analytica

User Guide

Analytica 3.1 for Windows

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

Copyright Notice

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

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

Contents

Chapter 4: Creating and editing a model

Chapter 5: Building effective models

Chapter 6: Creating lucid Influence Diagrams

Chapter 7: Formatting graphs and tables

Chapter 8: Creating and editing definitions

Chapter 9: Creating models used by others

Chapter 10: Expressions

Chapter 11: Arrays and indexes

Chapter 12: Function reference

Chapter 13: Expressing uncertainty

Chapter 14: Probability distributions

Chapter 15: Uncertainty and sensitivity

Chapter 16: Modeling changes over time

Chapter 17: Importing, exporting, & OLE linking data

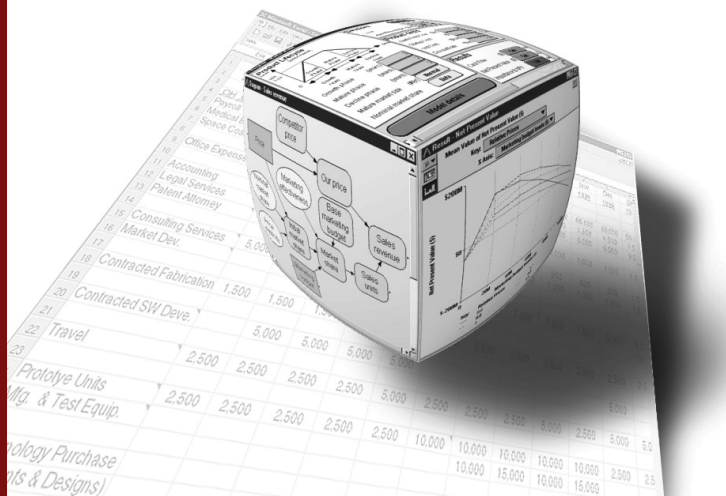
Chapter 18: Working with large models

Chapter 19: Building functions and libraries

Chapter 20: Procedural programming

Introduction

About Analytica



In this Chapter

This introduction tells you how to:

- Use this manual
- Install Analytica
- Use the online help system
- Access Analytica examples

It also reviews the features that are new to release 3.0 and 3.1.

About Analytica

Welcome

This document describes how to use Analytica 3.1 for Windows. If you want a hands-on introduction to the basics of Analytica, we encourage you to start with the Analytica *Tutorial*. This *User Guide* provides more depth and complete coverage.

Click on cross references

If you are reading this Guide on your computer, you can click on any cross reference to jump to that page (see page 8).

If you have any questions or comments about this Upgrade Guide or the Analytica 3.1 software, please email us at support@lumina.com.

If you don't read manuals

You may find that you can use many of Analytica's features without reading this *User Guide*, especially after going through the Analytica *Tutorial*. If so, use this *User Guide* as a reference when you need more help. You may still find it valuable to scan through selected chapters, which give valuable tips you may not otherwise find. We especially recommend these Chapters:

Chapter 5 provides guidelines for creating effective models, distilled from the experience of master modelers. It offers a practical guide for building effective models that are clear, reliable, and focus on what really matters—the decisions, objectives, and key uncertainties. These tips are helpful with any modeling software, but we designed Analytica to make them especially easy to adopt—for example, how to use Analytica's hierarchical Influence Diagrams to make large models comprehensible.

Chapter 6 offers tips on how to create diagrams that are truly lucid, clear and elegant—and how to avoid creating incomprehensible spaghetti diagrams.

Chapter 11 explains Analytica's Intelligent Arrays™. These let you create complex multidimensional models with surprising ease—once you understand the essential concepts. But, prior

experience with spreadsheets or arrays in programming languages may actually get in your way if you don't take a little time to understand them. So, we suggest you scan Chapter 11 if you plan to create models with extensive use of arrays.

Chapter 13 discusses how to select appropriate probability distributions to express uncertainties. It also provides an overview of how Analytica computes probability distributions using Monte Carlo and other random sampling methods, and your options for controlling and displaying probabilistic values.

Requirements

To use Analytica, you need the following minimum configuration:

- 486-66MHz (Pentium 500MHz+ recommended)
- 20MB disk space
- 32MB RAM (256MB recommended or more for large models)
- 8-bit color display
- Windows 98, 2000, NT 4, ME, or XP

Installation and license codes

After downloading the Analytica 3.1 installer from www.lumina.com, or inserting the Analytica CD-ROM into your CD or DVD drive, just double click on the installer to start installation. It will install onto your hard drive the software executable, all documentation as Adobe PDF files, libraries and example models. If you currently have Analytica 2.0 or Analytica 3.0 on your computer, it will leave it there.

The setup program requires some responses from you. For example, you will be asked to verify the directory name in which Analytica will be installed. Most users can accept the defaults provided by the setup program. By default Analytica is installed in C:\Program Files\Analytica 3.1

License codes

When you order or download a copy of Analytica, whether a Player, Trial, or purchased edition, Lumina will email you your license code, which you will need to activate the software. If

someone else purchased Analytica for you, you may need to ask that person to forward you the email with your license code.

