, add the `Multivariate Distributions.ana` library to your model.

Chapter 6 Optimizer Attribute Reference

This chapter describes special node attributes used in optimization problems:

- Domain and Bounds
- Intrinsic Indexes
- Initial Guess



Visible and Hidden Attributes

The default set of attributes Analytica displays depends on the edition. Standard attributes for Analytica Professional and Analytica Enterprise include *Identifier*, *Title*, *Units*, *Description*, and *Definition*. This chapter focuses on some additional attributes that are specific to Analytica Optimizer. These include:

Analytica Optimizer Attributes	Node Type	Visibility
Domain and Bounds	Decision	Always visible
Intrinsic Indexes	Decision, Constraint	Always visible
Initial Guess	Decision	Hidden by default, visible if chosen

Domain and Bounds

The domain attribute specifies the set of possible values for a variable. In general, a domain can be various combinations of continuous or discrete, and bounded or unbounded. The grouped integer category is a special case, requiring variables to be discrete, bounded, and unique within a variable group.

In most cases, you can specify the desired domain using the convenient popup menus in the Object window for a decision variable. Domains can also be entered in expression format which can add more flexibility (using array-based expressions for example).

Bounds

Entering bounds will constrain the variable within lower and upper limits. These do not count as constraints for optimization engines that impose limitations on the total number of constraints.

Domains

Automatic By default, the domain for a new decision variable is set to **Automatic**. This setting allows Analytica to determine appropriate domain based on the definition of the variable, leaving it unbounded. The domain will automatically switch from **Automatic** to **Continuous** when bounds are applied.

Continuous Allows any double-precision decimal value between -10^{40} to 10^{40}. Although Analytica's double float ranges from -8.988e10^{307} to8.988e10^{307} (-2^{1023} and 2^{1023}), most optimizer engines treat anything larger than 10^{40} as equivalent to infinity.

Integer Restricts values to integers.

Grouped Integer Some optimization problems require a solution where each variable is assigned a unique value among discrete choices. The Grouped Integer domain establishes a sequence of integers 1 through N, where N is the size of the group. All decision variables in the group are assigned an integer, and no two scalar values can share the same assignment.

If all elements of the decision array belong to the same group:

- Select **Grouped Integer** from the domain list.
- Enter a group name (This is optional if there is only one group in your model.)

Multiple decision nodes can share the same group name if all scalar values in the combined must have unique values. The size of the integer group is equal to the total number of scalar decision values assigned to it. If an array is assigned to a group, the group will contain all scalar elements in the array.

You can assign multiple groups within the same decision node by entering the Grouped Integer information in expression mode and entering an array expression for the group name parameter. (See Domain and Bounds Expressions below) This advanced technique is demonstrated in the Sudoku with Optimizer.ana example file.

Boolean Restricts values to either 0, 1.

Discrete This pop-up menu choice is not used in Optimization problems because discrete values must be listed explicitly. Instead of this choice, use Explicit Values or the Discrete() function in expression format.

Explicit Values This choice allows you to enter a list of possible values for the variable. These can be numbers, expressions, or text strings.

Copy From Index This choice populates the explicit values list using the elements of an existing index.

Domain and Bounds Expressions

Every Domain choice listed above can be entered in expression format. These expressions are fully generalized. They can even include array formulas and conditional statements. To enter a domain in expression format, select **Expression** from the Domain pop-up menu in the Object window.

Domain expression syntax	Examples

Continuous(lb,ub)	Leaving bounds unspecified:
	`Continuous()`
	Assign upper and lower bounds:
	`Continuous(lb:0, ub:2.734)`
	`Continuous(0, 2.734)`

Continuous(lb,ub,orZero:true)
Semi-continuous: Between bounds or zero.
	`Lower bound only:`
	Continuous(lb:1K, orZero:true)
	Both bounds:
	Continuous(lb:10,ub:15,orZero:true)

Integer(lb,ub)	Leaving bounds unspecified:
	`Integer()`
	Assign upper and lower bounds:
	`Integer(lb:0, ub:3)`
	`Integer(0, 3)`

Boolean() Restricts values to 0 or 1.	`Boolean()`

Discrete(*value1*, *value2*, ...)
Variable can only take on listed values.
	`Discrete(2,4,6,8)`
	`Discrete("Win","Place","Show")`

GroupedInteger(GroupName)
All variables within the same group are assigned different integer values.
	Single group assignment:
	`GroupedInteger("Sales_Ranking")`
	Assign multiple groups using an array expression:
	`GroupedInteger("C"&Column_index)`

Intrinsic Indexes

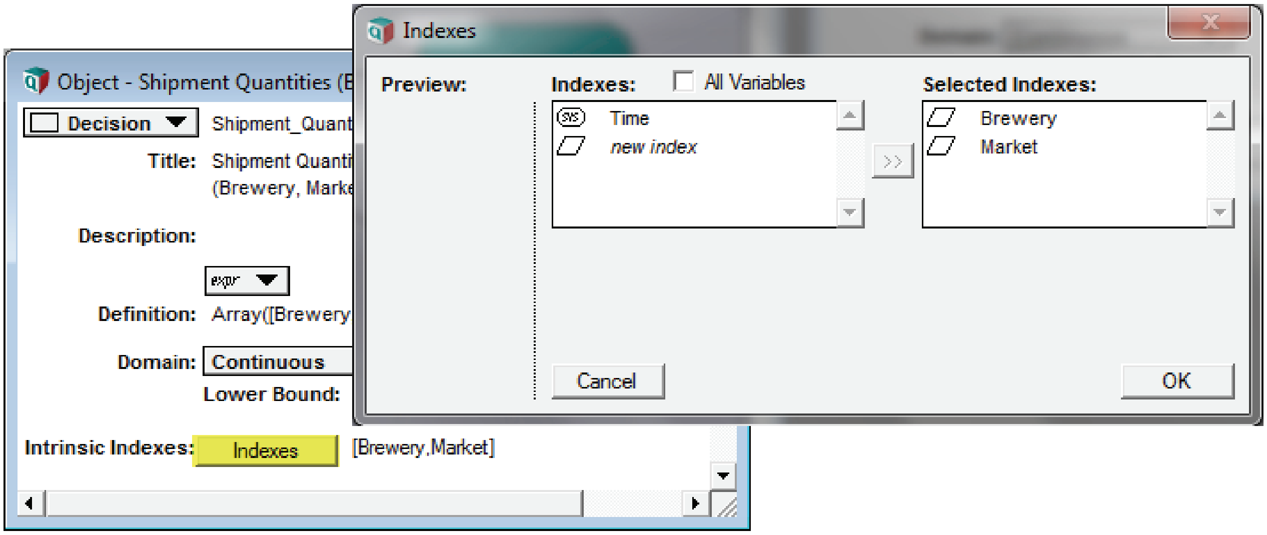
Intrinsic Index attributes are present in Decision and Constraint nodes. Analytica incorporates designated intrinsic indexes within optimizations and avoids abstracting over them. See Chapter 3 for a more complete explanation of the meaning of intrinsic and extrinsic indexes in the context of optimizations.

Specifying Intrinsic Indexes

To populate the Intrinsic index list:

- Select the Edit Tool.
- Open the Object window for Decision or Constraint node
- Press the Indexes button next to the Intrinsic Index attribute. An index selection window will appear.