During installation, Analytica will asks you for a license code. You can copy and paste from the email or just type it into the field. The license code will activate Analytica with the specified edition (*e.g.,* Player, Trial, Professional, Enterprise). It also activates the Optimizer if you that option.

Stale license codes

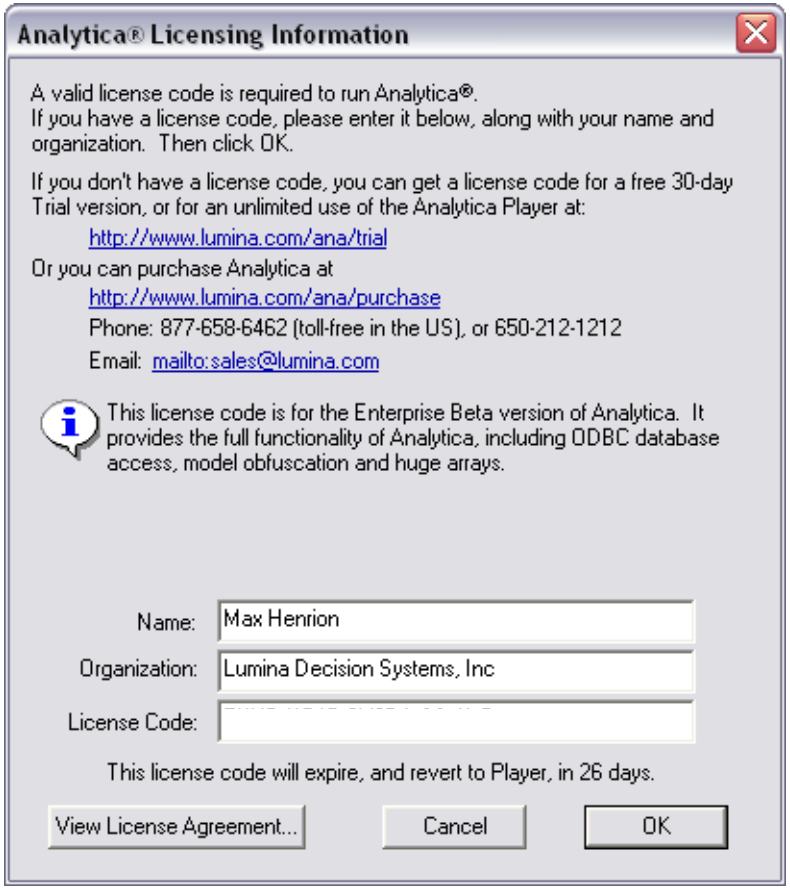
Each license code must be used within a few days, after which it goes ***stale***. If yours is stale—perhaps, because you didn't use it right away, or months later want to install Analytica on another computer—fear not! Click on the URL on the registration screen, or go to http://www.lumina.com/ana/stale. Provide the requested information, and it will immediately email you a fresh license code. The purpose of this mechanism, is to prevent unauthorized use of old license codes. Authorized users can always get a fresh license code.

Expiration dates

Some license codes—notably, for the Trial edition—have a limited life, after which they ***expire***. On expiration, Analytica reverts to the Player edition, so you will still be able to open, view, and evaluate your models. You just won't be able to make or save changes. *This is not the same as going stale.* A second Trial license code will not reactivate the Trial edition on the same computer. To reactivate Analytica after expiration, you may need to purchase a copy.

Updating your edition with a new license code

If you want to change your edition—say, purchase Analytica after testing the Trial edition, or upgrade from the Professional to Enterprise edition—you do not need to download and reinstall Analytica. Just select **Update License** from the **Help** menu to show the **Licensing Information** dialog (see next page) and enter your new license code. **NB:** ADE is a different application, and *does* require installation, even if you already have another edition of Analytica installed.



Uninstalling Analytica 2.0 or 3.0

Analytica 2.0 and 3.0 have uninstallers. If you wish to remove Analytica 2.0, say, select **Programs** from your **Windows Start** Menu. Find the **Analytica 2.0** folder, and select **Remove Analytica 2.0**.

Editions of Analytica

Analytica 3.1 is available in several editions, with varying sets of features. Below are descriptions and a table listing key features.

Professional

The "standard" edition. It provides most functionality, including the ability to create, edit, and save models.

Enterprise

Provides all the features of Analytica Professional, plus Huge Arrays, ODBC library for access to relational databases, profiling for analysis of computational effort by Variable, andobfuscation (encryption) of sensitive model elements. See Chapter 21 for details on Enterprise features.

Player

Lets you review, explore, and run models without purchasing Analytica. With the Player, you can change designated inputs to a model, run the model, view results, and examine selected model diagrams and Variables. It does not let you create new models, make changes other than to selected inputs, or save models.

Power Player

Power Player is designed to distribute applications created in Analytica Enterprise to end users. Power Player users change inputs and import input and output data from external applications and save models with their changes to input values. Power Player users can also access databases, utilize huge arrays, and evaluate all Enterprise-only functions.

Trial

A free edition of Analytica that provides the full functionality of Analytica Professional for a limited time, usually 30 days. After that, it reverts to the functionality of Analytica Player, so you can still view and run any models you have created, but not save changes.