- Select the desired indexes from list on the left. Press the transfer button to move them to the selected indexes list on the right.
- Click **OK**

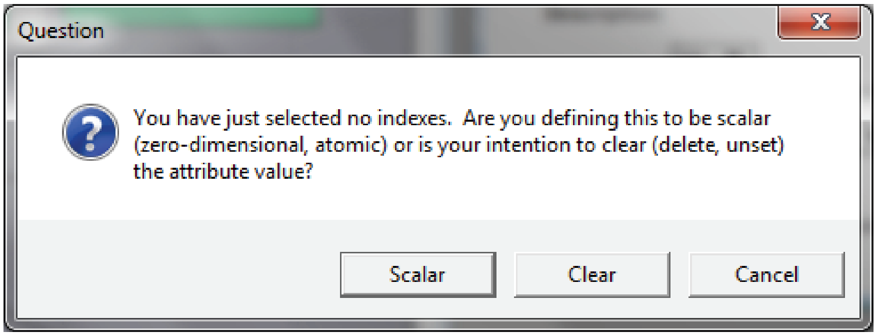


Specifying that there are *NO* Intrinsic Indexes

You can explicitly specify that a Decision should NOT have any intrinsic indexes. This means that the optimizer will treat the Decision as a single scalar value within any single optimization. This makes all dimensions of the array eligible for abstraction, whereby Analytica will perform separate optimizations for every element of the array.

To specify scalar status for a decision:

- Select the Edit Tool.
- Open the Object window for the Decision variable.
- Press the Indexes button next to the Intrinsic Index attribute. An index selection window will appear.
- If there are any indexes in the selected indexes list on the right, use the transfer button to move them back to the available indexes list on the left.
- With the selected indexes list blank, Click **OK.**
- A dialog box will appear asking if you want to clear index status (leaving status unspecified) or to specify the node as *scalar*. Choose Scalar.


Question

You have just selected no indexes. Are you defining this to be scalar (zero-dimensional, atomic) or is your intention to clear (delete, unset) the attribute value?

Leaving Intrinsic Index lists unspecified

If you leave the Intrinsic Index list unspecified, Analytica will analyze the optimization problem to infer the intrinsic indexes of your unspecified decision or constraint using heuristics. Although these inference algorithms are complex, there are a few guidelines:

- Indexes appearing in the various attributes of a decision — the definition, domain and bounds, or initial guess — provide the heuristics with a set of candidate indexes. If an index doesn't appear in any of these attributes, it won't be inferred to be an intrinsic or extrinsic index of the decision.
- If your model operates over a candidate index, such that the index is eliminated by the time a downstream constraint or objective is computed, then the index is inferred to be intrinsic.
- If an index is explicitly declared as intrinsic, or inferred to be intrinsic, in another decision or constraint within the optimization problem, then it is usually taken to be intrinsic to the decision or constraint in question.
- Extra dimensions in a decision's domain, bounds, or initial guess that aren't inferred to be intrinsic are inferred to be extrinsic. Extra dimensions found in the definition are inferred to be parametric and ignored for the purposes of optimization.
- Given the explicit and inferred intrinsic index assignment for all decisions, any indexes that end up in the objective are taken to be extrinsic.
- Extra dimensions in a constraint are inferred to be intrinsic if they appear specified elsewhere in the optimization as intrinsic, otherwise, they are inferred to be extrinsic.

Tip If there is any question about index status, consider the question, "Does the optimizer need to consider all elements of this index for an overall solution?" If YES, list the index in the Intrinsic Indexes attribute.

Initial Guess

Initial guesses are utilized for optimizations that use gradient search methods. These include non-convex QP problems and all NLP problems. Initial guess values will be ignored if they don't apply to the problem type.

By default, the defined value of the input decision node will be used as an initial guess for the optimization. Therefore, the initial guess attribute is hidden by default. If used, the initial guess attribute will override the defined value of the node.

To use the defined value of the Decision node as an initial guess:

- (No action necessary. Keep the Initial Guess attribute hidden)

To enter an Initial Guess that overrides the defined value of the Decision input node:

- Select **Attributes** from the **Object** menu.
- Toggle a check mark next to **Initial Guess**.
- Open the Object window of the Decision node.
- Enter override value in the **Initial Guess** attribute.

You can use general Analytica expressions as initial guesses, including expressions that are conditional or array valued.

Tip The Initial Guess attribute is a good way to run multiple optimizations with different starting points. If the Initial Guess expression evaluates to an array with dimensions that are not intrinsic elsewhere in the optimization, Analytica will perform separate optimizations for each starting point.

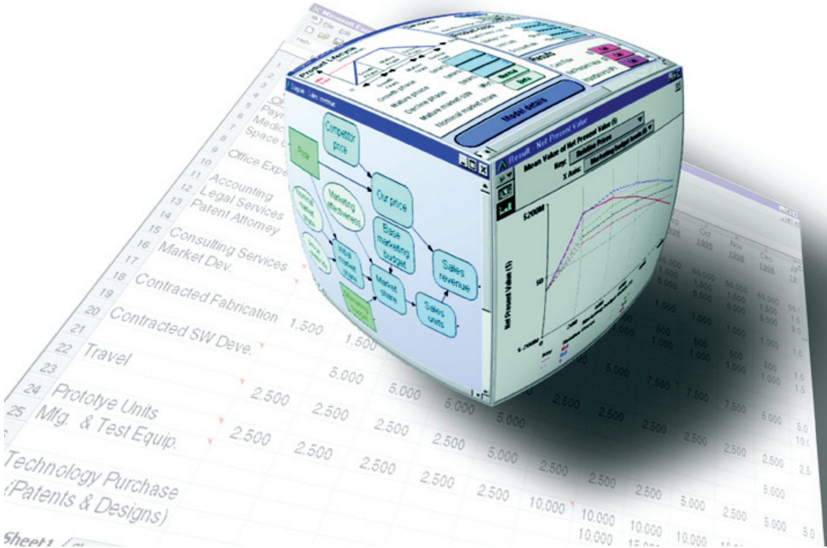
To return Initial Guess to the default state (not visible):

- Delete all Initial Guess expressions.
- Select **Attributes** from the **Object** menu.
- Toggle off the check mark next to **Initial Guess**.

Chapter 7 Optimizer Function Reference

This chapter lists and defines Analytica optimization functions current in versions 4.3 and higher:

- **DefineOptimization()**
- **OptSolution()**
- **OptObjective()**, **OptObjectiveSa()**
- **OptStatusText(), OptStatusNum()**
- **OptInfo(), OptEngineInfo()**
- **OptShadow(), OptRhsSa()**
- **OptReducedCost()**
- **OptSlack()**
- **OptRead(), OptWrite()**
- **OptFindIIS(), OptWriteIIS()**



Using Named Parameters

When calling a function in Analytica, you can use the conventional method of listing parameters in their standard sequence, or named-parameter syntax, where you type the parameter name, followed by a colon (:), followed by the parameter value. Here is an example:

```
DefineOptimization(
    Decisions: [d1, d2],
    Constraints: [c1, c2, c3],
    Maximize: x)
```

Since **DefineOptimization()** has a large number of optional parameters, named-parameters are much more convenient to write and read. So, we use that method in our examples.

You can view the full parameter declarations from Analytica, in the actual parameter order, by selecting **Definition > Optimizer > <*function*>** from the Analytica menu.

Primary Optimization Functions

Using just these three functions, you will be able to define any optimization and view results.

- **DefineOptimization**()
- **OptSolution**()
- **OptObjective**()

DefineOptimization()

This function defines an optimization problem. It automatically determines the type of optimization: LP, QP, or NLP.[3] If you want to specify a particular type of optimization instead of having Analytica make the choice auto-matically, use **DefineOptimization()** with the optional **Type** parameter (see sub-section below).

When DefineOptimization() is evaluated, it returns a special object, which displays as: «LP», «QP», «QCP», «NCQCP», «NLP», or «NSP», depending on the type of problem. This object encapsulates an encoding of the problem definition that is passed to the solution functions.

Tip Double-clicking on the special object in the DefineOptimization() evaluation window will display a new window containing details about the optimization. The new window may also contain clickable objects that let you drill down to more specific levels of detail.

[3] DefineOptimization() replaces **LpDefine()**, **QpDefine()** and **NLPDefine()** used in Analytica Optimizer releases previous to 4.3. These legacy functions are still supported for backward compatibility with older models. See the Analytica Wiki **wiki.lumina.com** for details.)