Lite

The Lite edition is available only for educational purposes, teaching and research. It omits these features from the Professional edition: Graphing with Microsoft Excel, OLE hot-linking, ability to create input and output nodes and forms.

The Analytica Decision Engine (ADE)

ADE runs models built with Analytica Enterprise on a server computer. It provides an API (Application Programming Interface) to provide access to view, edit, and run models from another application, including a web server. You can create a user interface to models via a web browser, so that many end users may view and run a model via the Internet.

ADE supports all the modelling features of Analytica Enterprise—that is, all features not associated with the Analytica user interface. The ADE Kit is distributed with a copy of Analytica Enterprise, which serves as the development tool for ADE models.

Optimizer

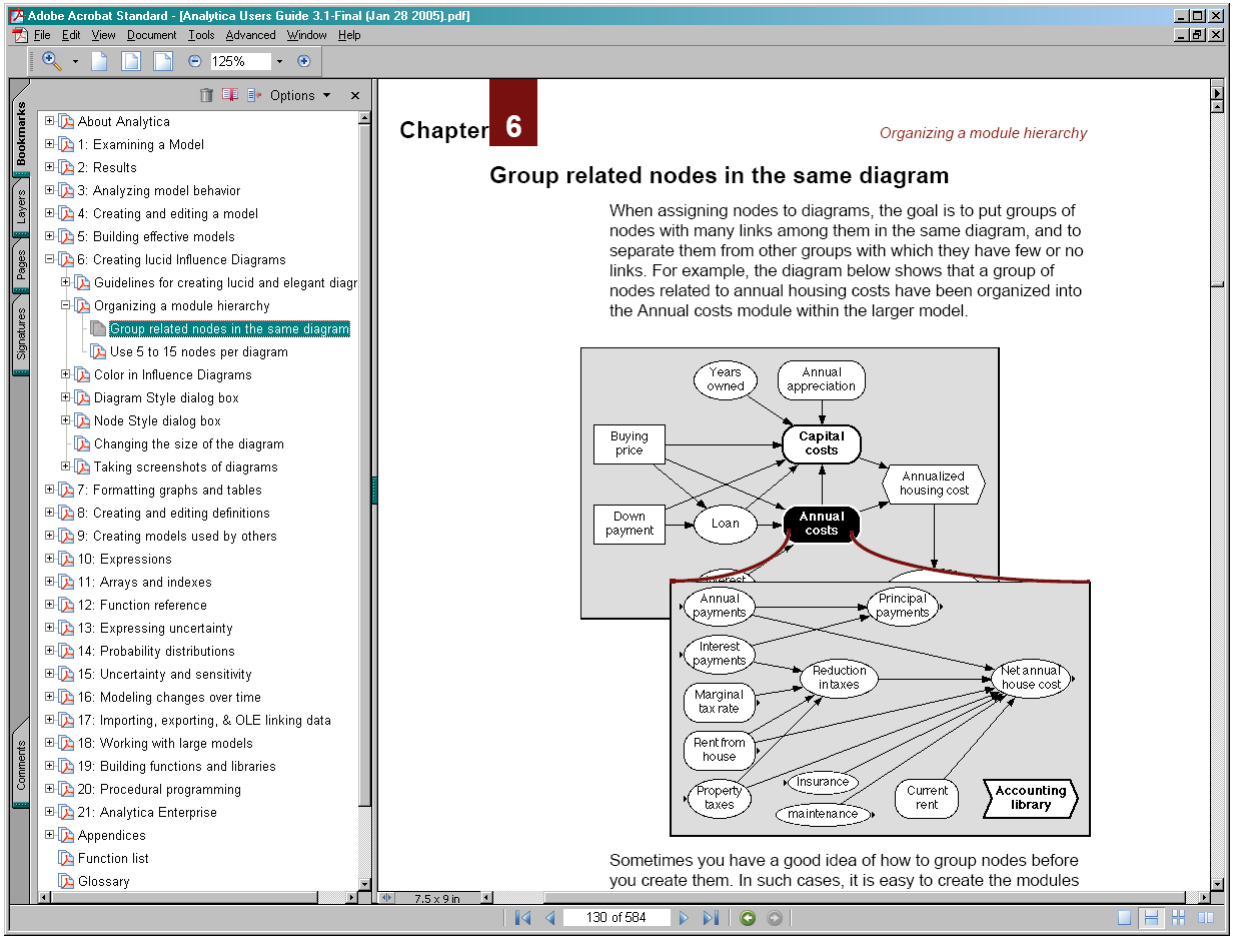
The Analytica Optimizer provides powerful solver and optimization methods, including linear programming (LP), quadratic programming, and nonlinear programming (NLP). The Optimizer is an extension to Analytica Enterprise and ADE. The Optimizer is available as an extension to Analytica Enterprise, Power Player, and ADE. See the *Analytica Optimizer Guide* for details.

Product features by edition

Features	Editions of Analytica 3.1							
	Player	Power Player	Trial	Lite	Professional	Enterprise	ADE Kit	ADE Extra
Open models, change inputs, & view results	✓	✓	✓	✓	✓	✓	✓	✓
Create, edit, and save changed models		-/✓	✓	✓	✓	✓	✓	✓
No marking of printout		✓		✓	✓	✓	✓	✓
Hierarchical influence diagrams	✓	✓	✓	✓	✓	✓	✓	
Monte Carlo uncertainty analysis	✓	✓	✓	✓	✓	✓	✓	✓
Intelligent Arrays™, see page XXX	✓	✓	✓	✓	✓	✓	✓	✓
Procedural programming	✓	✓	✓	✓	✓	✓	✓	✓
Graphing & OLE Linking to/from Excel, see page 139 & page 377	✓	✓	✓		✓	✓	✓	
Outline Window, see page 397	✓	✓	✓		✓	✓	✓	
Create Input and Output controls and Forms, see page 161			✓		✓	✓	✓	
General Function libraries: Math, Array, Distributions, Special, Statistical, Text	✓	✓	✓	✓	✓	✓	✓	
Advanced Function libraries: Advanced math, Financial, and Matrix	✓	✓	✓		✓	✓	✓	
Save browse-only models and hide sensitive model details, see page 487						✓	✓	✓
Huge Arrays™ — dimension or sample size up to 100 million, see page 492						✓	✓	✓
ODBC database access, see page 484						✓	✓	✓
Time and Memory Profiling, see page 493						✓	✓	✓
Optimizer available		✓				✓	✓	✓
Application Programming Interface							✓	✓

Online help and electronic documentation

Analytica provides its User Guide and Tutorial as Adobe PDF documents for online help, available while you are using Analytica. You can read and search these documents using the free Adobe® Reader.



The expandable outline

You can expand and contract chapters and sections of the outline by clicking the  or  icons. Click on a section title to jump to that section.

Magnification

You can change the page magnification and window size to suit your screen and eyesight.

Obtaining Adobe Reader

If you don't already have the Adobe® Reader (formerly known as the Acrobat Reader), you can download a free copy from http://www.adobe.com/.

Accessing help from Analytica

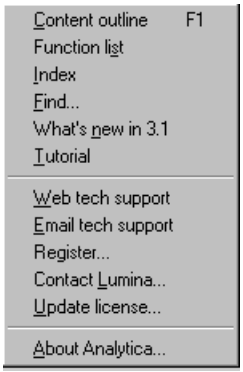
You can access electronic documentation via the **Help** menu (see below) or simply by pressing the *F1* key.

Alphabetical index

If you find too many unhelpful occurrences of the term, try the **Alphabetical Index** in the bookmarks, which usually links to the best explanation for each term.

Help menu

The **Help** menu contains these options:



Analytica Note: *The options that appear on the help menu will vary depending on your computer setup and the version of Analytica you have. If you do not have Adobe Acrobat installed on your computer, the items that appear above the line will change to only:*

- *User guide*
- *Optimizer (if you have purchased the Optimizer)*
- *Tutorial*

Users with free Acrobat Reader version 6.0 or earlier, or users with Acrobat Standard or Acrobat Professional, see the expanded listing shown. Users with the free Acrobat Reader version 7.0 or later will only see the truncated list.

Content outline

Opens the User Guide with Acrobat showing chapters, sections, and subsections as an expandable outline, using bookmarks.

Function list

Opens a page listing all functions, operators, and other constructs, organized by type. Click on a name to jump straight to an explanation of how to use it.

Index	Opens the User Guide to its alphabetized index. Select the first letter of the term from the bookmark outline, and click the term you want to see details.
Find…	Opens the **Find** dialog box in Adobe Acrobat so you can search for a term.
What's new in 3.1	Opens the "What's New in 3.1" section in the Analytica 3.1 User Guide.
Tutorial	Opens the Analytica Tutorial, with the bookmark outline for quick access to any section.
Optimizer help	Opens the Analytica Optimizer Guide (available with Analytica Optimizer only).
Web tech support	Opens Lumina's Analytica tech support web page in your default Web browser, showing answers to frequent questions.
Email tech support	Starts an email using your default email program to be sent to Lumina's tech support.
Contact Lumina…	Opens a dialog box showing Lumina contact information including web and email.
Update license…	Opens the **Licensing Information** dialog box so you can review your or enter a new License Code to upgrade your copy of Analytica.
About Analytica…	Opens the startup flash screen, showing information about Analytica and your License Code

What's new in Analytica 3.0 and 3.1?

Analytica releases 3.0 and 3.1 introduce a wide range of new and improved features. We selected most of these directly in response to users' requests. Below is an overview of the changes since 2.0 (Online viewers may click on page numbers to see details).

New to release 3.0

Procedural programming

Release 3.1 adds most of the standard constructs of a procedural programming language, such as Visual Basic or C, including Begin-End blocks, local Variables, assignment, While loops, recursion, and dimensional specifications in function declarations. These constructs add great power and flexibility for those that like to program. They also enable the creation of convenient

function libraries for the benefit those who do not like to program themselves. (See page 438 for a table summarizing the new constructs and page 435 for an example and overview.)

Local indexes

You can now create index Variables within a Variable definition to identify dimensions of the resulting arrays. These local indexes greatly simplify using Variables and functions that generate arrays with new dimensions. (See page 450.)