Parameters of DefineOptimization()

Decisions *type*: variable

A list of identifiers of decision variables. The Optimizer searches for decision values that maximize (or minimize) the Objective and/or meet Constraints. Variables should be separated by commas, and (optionally) enclosed in square brackets . Use **All** to include all decision variables in the model, or **All in *m*** to include all decision variables in a module, *m*.

Examples:

```
DefineOptimization(Decisions: [Height, Radius],...)
DefineOptimization(Decisions: All,...)
DefineOptimization(Decisions: All in LP_Module,...)
```

Constraints type: variable (optional)

A list of Constraint node identifiers, or a direct equality or inequality expression. Use **All** to include all constraint nodes in the model, or **All in *m*** to include all constraint nodes in a module, *m*. May be omitted for unconstrained optimizations.

Examples:

```
DefineOptimization(...,Constraints:[Height_limit, Volume_limit],...)
DefineOptimization(...,Constraints: All,...)
DefineOptimization(...,Constraints: All in LP_Module,...)
DefineOptimization(...,Constraints: 4*x+3*x*y-y^2 >= z^2, ...)
```

Minimize / Maximize *type*: expression (optional)
Synonyms: **min** / **max**

Use one of these parameter names to identify the Objective variable or expression to minimize or maximize. The value of the Objective must be scalar for each optimization run. You may omit this parameter for constraints-only optimizations where the goal is simply to find some feasible solution meeting all constraints.

Examples:

```
DefineOptimization(...,Minimize: Surface_Area,...)
DefineOptimization(...,Minimize: 2*pi*R^2+2*pi*R*H,...)
```

If the variable or expression evaluates to an array, it performs a separate optimization for each value.

For example:

```
DefineOptimization(...,
   Maximize: Array(Objective_index,[Profit, Revenue]),...)
```

will perform two optimizations, to maximize Profit and Revenue respectively, returning the results as an array.

Guess *type*: expression (optional)
Synonyms: **InitialGuess**, **InitialValue**

Indicates an initial guess for optimization types that may have local minimum. Initial guesses have no effect on LP and convex QP optimizations. To enter guess parameters for multiple decision variables, separate by commas in the same order as the decision nodes are listed after the **Decisions** parameter.

Example applying the same guess value to every element of a decision array:

```
DefineOptimization(Decisions: Decision_Array, Guess: 100)
```

Example applying different guesses to different decision variables:

```
DefineOptimization(
    Decisions: Height, Radius,
    Guess: 12, 7)
```

If you have specified *N* decisions, you can list up to *N* guess values (or expressions) separated by commas. The guess values are placed in positional correspondence with the decisions listed — in the example, 12 is used as the guess for **Height**, 7 as the guess for **Radius**. You can pass Null for for any guess, in which case the guess obtained from the decision object is not overridden by the parameter, e.g.:

```
DefineOptimization(
    Decisions:Height, Radius,
    Guess: Null, 7 )
```

When only one guess parameter is specified, then it applies to all decisions. If you wish to override only guess for the first parameter, then you should specify the second guess as Null.

When an expression passed to the **guess** parameter evaluates to an array, the array value is used as the guess. Any indexes that are extrinsic to the decision thus result in an array abstraction with multiple optimizations from distinct starting points. Any indexes that are intrinsic to the decision are consumed, specifying an array-valued initial guess.

Unless you have a specific reason to enter a guess value as a **DefineOptimization()** parameter, it is recommended to enter it as a Decision attribute instead. When guess values exist in both locations, the **DefineOptimization()** parameter will override the Decision variable's attribute.

Domain and Bounds

type: domain expression (optional)

You can use domain expressions to determine variable types and bounds. When used as a parameter in DefineOptimization, the expression follows the word **Domain** as in these examples:

When only one domain specification is listed, it applies to all decision variables. In this example, the **domain** parameter overrides the domain of both `Height` and `Radius` to force a continuous optimization with a lower bound of zero:

```
DefineOptimization(
    Decisions: Height, Radius,
    Domain: Continuous(LB:0),
    ...)
```

When you have listed *N* decisions, you may list up to *N* domain specification, which apply in positional correspondence with the decisions:

```
DefineOptimization(
    Decisions: Height, Radius,
    Domain: Integer(LB:0), Continuous(LB:0, UB:5),
    ...)
```

Use Null when you do not wish to override the domain for one decision while specifying the domain for others. E.g., to override only the domain for Radius:

```
DefineOptimization(
Decisions: Height, Radius,
Domain: Null, Continuous(LB:0, UB:5),
...)
```

Note: *The only way to specify the integer type and bounds for a local variable that used as a decision variable is by using the domain parameter.*

For details on Domain expression syntax, see Domain and Bounds section in the Attribute Reference chapter.

Tip Unless you have a specific reason to enter domains and bounds as a **DefineOptimization()** parameter, it is recommended to enter them into the Decision's Domain attributes instead. When domain specifications exist in both locatons, the **DefineOptimization()** parameter takes precedence.

Type *type*: text (optional)

Indicates the type of optimization problem. If omitted, Analytica analyzes the decisions, objective, and constraints to determine the problem type automatically. This parameter is useful if you expect a particular type of optimization (LP, for example) and you want it to give a warning if the model characteristics don't match your expectation — for example, if an expected LP has non-linear characteristics. Type setting can also affect the choice of engine and processing speed.

Quotation marks must be included since this parameter is passed as text. Example:

```
DefineOptimization(...,Type: "QP")
```

Possible Type values are:

- **"LP": Linear Program**
  Optimization is limited to linear programs only. An error will display if the optimization has any non-linear characteristics.
- **"QP": Linearly Constrained Quadratic Program**
  Most restrictive QP category. It displays an error if constraints are not linear.
- **"QCP": Quadratically Constrained Program.**
  Solution set may be non-convex. The solution may depend on initial guess. Optimizer will spend some processing time evaluating convexity characteristics.
- **"NCQCP": Non-Convex Quadratically Constrained Program**
  Least restrictive QP category. Solution set may be non-convex. Optimum solution may depend on initial guess. Faster than QCP setting for non-convex problems since the optimizer will not spend time evaluating convexity characteristics. May be slower than QCP setting for convex solution sets.
- **"NLP": Non-linear Program**
  Smooth non-linear program. Optimization uses gradient and Jacobian based strategies that assume continuous functions.
- **"NSP": Non-smooth Program**
  Selects the Evolutionary engine. Suitable for:
  - Hard integer problems for which non-integer values cannot be computed or are not meaningful
  - Problems with discontinuous relationships between decisions and solutions.

SetContext *type:* variable list (optional)

SetContext avoids unnecessary computation when solving non-linear programs (NLPs) with extinsic indexes present in the model. An extrinsic index is an index along which multiple individual optimizations are performed. For any given optimization run, only a single element of the index is relevant. As the optimization engine runs repeatedly re-evaluates your model at different search points, arrays indexed by extrinsic indexes will be repeatedly recomputed. Since only one element of an extrinsic index is relevant, all but a single slice of the array will be discarded by the optimization engine. This inefficiency can prolong NLP computation times by orders of magnitude. If more than one extrinsic index is involved, the computational inefficiency could scale quadratically or cubically, etc. with the size of the indexes.

This inefficiency can be avoided by listing a set of *context variables* after the SetContext parameter. When an optimization begins, the full value of the context variable is replaced with just the slice corresponding to the coordinates of the optimization that is active. When used appropriately, this prevents the extrinsic index from being propagated downstream from the context variable, and thus saves unnecessary computation.

The best *context variable* candidates are arrays that include extrinsic indexes and are computed only once for the optimization (i.e. they are not downstream of a Decision variable). Variables for which context is set will not propagate extrinsic indexes to downstream arrays during optimization. The set of context variables should be inclusive enough eliminate extrinsic indexes from every downstream array that is repeatedly evaluated during the optimization (i.e. every array positioned in the influence stream between a Decision variable and the Optimization node). But the set should not include the repeatedly evaluated arrays themselves.