Improved support for Intelligent Arrays™

Some users think Intelligent Arrays are Analytica's most valuable feature: It lets you define Variables and functions without having to worry about the number of dimensions of the Variables they use—except when a particular dimension is relevant to the operation. (See page 451 for an overview of the Benefits of Intelligent Arrays.) Most functions and constructs inherently support Intelligent Arrays. Analytica 3.1 adds some new features to help you ensure that all definitions support Intelligent Arrays, even when they use those one of the few constructs that do not. (See page 451.)

Complex data structures

The new reference, \, and dereference, #, operators let you construct complex data types, such as trees and some types of non-rectangular arrays. (See page 458.)

New functions

We have added about 30 new built-in functions, including:

Text Functions: &, Asc, Chr, FindinText, TextUppercase, TextLowercase, TextSentenceCase, JoinText, SplitText, TextReplace, SelectText. (See page 194.)

Probability distributions: Poisson, Geometric, Hypergeometric, Exponential, Weibull, ChiSquared, Logistic, StudentT. (See page 313.)

Others: attrib Of x, CopyIndex, Evaluate, IsUndef, Iterate, Error, MsgBox, ReadTextFile, MDTable, Slice(L,x), SingularValueDecomp. (See Chapter 12, "Advanced Array Functions".)

Web links in models

Model documentation can include hyperlinks (URLs) to any page on the world wide web, to provide detailed explanation, references, and justifications for a model assumptions. (See page 173.)

Electronic user documentation

Online help now provides direct access to electronic copies of Analytica's acclaimed User Guide and Tutorial as Adobe PDF files, using the free Adobe Acrobat reader. The PDF format provides clear layout and illustrations comparable to the printed manuals, with rapid search, an expandable outline, and hyperlinked cross-references, contents, and index. (See page 9.)

XML format for model files

Analytica saves models in new file format, using the hugely popular XML (extended Markup Language). This enables model files to be viewed, edited, and exchanged with a large number of other applications, including Internet Explorer and most database systems. Release 3.1 can still read and write the earlier file formats for upward and backward compatibility. (See page 170.)

Huge arrays

In earlier releases of Analytica, the largest size of an index is 30,000 elements. The Enterprise and ADE editions of Release 3.1 can handle arrays of up to 100 million elements per index—limited only by available memory. Huge Arrays allow a corresponding 3000-fold expansion of sample size for probabilistic simulations and for the size of datasets read in from databases. (See page 492.)

Time and memory profiling

The Enterprise edition records how long it takes to evaluate each variable and will list the most compute-expensive variables to help you optimize large models. (See page 493.)

Speed doubling

Analytica 3.1 contains numerous optimizations which together typically speed up model evaluation by an average factor about two—depending on the model. (See page 413.)

Superseded constructs

New functions and constructs have superseded a few of the old ones. "Forward and backward compatibility" on page 531 lists the superseded functions and their recommended replacements.

New to release 3.1

Parameter qualifiers

Syntax and qualifiers for parameters to user-defined functions: 3.1 introduces a much richer syntax calling parameters, which allows optional parameters and the ability to specify parameters by name instead of by position, analogous to how Analytica lets you specify array indexes by name rather than position. It also offers a much richer set of qualifiers to use in parameter declarations to specify what type of parameter is expected and how it should be treated. Evaluation Mode Qualifiers specify whether parameter should be evaluated deterministically (Determtype), probabilistically (ProbType or Sample) or passed as a Variable name (VarType) or Index (IndexType). Type Checking Qualifiers include Numeric, Positive, TextType, and ReferenceType. Array Type Qualifiers specify whether a parameter should be a single value (Scalar), have one dimension (Vector), or an Array with listed Indexes. They can make sure functions generalize appropriately for array-valued parameters. (See page 422.)

Colors

Analytica 3.1 provides a revised set of default colors for each type of Object in diagrams and the background. You can still change the colors of selected nodes or the background, using **Show Color Palette** from the **Diagram** menu. (See page 120.)

Assignment to global Variables

Analytica desn't let you assign new definitions or values to global Variables in the Definitions of Variables — to keep models understandable and predictable. But, now you can assign to global variables in Button Scripts, or in Functions called from Button Scripts, using *(x := expr)*. This is very useful, for example, if you have a very long computation whose result you want to save for reuse in the future without having to compute it again. You can assign the result value or table as the definition of a variable in a Button Script. You can also assign new values to other attributes (other than Definition) of an Object using *(attrib Of obj := expr).* See "Assignment to a non-local Variable" on page 441.

Better graphing of discrete distributions

When the samples from a distribution are numeric, it is not always clear whether the distribution is discrete or continuous, and thus whether the result should be graphed as a probability distribution function or a probability mass.

Analytica 3.1 makes more intelligent guesses about whether a distribution sample is from a discrete or continuous distribution. When displaying a probability mass or cumulative mass, the solid bar graph style is used, and when displaying a continuous density, the histograph style is used.

Power Player

Power Player allows users to view existing models created with Analytica Enterprise. With Power Player, users can do everything they can do with Analytica Player, plus change inputs and import input and output data from external applications. Unlike the free Analytica Player, users can save models with their changes to input values; they can also access databases, utilize huge arrays, and evaluate all Enterprise-only functions. Power Player is designed to distribute applications created in Analytica Enterprise to end users who need to use models with Enterprise-level features, but do not need the ability to create new models or refine existing ones.

When a browse-only model, created using Analytica Enterprise Edition, is loaded into Analytica Professional or Analytica Lite it runs in Power Player mode.

Optimizer module

The Optimizer module is a powerful, general purpose solver and optimizer, available as an extension to Analytica Enterprise, Power Player, and ADE. It offers efficient linear programming (LP) with up to 1000 decision Variables and 2000 constraints,

mixed integer programming, quadratic programming (QP), and a powerful hybrid classic and evolutionary nonlinear programming (NLP) optimizer for up to 500 decision Variables. For more details, see

http://www.lumina.com/ana/optimizer.html

from where you can download the *Analytica Optimizer User Guide*.

New functions

We have added about 20 new built-in and library functions, including:

Probability distributions: Pert, Log_Normal_m_s_sd (see "14: Probability distributions" on page 301).

Multivariate Distributions: There is a new multivariate distribution library containing the following new functions: Binormal, Correlate_Dists, Correlate_With, Dirichlet, Gaussian, Multinomial, and SampleCovariance (see "Multivariate distributions" on page 327).

Control Functions: CurrentDataDirectory, CurrentModelDirectory , WriteTextFile, MemoryInUseBy,IgnoreWarnings (see "Iteration loops and recursion" on page 443).

Others: IsNotSpecified, CurrentDataDirectory , CurrentModelDirectory, WriteTextFile, MatrixMultiply, EigenDeComp. (See Chapter 12, "Advanced Array Functions".)

New current directory mangement

Analytica 3.1 now maintains two "current" directories: A current model directory and a data directory. When you first load a model, they start out the same (normally "`My Documents`"), but either can be changed through the File Finder dialog or using the `CurrentModelDirectory` or `CurrentDataDirectory` functions. The loading of models and modules use the current model directory, while data functions (`ReadTextFile, WriteTextFile, Export..., Import..`, *etc.*) use the current data directory.

Analytica will use your "`My Documents`" folder by default initially, but you can change this with the registry setting: `HKCU\Software\Lumina Decision Systems\ Analytica\StartFolder`

New example models

Analytica 3.1 comes with new sample models. The Optimizer module comes with ten optimizer example models. In addition, the following areas have new sample models:

- Data Analysis: Principle Components.ana
- Data Analysis: Moving Average Example.ana

- Function Examples: Lookup reindexing.ana
- Dynamic Models: Tunnel through earth.ana
- Decision Analysis: Multi-Attribute Utility Analysis.ana

New keyboard shortcuts

With Analytica 3.1, you can use the keyboard shortcut F2 to switch between edit and arrow modes.

For nodes in same diagram, there is now a shortcut for typing identifier names in an Attribute pane. While typing a definition, depress the ALT key and click on the desired node; the node's identifier is inserted into the definition. (This only works from the Attribute pane.)

Use of up to 3 GB in Windows Application Server

Windows XP, 2000, NT, Me, and 98 limit the memory space of any 32-bit process to 2 Gigabytes. However, Windows Application Server 2003 allows 32-bit processes to access up to 3 Gigabytes. Analytica 3.1 and ADE 3.1 are now able to utilize up to 3 Gigabytes in Windows Application Server.

Consolidated user documentation

Analytica 3.1 documentation has consolidated the previous Users Guide for Alaytica 2.0, the Analytica 3.0 Upgrade Guide, and the documentation for the new Analytica 3.1 features into a single Analytica 3.1 Users Guide.

Conventions used in this guide

This guide uses the following conventions.

Typographic conventions

Example	Meaning
behavior analysis	Key terms when introduced. Most of these terms are included in the Glossary
Diagram	Menus and menu commands
`Sequence()`	Functions
`Price - DownPmt`	Expressions, definitions
Enter	A key on keyboard
Foxes at end	Title or identifier of a Variable or other Analytica Object

Terms for interacting with your computer

Term	Meaning
press	Press and hold down the mouse button
click	Press and release the mouse button once
double-click	Press and release the mouse button twice in quick succession
drag	Press and hold down the mouse button, move the cursor to a new location on the screen, and then release the mouse button
select	Click on an interface Object, such as a node in a diagram or a cell in a table; selected objects appear highlighted

Analytica Note: *These notes give you useful or important information.*

Online help

At any point, you can access Analytica's online help system by pressing the *F1* key or by using the pull-down **Help** menu.

User guide examples folder

In the Examples folder distributed with Analytica is a folder of User Guide Examples. This folder contains an Analytica model for each of Chapters 9, 10, 11, 12, 14, 15, and 16, and three Analytica models for Chapter 17. Use these models to more closely examine the examples shown in this Analytica *User Guide*.

See Chapter 8 in the *Tutorial* for a brief description of all the models contained in the Examples folder.

How to contact us

If you have any questions or comments about Analytica, or just want to keep up to date on changes to Analytica, please contact us.