Example:

```
DefineOptimization(...,
   SetContext: Base_demand, Price_elasticity, Discount_rate)
```

Engine *type:* text (optional)

Usually, the Optimizer chooses a solver engine to match the problem type. But, you can use this parameter to specify which engine you want to use. To see a list of installed engines, evaluate the expression `OptEngineInfo('All', 'Name').`

Quotation marks must be included since this parameter is passed as text. Example:

```
DefineOptimization(..., Engine: "GRG Nonlinear")
```

Standard Engines:
The following engines come standard with Analytica Optimizer:

- **"LP/Quadratic"**
  The LP/Quatratic engine handles linear programs and quadratic programs with linear constraints. It implements Primal and Dual Simplex methods plus a Quadratic extension. Memory is efficiently managed using a sparse representation of the LP simplex matrix. For integer domains, the engine first computes a continuous solution, then uses a Branch and Cut method to find the best integer solution. The number of decision variables is limited to 8,000 or less.

- **"SOCP Barrier"**
  This engine uses a Second Order Cone Programming technique designed specifically for quadratically constrained convex problems. The GRG Nonlinear engine is often a good alternative for this type of problem, especially if the constraints end up being non-convex. The number of decision variables is limited to 2,000 or less.

- **"GRG Nonlinear"**
  The Generalized Reduced Gradient nonlinear engine is suitable for problems that are relatively smooth with few local optima. Problems that allow gradients and Jacobians to be computed analyitically will run much faster than problems that require finite difference methods. The number of decision variables is limited to 500 or less.
- **"Evolutionary"**
  The Evolutionary engine is the best choice for problems that do not allow non-integer values to be explored during the intermediate steps of a search. These include hard integer problems and problems with a large number of discontinuities. The number of decision variables is limited to 500 or less.

Add-On Engines:
Lumina offers the following add-on engines for advanced applications. Please visit www.lumina.com for purchase details.

- **Large-Scale LP/QP**
  Allows an unlimited number of variables and constraints. It has been used to solve problems with millions of variables.
- **Large-Scale SQP**
  The most versatile solver for large scale problems. Handles linear, quadratic, conic, smooth nonlinear, and non-smooth problems with no fixed limits on problem size.
- **Large-Scale GRG Solver**
  Extends the variable limit of the GRG nonlinear engine to 12,000 decision variables. It uses sparse matrix storage methods for memory efficiency, and advanced methods for selecting a basis and dealing with degeneracy.
- **Gurobi**
  Gurobi is the fastest available engine for linear and mixed-integer linear (LP/MIP) problems and also handles QP. It has been engineered to expliot multi-core processors more effectively than other solvers.
- **XPRESS**
  XPRESS is designed for high-speed performance on standard LP and mixed-integer linear and quadratic problems (LP/MIP and QP/MIP). It has no fixed limit on the numbers of variables or constraints.
- **MOSEK**
  MOSEK handles large scale LP and QP problems including those with quadratic and second order cone constraints with speed comparable to linear problems. It inclues both simplex and self-dual interior point methods. Also handles convex non-linear problems. It has no fixed limit on problem size.
- **KNITRO**
  KNITRO is designed especially for smooth nonlinear problems. It includes state-of-the-art implementation of interior point nonlinear methods and "active set" Sequential Linear / Quadratic Programming techniques (SLQP). It has no fixed limit on problem size.
- **OptQuest**
  OptQuest is designed to work will all types of models including those with discontinuous functions. It uses advanced methods including tabu search and scatter search to identify solutions that are globally optimum or close to globally optimum. Supports up to 5,000 variables and 1,000 constraints, though the practical size of problems that can be solved to near global optimality may be less than these limits.

Over *type:* index list (optional)

The **Over** parameter forces array abstraction *over* an index, resulting in separate optimization runs for each element of the index. To abstract over multiple indexes, enter a list of index identifiers separated by commas:.

Example:

```
DefineOptimization(...,Over: Scenario_Index, Objective_Index,...)
```

TraceFile *type:* text (optional)

Creates a trace file containing details about the optimization process. Including the filepath is optional. The file will appear in the **CurrentDataDirectory**, usually the same directory as the model unless changed, if you omit the path.

The traceFile can be extremely helpful when debugging convergence issues, i.e., why didn't the optimization find a solution, or didn't find the solution I expected it to find?

Examples:

```
DefineOptimization(...,TraceFile: "c:\ana_models\mytracefile.log")
DefineOptimization(...,TraceFile: "mytracefile.log")
```

The traceFile is written only when a model is solved by an algorithm that repeatedly evaluates the model, which generally means that a non-linear search algorithm must be used. The **TraceFile** parameter is ignored when solving linear or quadratic problems via simplex or Barrier methods.

Tip To produce a traceFile for a linear or quadratic problem, specify Engine:"GRG Nonlinear" to force the use of a non-linear search algorithm. Remove this setting once you've finished your debugging.

Engine Settings Each optimizer engine has a list of setting commands. You can pass setting commands to the optimizer engine using the Parameter and Setting parameters. These are text values that must be enclosed in quotation marks.

See Control Settings chapter for details on available settings.

You can set parameters for the optimization engine using the following syntax:

```
SettingName: "setting_name", SettingValue: setting_value
```

The parameter name and setting name must be inside quotes since they are text strings passed directly to the optimization engine. To change more than one setting, **SettingName** and **SettingValue** must be arrays that have exactly one common index.

See Settings chapter for a full list of available settings and value ranges.

OptSolution(Opt, *Decision, PassNonFeasible*)

The **OptSolution()** function causes the optimization problem to be solved and returns optimized values of the decision variables. Analytica does not change the values of the input decision nodes, nor does it reveal the optimization results when you evaluate the optimization node.

Parameters:

Opt *type*: variable

Identifies the node defined using **DefineOptimization()**.

Decision *type*: variable (optional)

The Decision parameter identifies the decision input node containing the desired values. If you omit the Decision parameter, **OptSolution** lists all scalar decision variables as a flattened 1-D array using **.DecisionVector** as an local index.

If you have more than one decision node, it is usually convenient to have a separate **OptSolution** counterpart for each input node. This way, your optimized results will be in the same format as the input decision nodes.

PassNonFeasible *type*: boolean (optional)

If omitted or set to 0 (FALSE) **OptSolution()** will be «null» if no feasible solution is found. When set to 1 (TRUE), **OptSolution()** will return the last computed solution before infeasibility was determined.

OptObjective(Opt, *PassNonFeasible*)

The **OptObjective()** function causes the optimization problem to be solved and returns the value of the objective at the final solution. If multiple optimizations are performed, **OptObjective** returns an array along extrinsic indexes.

Parameters:

Opt *type*: variable

Identifies the object defined using **DefineOptimization()**.

PassNonFeasible *type*: boolean (optional)

If omitted or set to 0 (FALSE) **OptObjective()** will be «null» if no feasible solution is found. When set to 1 (TRUE), **OptObjective()** will return the objective value corresponding to the last computed solution before infeasibility was determined.

OptObjectiveSa(Opt, *Decision*)

For LP problems only. Performs sensitivity analysis on objective value relative to decision variables. The optimal solution for an LP occurs at the intersection of some subset of constraints, which is a vertex in the simplex of feasible solutions. This set of constraints is often referred to as the basis of the solution. If we were to increase or decrease one linear objective coefficient slightly, the value of the objective would change, of course, but over some range of values the optimal solution would remain at the same vertex (basis). **OptObjectiveSa()** computes the the range (i.e., lower and upper limit) over which each linear objective coefficient can be changed without causing a change in the basis for the optimal solution. The result is indexed by a local index `.Range` having values `["lower","upper"]`.

Parameters:

Opt *type*: variable

Identifies the object defined using **DefineOptimization()**.