By mail

Lumina Decision Systems, Inc.
26010 Highland Way
Los Gatos, CA 95033
USA

By electronic mail

- Sales or customer support: info@lumina.com
- Technical support: support@lumina.com

By phone

(650) 212-1212

Web site

For the latest information about Analytica, please visit our Web Site located at

`http://www.lumina.com`

Credits

Analytica *User Guide*

The Analytica *User Guide* was written by Max Henrion, Brian Arnold, Lonnie Chrisman, Fred Brunton, David Dvorkin, and Lynda Korsan with Randa Mulford (Expert Support, Inc.).

The *User Guide for Analytica 3.1* was written and edited by Lonnie Chrisman, Max Henrion, and Richard Morgan. Previous releases were edited by Jason Harlan, Lynda Korsan, Rich Sonnenblick, Brian Sterling, Eric Wainwright, and Randa Mulford and Adrienne Esztergar (Expert Support, Inc.).

Layout design by Mike Marsh.

This document was created electronically using Adobe Framemaker® Release 5.5 and 7.0 for Microsoft Windows®.

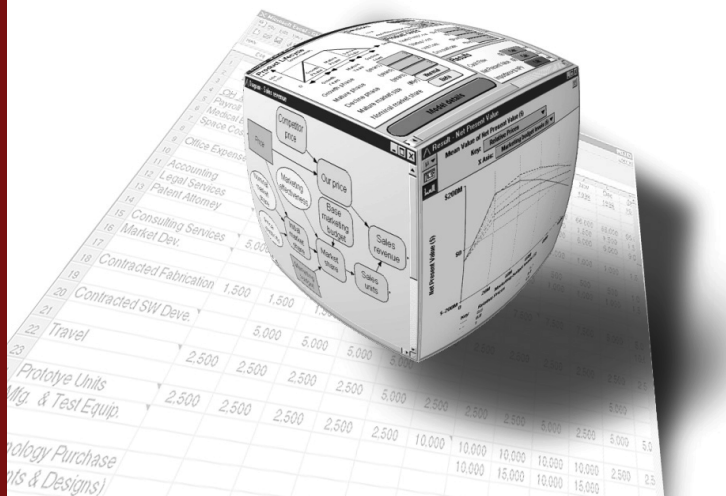
The color used in this manual is Pantone Matching System® 668.

Cover design and production by Zoom Studio, Portland, Oregon.

Printing and binding by Bridgetown Printing, Portland, Oregon.

Chapter 1

Examining a Model



In this Chapter

This chapter shows you how to:

- Start up a model
- Explore its Diagram window
- Explore its Object window
- Explore its Result window
- Print the contents of windows

1: Examining a Model

This chapter introduces the basics of interacting with a model in Analytica. It describes how to start up and explore a model, including its Diagram, Object, and Result windows, and how to print the contents of windows.

Opening, closing, and switching models

Models

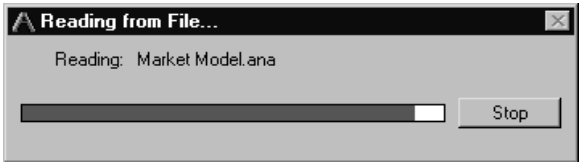
A ***model*** is a collection of Variables and modules used to represent a situation of interest. Between sessions, a model is stored in an Analytica document file with the file type "`.ana`".

Opening a model

To open an existing model, do the following:

1. Start Analytica. The program begins with a new, untitled model open.
2. Click on **File** and select **Open Model** in the pull-down menu. A file dialog prompts you to locate and open a model.

As the model is read, a dialog box indicates progress:



Analytica Note: *Clicking on the* **Stop** *button halts the model reading process and results in a partially loaded model (the diagram will be incomplete).*

After reading in the model, Analytica checks the definitions.

Analytica Note: *Clicking on the* **Stop** *button halts the checking of definitions and results in a diagram with missing arrows.*

If the model contains Variables without syntactically correct definitions, the "invalid Variables" window displays (see “Invalid Variables” on page 403).

Closing a model

To close a model, select **Close Model** from the **File** menu. If you have made any changes to the model, a dialog box asks you whether you want to save the changes before closing.

Switching to another model

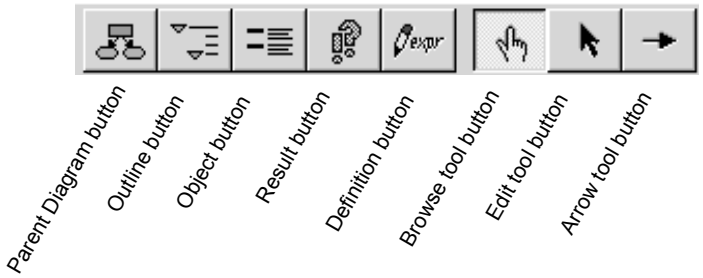
Only one Analytica model can be open at a time. To switch to another model, first close the model. Then select **Open Model** from the **File** menu. A dialog box prompts you to locate and open another model.

Quitting Analytica

To quit Analytica, select **Exit** from the **File** menu. If you have made any changes, a Save dialog asks you whether you want to save your model before quitting.

The tool palette

When you open a model, the tool palette appears across the top of your screen. It provides a set of buttons to navigate around a model, to open different views of a model, and to change between browse and edit modes.



The first five buttons on the tool palette apply to the active Analytica window (the frontmost window).