Decision *type*: variable (optional)

If Omitted, the function will return sensitivies for relative to all scalar decision variables flattened into a 1-D local index named **.DecisionVector**. If the Decision parameter is included, it will return sensitivities for that decision with the same dimensionality as the decision along with the **.Range** index.

Optimization Status Functions

Optimization status functions can reveal important details such as the selected engine matrix coefficients, and other information that may be helpful with troubleshooting.

OptStatusNum(Opt) / OptStatusText(Opt)

Returns the status number as an integer and corresponding text message, respectively, of the optimization problem Opt. It is wise to examine the status before evaluating **OptSolution()** to avoid an error message. Possible results are shown in the table below.

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

Status Number	Status Text
22	Cone overlap status. Returned when a variable appears in more than one cone.
999	Exception occurred status. Returned when an exception has been caught by try/catch top-level.
1000	Custom base status. (Base for Solver engine custom results.)
1102	The quadratic constraints are non-convex, the SOCP engine cannot solve this problem.

OptInfo(Opt, "Item", *Decision*, *Constraint, asRef*)

OptInfo() is a varsatile function that can reveal any available details of the optimization. The most common information available through OptInfo() can also be viewed by evaluating the DefineOptimization() function and doulble-clicking the encoded object (e.g. <<LP>>). This opens a hierarchy of information and click-able reference objects that reveal finer levels of detail.

Parameters:

Opt *type*: variable

Identifies the node defined using DefineOptimization().

Item *type*: text

The characteristic of the optimization you are interested in.

Decision*, *Constraint *type*: variable, optional

Optional *Decision* and *Constraint* parameters can filter information to be relevant to individual decisions and constraints.

asRef A Boolean value (0 or 1). If TRUE (1), the result will be encoded in a click-able reference object.

The following table shows the relevance of various information items to optimization engines, along with a description of the information revealed.

Item	LP	QP	QCP	NLP	Description
"All"	●	●	●	●	Returns a compregenive view inside the optimization, listing most items shown here. You can see the same information by double-clicking the Optimization object displayed when the DefineOptimization() function is evaluated: («LP», «NLP», etc.)
"DecisionVector"	●	●	●	●	Lists all scalar decision variables in a one-dimenansional list.
"ConstraintVector"	●	●	●	●	Lists all scalar scalar constraints in a one-dimenansional list.
"Decisions"	●	●	●	●	Lists the names of each structured decision array in the optimization.
"Constraints"	●	●	●	●	Lists the names of each structured constraint array in the optimization.
"ObjCoef"	●	●	●		Lists scalar objective linear coefficients.
"Q"		●	●		Displays the matrix of coefficients in the quatratic objective matrix
"Lhs"	●	●	●		Displays the matrix of linear constraint coefficients. You may optionally specify «constraint» to obtain the coefficients for just one constraint corresponsing to all scalar decision variables. Or you may specify both «decision» and «constraint» to get the coefficients for one decision and one constraint.

Item	LP	QP	QCP	NLP	Description
"LhsQ"			●		Displays the matrix of quadratic constraint coefficients for QCP programs. Dense matrixes may be too large to fit in memory for some large QCPs. You may optionally specify «decision» and/or «constraint» to obtain the quadratic coefficients for just that structured decision and constraint.
"Rhs"	●	●	●	●	Displays right-hand side coefficients for all scalar constraints. You may optionally specify <<constraint>> to obtain the coefficients for just one constraint corresponsing to all scalar decision variables. Or you may specify both <<decision>> and <<constraint>> to get the coefficients for one decision and one constraint. For a linear or quadratic constraint, the RHS will be the constant term. There is no guarantee of the sign, since it depends on how DefineOptimization re-arranges the constraint when it processes the coefficients. For a non-linear constraint, Rhs will usually be 0. For a range constraint, e.g., a <= f(x) <= b, the far right constant (b) is returned. It is better to use "constraintLb" and "constraintUb" for range constraint
"ConstraintUb"	●	●	●	●	Upper bound for each scalar constraint. You may optionally specify «constraint» to obtain the values for a single structured constraint.
"ConstraintLb"	●	●	●	●	Lower bound for each scalar constraint. You may optionally specify «constraint» to obtain the values for a single structured constraint.
"Sense"	●	●	●	●	Shows the inequality operator for each scalar constraint ('<=','<=','=') or 'R' for Range (lb & ub). You may optionally specify «constraint» to obtain the values for a single structured constraint.
"Lb"	●	●	●	●	Lower bound for each scalar variable. You may optionally specify «decision» to obtain the value for a single decision array.
"Ub"	●	●	●	●	Upper bound for each scalar variable. You may optionally specify «decision» to obtain the value for a single decision array.
"IntegerType"	●	●	●	●	The type of all scalar decision variables. Optionally you may specify «decision» to get the integer type(s) for a single decision array. Possible values are: ('Continuous','Integer','Boolean','Grouped Integer', or 'Semi-Continuous)'
"Group"	●	●	●	●	Applies only to a grouped integer variable, returns the group number for each scalar decision variable. You may optionally specify «decision» to obtain the groups for a single decision array.
"Maximize"	●	●	●	●	Indicates whether the optimization maximizes an objective ('TRUE') or minimizes an objective ('FALSE'), or whether there is no objective at all for a constraints-only problem ('null')
"Engine"	●	●	●	●	Indicates the engine chosen by Analytica or by user override.
"Settings"	●	●	●	●	Displays a list of engine setting names and corresponsing setting values.
"Type"	●	●	●	●	The problem type. This matches the object displayed when DefineOptimization() is evaluated: ('LP','QP','QCP','CQCP','NCQCP','NLP','NSP')
"Intrinsic Indexes"	●	●	●	●	Displays a table of indexes of Decision and Constraint arrays that are intrinsic to the optimization.
"Extrinsic Indexes"	●	●	●	●	Displays a table of indexes of Decision and Constraint arrays that are abstracted, resulting in multiple optimizations. You may optionally specify either <<decision>> or <<constraint>> to get the extrinsic indexes for a single array. This option is very useful when an index is abstracted unexpectedly. The Extrinsic Indexes display can point you to the source of the extra dimension, just like Stephen Hawking.
"Decision Intrinsic Indexes"	●	●	●	●	Lists all decision arrays in the optimization, and for each, a set containing the indexes that are intrinsic to the decision variable. You may optionally select a single decision node by specifying <<decision>>.
"Decision Extrinsic Indexes"	●	●	●	●	Lists all decision arrays in the optimization, and for each, a set containing the indexes that are extrinsic to the decision variable. You may optionally select a single decision node by specifying <<decision>>.
"Objective Dims"	●	●	●	●	Lists all dimensions of the Objective that warrant array abstraction, resulting in an array of optimizations. Since Parametric indexes of Decision variables are ignored by the optimization, they are not listed in the OptInfo() result even though they can be seen in the evaluation of the Objective array.

OptEngineInfo("Engine", "Item", *asRef*)

The SolverInfo function provides information about a specific optimizer engine, or about the solver engines that are currently installed and ready for use.

Parameters:

Engine *type*: text

The name of a solver engine. The following are included with Analytica:

- "Lp/Quadratic"
- "SOCP Barrier"
- "GRG Nonlinear"
- "Evolutionary"

Various add-on engines can be purchased separately. These include:

- "LSLP"
- "LSGRG"
- "LSSQP"
- "Knitro"
- "OptQuest"
- "MOSEK"
- "XPress"
- "Gurobi"

The engine parameter can be specified as "All" to obtain the indicated information for every installed engine.

Item *type*: text

Item	Type	Description
"SettingNames"	numeric	Array of control setting names
"MaxSetting"	numeric	upper bounds for setting
"MinSetting"	numeric	lower bounds for setting
"Default"	numeric	default value for setting
"EngineName"	text	The engine name (null without error if engine not installed)
"DLL"	text	File path to solver engine’s DLL, "" for built-in engines
"TrialPeriod"	numeric	number of days intil Frontline solver trial license expires

Item	Type	Description
"ProblemTypes"	boolean	A list of the problem types handled by each engine
"MaxVars"	numeric	Maximum number of decision variables supported by engine
"MaxIntVars"	numeric	Maximum number of integer variables supported by engine
"MaxConstraints"	numeric	Maximum number of constratints supported by engine
"MaxVarBounds"	numeric	Maximum number of variable bounds supported by engine
"Milliseconds"	numeric	Time spent in computation
"Iterations"	numeric	Number of iterations engine has performed
"Calls"	numeric	Number of function evaluations that have occurred
"Jacobians"	numeric	Number of Jacobian evaluations that have occurred
"Hessians"	numeric	Number of Hessian evaluations that have occurred

OptShadow(Opt, *Constraint*, *PassNonFeasible*) and

Note: *The **OptShadow()** function applies only to LP and QP type problems with continuous decision variables and linear constraints.*

If a constraint is relaxed, i.e., by increasing the right-hand side, b_i, by one unit, how does 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 **OptRhsSa()** function), and is computed by the function

```
OptShadow(Opt)
```

where **Opt** is a linear program object returned by **DefineOptimization()**. The result is indexed by `.ConstraintVector`. Mathematically, the shadow price is given by this equation.

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

This is 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.*

OptReducedCost(Opt, *Decision*, *PassNonFeasible*)

Note: *The **OptReducedCost()** function applies only to LP type problems with continuous decision variables.*

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, any change in the objective function coefficient changes 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

```
OptReducedCost(Opt)
```

The result is indexed by `.DecisionVector`.

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 “Parts of a Linear Program (LP)” on page 17) there corresponds a dual linear program given by this.

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

such that

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

$$\ldots$$

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

The new variables in this program, $y_{1,}y_2, \ldots, y_m$, are the shadow prices, and the slack value for each constraint is 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.

OptObjectiveSa(Opt, *Decision*)

Note: *The **OptObjectiveSa()** function applies only to LP type problems with continuous decision variables.*

If we change a coefficient in the objective function, the solution ($x_1, \ldots, x_n$) continues 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 affected. This range can be computed with this function.

```
OptObjectiveSa(Opt: OptType; Decision: optional)
```

The first parameter, **Opt**, is a linear program defined using **DefineOptimization()**. 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 index `.DecisionVector`. If the range for only a single decision variable (or a small subset) is required, the second parameter, **Decision**, is used to indicate the decision variable for which the sensitivity is to be computed.

The result returned from **OptObjectiveSa()** 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 like this.

```
Var sa:= OptObjectiveSa(Opt) DO
sa[.range='lower']
```

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

OptRhsSa(Opt, *Constraint*)

Note: *The **OptRhsSa()** function applies only to LP type problems with continuous decision variables.*

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

```
OptRHSSa(Opt: LpType; constraint: Optional)
```

This computes the range over which the 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 remains 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 `.ConstraintVector`. If the range is needed for only a single coefficient, the second parameter specifies a Constraint node, and only the range for that constraints in the designated array are computed.

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

OptSlack(Opt, *Constraint*, *PassNonFeasible*)

When you have a constraint

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

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

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

The constraints that have zero slack are of particular interest, since they are instrumental in constraining the optimum. If these constraints are relaxed (e.g., by increasing b_i), a larger maximum value can be obtained. However, as critical constraints are relaxed, other constraints might 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 this function.

```
OptSlack(Opt)
```

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

OptFindIIS(Opt, *newLp*)

***Note:** The **OptFindIIS()** function applies only to LP type problems with continuous decision variables.*

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

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

OptWriteIIS(Opt, filename, *format*)

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()**.

Format values:

- "LP" (or 1): CPLex LP format
- "MPS" (or 2): a legacy format used infrequently
- "LPFML" (or 3): Open Solver Interface[4]

OptRead(Opt, *DecisionVector*, *ConstraintVector*, *format*)

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

-

[4] Also synonymous with "OSI" and "OSIL"

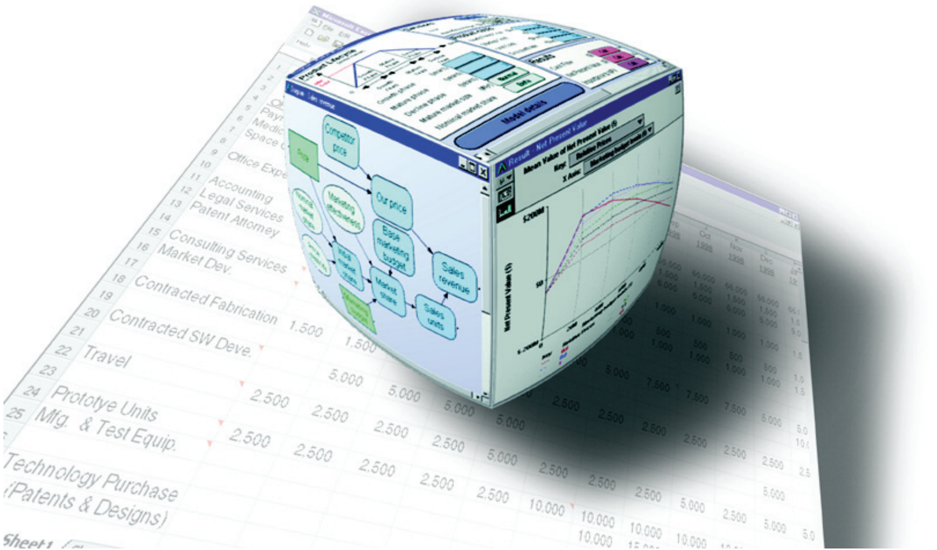
OptWrite(Opt, filename, *format*)

Writes a text description of a Linear Program (LP) or Quadratic Program 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 **OptWrite()** contains the result of only the last **lp**.

Chapter 8 *Control Settings*

This chapter shows you how to:

- Specify Optimizer engine settings in **DefineOptimization()**
- Determine what setting are available for each engine, defaults, and possible range
- Determine size capacities for installed engines
- Control termination criteria during optimization
- Select search algorithms
- Specify numeric precision



Controlling the search

The optimization engine exposes several settings that you can change to influence how the search for the optimum proceeds and when it terminates. The specific collection of available settings is a function of which engine is used to solve the optimization, so that if you install and use an add-on engine, other than the engine that comes standard with Analytica Optimizer, the possible settings might be different. The **OptInfo()** function can be used to view current values for a problem.

To see this, define a variable as:

```
OptInfo(Opt, "Settings")
```

Where Opt identifies the variable containing the **DefineOptimization()** function.

Settings can be changed for a particular problem by specifying values for the **SettingName** and **SettingValue** parameters to **DefineOptimization()**. The first subsection below describes how you specify and view settings, while the subsequent sub-sections detail particular settings used by engines the come standard with Analytica Optimizer.

Selecting the optimization engine

Four optimization engines come standard with Analytica Optimizer:

- **LP/Quadratic:**
  The LP/Quadratic engine uses a dual simplex method combined with branch-and bound for mixed-integer constraints, with a variety of integer cut-set procedures. This is generally the engine of choice for LPs and mixed-integer LPs. For hard mixedinteger LPs, however, the Evolutionary engine uses a very different approach and might be worth trying.
- **SOCP Barrier:**
  The Second Order Cone Barrier engine uses interior point methods designed specifically for quadratically constrained convex problems. The GRG Nonlinear engine is often a good alternative for thi type of problem, especially if the constraints end up being non-convex.
- **GRG Nonlinear:**
  The Generalized Reduced Gradient solver is suitable for smooth non-linear problems. If gradients and Jacobians can be analytically determined, the speed of this method will be dramatically faster.
- **Evolutionary:**
  Best suited for non-smooth problems the evolutionary engine creates a population of potential solutions and keeps the best ones.. By default, the Evolutionary engine does not use gradient information. However, if the **LocalSearch** setting is on, then it optimizes sample points before adding them to the population using various techniques including gradient-based search.

The following matrix shows engine compatbility for each problem type:

		LP/Quadratic	SOCP Barrier	GRG Nonlinear	Evolutionary
LP	Linear Program	●	●	●	●
QP	Quadratic Program (linearly constrained)	●	●	●	●
QCP	Quadratically Constrained Program		●[1]	●	●
CQCP	Convex QCP		●	●	●
NCQCP	Non-Convex QCP			●	●
NLP	Non-Linear Program (smooth)			●	●
NSP	Non-Smooth Program			●	●

[1] You may not know whether your QCP is convex when you formulate it, and DefineOptimization's quadratic analysis does not determine convexity. Testing for convexity can be more computationally intensive than solving the problem, so if you think SOCP Barrier is the preferred engine, you can attempt to solve it using SOCP Barrier. During the solution, it may succeed, or it may detect the non-convexity and terminate without a feasible solution. Always check **OptStatusText()**.

If you have purchased other add-on engines, other options might also be available to you. You can obtain a full list of installed engines and the problem types supported by each by evaluating the following Analytica expression.

```
OptEngineInfo("All","ProblemTypes")
```

To explicitly select the engine to be used, include the **Engine** parameter to **DefineOptimization()**.

```
Engine : Optional Text
```

For example:

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

If you do not specify the engine, Analytica selects an appropriate engine based on the properties of the problem that you specified. However, if the engine does not perform satisfactorily on that problem, you might obtain better results with a different engine.

To determine what engine is actually used on a problem, evaluate this Analytica expression.

```
OptInfo(Opt, "Engine")
```

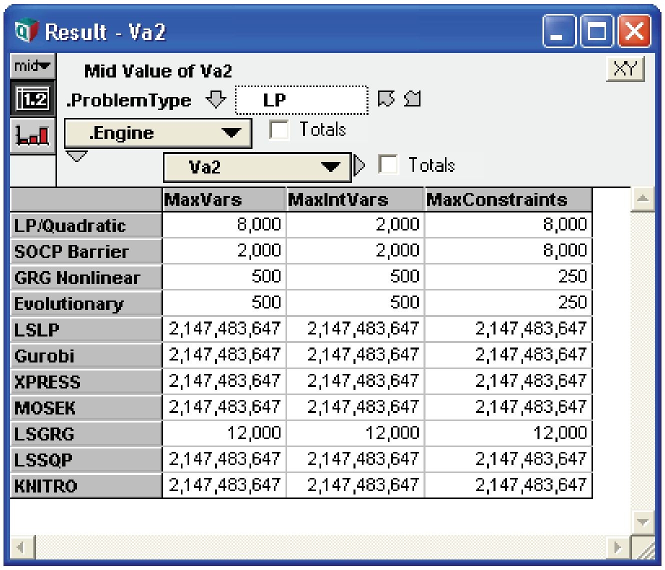
Here `Opt` is the object returned by **DefineOptimization()**.

Examining engine capabilities

Information about the limits on the maximum number of variables or constraints allowed by each installed engine can be accessed using this expression.

```
OptEngineInfo( "All",["MaxVars","MaxIntVars","MaxConstraints"])
```

This returns a table indexed by `.ProblemType`, `.Engine` and the limit type, e.g.:



Mid Value of Va2

.ProblemType LP

.Engine □ Totals

Va2 □ Totals

	MaxVars	MaxIntVars	MaxConstraints
LP/Quadratic	8,000	2,000	8,000
SOCP Barrier	2,000	2,000	8,000
GRG Nonlinear	500	500	250
Evolutionary	500	500	250
LSLP	2,147,483,647	2,147,483,647	2,147,483,647
Gurobi	2,147,483,647	2,147,483,647	2,147,483,647
XPRESS	2,147,483,647	2,147,483,647	2,147,483,647
MOSEK	2,147,483,647	2,147,483,647	2,147,483,647
LSGRG	12,000	12,000	12,000
LSSQP	2,147,483,647	2,147,483,647	2,147,483,647
KNITRO	2,147,483,647	2,147,483,647	2,147,483,647

The problem types displayed include are:

Element	Description
LP	linear program
QP	quadratic objective, linear constraints
QCP	quadratic with convex quadratic constraints (solvers designed specifically for quadratics treat this as if the problem is convex)
NLP	smooth nonlinear
NSP	non-smooth nonlinear

Specifying settings

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

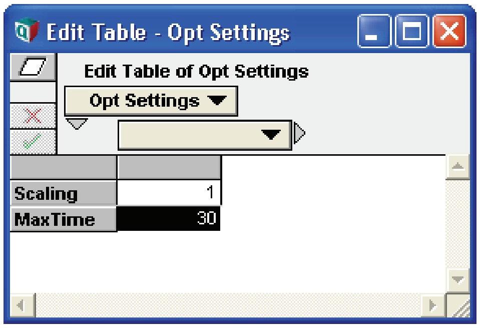
```
DefineOptimization( .., settingName: "Scaling", settingValue: 1 )
```

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

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

1. Drag a variable node to your diagram, title it `Opt Settings`.

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



Scaling	1
MaxTime	30

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

```
DefineOptimization( ..., settingValue: Opt_Settings )
```

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

Examining available settings

The following function returns the set of control settings used for a problem.

```
OptInfo(opt, "Setting")
```

Replace ***opt*** with the name of the variable holding the result from **DefineOptimization()**.

You can also access the range of allowed values for each setting, as well as the default value, using **OptInfo()** or **OptEngineInfo()**. **OptInfo()** is used when you have a problem instance, **OptEngineInfo()** is defined when you know the name of the engine but don’t have a problem instance.

The range (min/max) of possible values for each setting, and the default value, can be obtained using these — first case using an existing problem instance, second case using the engine name:

```
OptInfo( opt,["MinSetting","MaxSetting","Defaults"])

OptEngineInfo("LP/Quadratic",["MinSetting","MaxSetting","Defaults"])
```

Termination controls

Iterations Specifies the maximum number of iterations (pivots) by the simplex algorithm during the optimization. If this is exceeded, **OptStatusNum()** returns 3 (Iterates limit reached. Indicates an early exit of the algorithm.). Maximum number of generations in *Evolutionary* solver. Maximum number of gradient descent steps by *GRG Nonlinear*. If the problem has integer or grouped integer domains, it is preferred to use the MaxSubproblems setting instead of Iterations.

Default: no limit

MaxSubproblems Applies only to problems with integer or grouped integer domains. Places a limit on the number of subproblms the Branch & Bound algorithm explores before pausing and prompting the user to stop or continue.

Default: no limit

MaxIntegerSols Applies only to problems with integer or grouped integer domains. Places a limit on the number of integer solutions the Branch & Bound algorithm explores before pausing and prompting the user to stop or continue.

Default: no limit

MaxTime Maximum number of seconds the Optimizer spends on the problem. If exceeded, **OptStatusNum()** is 10 (Time out status. Returned when the maximum allowed time has been exceeded. Indicates an early exit of the algorithm.).

Default: no limit

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

Default: 30 seconds

Allowed range: positive

IntTolerance In a MIP optimization, if the branch-and-bound algorithm can determine that the best solution found so far is within this relative tolerance of the true optimal, it terminates 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 quickly locates a nearly optimal solution, but then spends 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 continues until the entire search space is eliminated so that the global optimum is reached.

Default: 0%

Allowed range: 0 to 1

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

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

Default: 10^{-4}

Allowed range: 0 or 1

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

Default: 0

Allowed range: 0 to 1

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

Default: 10^{-5}

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

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

Default: no limit

Allowed range: positive

Algorithm selection

Preprocessing

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

Default: False

Allowed range: 0 or 1

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

Default: True

Allowed range: 0 or 1

Engine: LP/Quadratic

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

Default: 1

Allowed range: 0 or 1

Engine: LP/Quadratic

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

Default: 1

Allowed range: 0 or 1

Engine: LP/Quadratic

Debugging

SolveWithout Means "Solve Without Integer Constraints." When this is True, any integer domain constraints are ignored, and the continuous, and the continuous version of the problem is solved instead. The effect is the same as changing the domain to **Continuous** while leaving the variable bounds in, but can be more convenient in some cases when debugging.

Default: True

Allowed range: 0 or 1

IISBounds Determines whether variable bounds should be included in the infeasibility search conducted by **OptFindIIS()** or **OptWriteIIS()**. When set to 1, only a subset of the scalar constraints along the **.ConstraintVector** index is considered. When set to 0, variable bounds can be eliminated in order to find an IIS with a greater number of constraints. This parameter is only used by **OptFindIIS()** when the second optional parameter, **newLp**, is True. When **newLp** is True, **OptFindIIS()** returns a new «LP» object, from which you can use **OptInfo()** to access the list of constraints and list of variable bounds present in the IIS. When **newLp** is False, since only a subset of the **.ConstraintVector** index is returned, **OptFindIIS()** relaxes only constraints, leaving variable bounds in tact.

Default: 0

Allowed range: 0 or 1

Numeric estimation

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

- `1 = forward`: This is the default if **Jacobian** and **gradient** parameters are not supplied. The Optimizer estimates derivatives using forward differencing, i.e.,
$$\frac{\partial}{\partial(x)} \approx \frac{f(x+\Delta)-f(x)}{\Delta}$$
- `2 = central`: The Optimizer estimates derivatives using central differencing, i.e.,
$$\frac{\partial}{\partial x} \approx \frac{f(x+\Delta)-f(x-\Delta)}{2\Delta}$$
- `3 = jacobian`: The Optimizer computes derivatives using the supplied **Jacobian** and **gradient** expressions. This is the default if these are supplied.
- `4 = check`: The Optimizer computes derivatives using the supplied Jacobian expression and also estimates the Jacobian using finite differencing. If they don't

agree to within a small tolerance, the optimization aborts with **OptStatusNum()** = 67 (“error in evaluating problem functions”). This option is useful for testing whether the Jacobian is accurate.

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

Default: 10^{-6}

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

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

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

Default: 0

Allowed range: 0 or 1

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

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

Default: 0

Allowed range: 0 or 1

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

Engine: GRG Nonlinear

Default: 0 (select default)

Allowed range: 0 or 1

SOCP barrier search

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

These parameters are set using the **settingName** and **settingValue** parameters to **DefineOptimization()**, as described in “Specifying settings” on page 92.

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

Default: 0 (off)

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

Engine: SOCP Barrier

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

Default: 1

Allowed range: non-negative integer

Engine: SOCP Barrier

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

Default: 0.99

Allowed range: 0.00 to 0.99

Engine: SOCP Barrier

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

Engine: SOCP Barrier

Default: 10^{-6}

Allowed range: 0 to 1

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

Engine: SOCP Barrier

Default: 10^{-6}

Allowed range: 0 to 1

Evolutionary search controls

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

Engine: GRG Nonlinear, Evolutionary

Default: 0 (automatic)

Allowed range: 0, or integer >= 10

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

Engine: Evolutionary

Default: 0.075

Allowed range: 0 to 1

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

Engine: Evolutionary

Default: 0.5

Allowed range: 0 to 1

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

Engine: GRG Nonlinear, Evolutionary

Default: (use Analytica's random seed)

Allowed range: non-negative integer

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

Default: 0

Allowed range: 0 or 1

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

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

Engine: Evolutionary

Default: 0

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

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

Engine: Evolutionary

Default: 0

Allowed range: 0 or 1

Mixed-integer controls

Integer branch and bound

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

Default: no bounding

UseDual When True, the LP/Quadratic engine uses the ***dual simplex method***, starting from an advanced basis, to solve subproblems generated by the branch-and-bound method. When False, it uses the ***primal simplex method*** to solve subproblems. Use of dual simplex often speeds up the solution of mixed-integer problems.

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

Default: 2

Allowed range:
1 = Primal
2 = Dual

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

Default: 0

Allowed range: 0 or 1

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

Default: 0

Allowed range: 0 or 1

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

```
Constraint Ct1 := X + 2*Y + 3*Z <= 10
Constraint Ct2 := X + 2*Y + 3*Z >= 10
```

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

Default: 0

Allowed range: 0 or 1

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

Default: 0

Allowed range: 0 or 1

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

Engine: LP/Quadratic

Default: 0

Allowed range: 0 or 1

FeasibilityPump An incumbent finding heuristic used by branch-and-bound to find good incumbents quickly.

Engine: LP/Quadratic

Default: 1

Allowed range: 0 or 1

GreedyCover Another incumbent finding heuristic used by branch-and-bound to find good incumbents quickly.

Engine: LP/Quadratic

Default: 0

Allowed range: 0 or 1

Cut generation control

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

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

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

Engine: LP/Quadratic

Default: -1 (automatically determined)

Allowed range: -1or more

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

Engine: LP/Quadratic

Default: 10

Allowed range: 0 or more

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

Engine: LP/Quadratic

Default: 0

Allowed range: 0 or 1

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

Default: 20

Allowed range: non-negative

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

Default: 1

Allowed range: non-negative

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

Engine: LP/Quadratic

Default: 0

Allowed range: 0 or 1

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

Default: 20

Allowed range: non-negative

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

Default: 1

Allowed range: non-negative

ProbingCuts Controls whether probing cuts are generated. Probing involves setting certain binary integer variables to 0 or 1 and deriving values for other binary integer variables, or tightening bounds on the constraints.

Engine: LP/Quadratic

Default: 1

Allowed range: 0 or 1

OddHoleCuts Controls whether ***odd hole cuts*** (also called ***odd cycle cuts***) are generated. This uses a method due to Grotschel, Lovasz, and Schrijver that apply only to constraints that are sums of binary variables.

Engine: LP/Quadratic

Default: 0

Allowed range: 0 or 1

MirCuts, TwoMirCuts Mixed-integer rounding cuts and two mixed-integer rounding cuts.

Engine: LP/Quadratic

Default: 0

Allowed range: 0 or 1

RedSplitCuts Reduce and split cuts are a variant of Gomory cuts.

Engine: LP/Quadratic

Default: 0

Allowed range: 0 or 1

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

Engine: LP/Quadratic

Default: 0

Allowed range: 0 or 1

FlowCoverCuts Controls whether flow cover cuts are used.

Engine: LP/Quadratic

Default: 0

Allowed range: 0 or 1

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

Engine: LP/Quadratic

Default: 0

Allowed range: 0 or 1

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

Engine: LP/Quadratic

Default: 0

Allowed range: 0 or 1

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

Engine: LP/Quadratic

Default: 0

Allowed range: 0 or 1

Coping with local optima

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

Engine: GRG Nonlinear

Default: 0 (off)

Allowed range: 0 or 1

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

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

Engine: GRG Nonlinear, Evolutionary

Default: 1 (on)

Allowed range: 0 or 1

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

Engine: GRG Nonlinear

Default: 0 (off)

Allowed range: 0 or 1

Numeric tolerance and precision

ReducedTol The optimal or reduced cost tolerance. The simplex method looks for a variable to enter the basis that has a negative reduced cost. Decision variables whose reduced cost is less than the negative of this tolerance are candidates for entering the basis during the simplex search.

Default: 10^{-5}

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

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

Default: 10^{-5}

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

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

Default: 10^{-6}

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

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

Engine: LP/Quadratic

Default: 10^{-7}

Allowed range: 0 to 1

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

Engine: LP/Quadratic

Default: 10^{-7}

Allowed range: 0 to 1

Unused

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

Crashing

IntCutoffHigh, deprecated, used **IntCutoff**

IntCutoffLow, deprecated, use **IntCutoff**

PrecisionTol

SolutionAccuracy

SolutionResolution

SolutionTol

VariableReordering