Parent Diagram button

Opens a Diagram window for the module or model containing the active diagram window or Variable. This button is grayed out if you are looking at the Diagram window for the top-level (or root) model.

Outline button

Opens the Outline window and highlights the selected node or active module in the outline. See “The Outline window” on page 397

Object button

Opens an Object window for the selected node or active module. See “The Object window” on page 32

Result button

Opens a Result window for the selected Variable. (See “The result window” on page 45) This button is grayed out if no Variable is selected. The keyboard equivalent for the result button is Ctrl-R.

Definition button

Opens a view of the definition of the selected Variable. If the Variable is defined as a distribution or sequence, the Object Finder opens (see “Object Finder dialog box” on page 152); if it is defined as a table or probability table, its Edit Table window opens (see “Viewing an array as an Edit table” on page 209). Otherwise, an Attribute panel (see “The Attribute panel” on page 34) or an Object window (see “The Object window” on page 32) opens, depending on the Edit Attributes setting in the Preferences dialog box (see “Preferences dialog box” on page 88.) This button is grayed out if no Variable is selected. The keyboard equivalent for the

definition button is Ctrl-E.

The three tool buttons determine the mode of interaction with the model. One mode is always selected.

Browse tool button

Use to interact with the model in Browse mode. See “Browsing the window” on page 27

Edit tool button

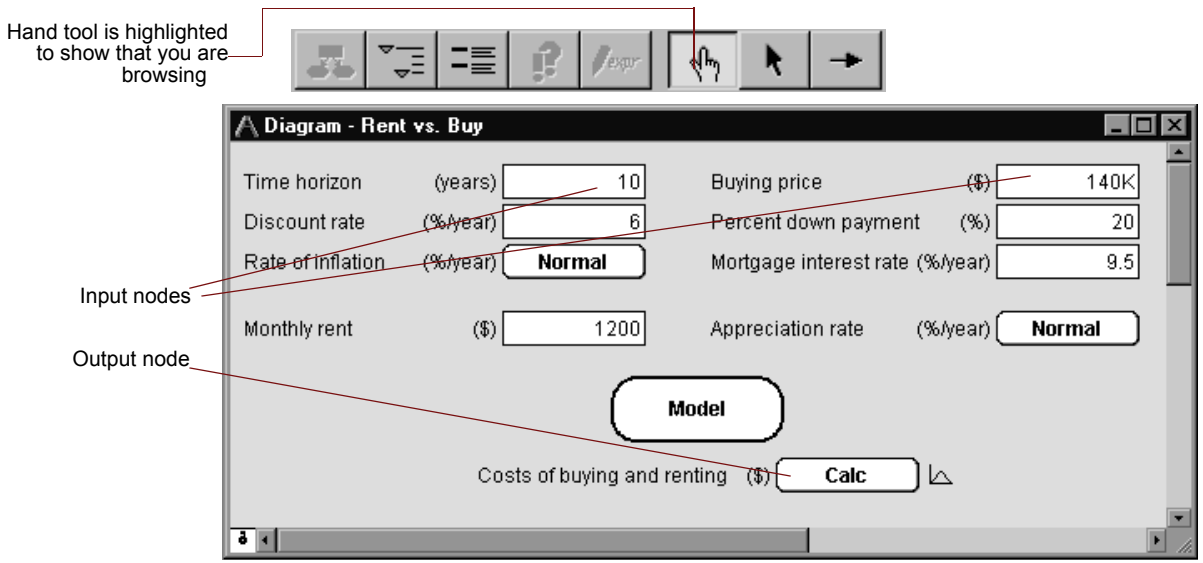
Use to interact with the model in Edit mode. See “Creating and editing nodes in a diagram” on page 73.

Arrow tool button

Use to interact with the model in Arrow mode, to add or delete arrows (dependencies) among the components of the model. See “Drawing arrows in a diagram window” on page 77

Browsing with input and output nodes

When you open a model with input and output nodes, Analytica first displays a top level Diagram window, similar to the example here.



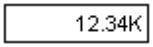
The ***input nodes*** show values that you can change to see their effects on the final values. The ***output nodes*** each show a **Calc** button. At least one node contains the details of the model.

Browsing the window

An existing model opens in Browse mode. In Browse mode, the Browse tool button is highlighted in the tool palette, and the cursor is a hand ( ).

Use the Browse tool to change input node values, view output node results, and examine the model by opening windows to see more detail.

Viewing input node values



An input field lets you see a single numeric or text value. Click in the box to change the value.

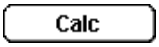
A popup menu lets you choose from a menu of alternatives. To see the alternatives, click on the popup menu. To select an alternative, click on it.

A **List** button lets you see an ordered set of values. To see the values, click on the button. To change a value, click in its cell. For more about lists, see “Editing a list” on page 221.

An **Edit Table** button lets you see a multidimensional set of values in one or more tabular spreadsheet-like windows. To see the values, click on the button. To change a value, click in its cell. For more about tables, see “Editing a table” on page 228.

A **Distribution** button lets you see a probability distribution. To see the distribution and its parameters, click on the button. For more about probability distributions, see “Probabilistic calculation” on page 290

Viewing output node values



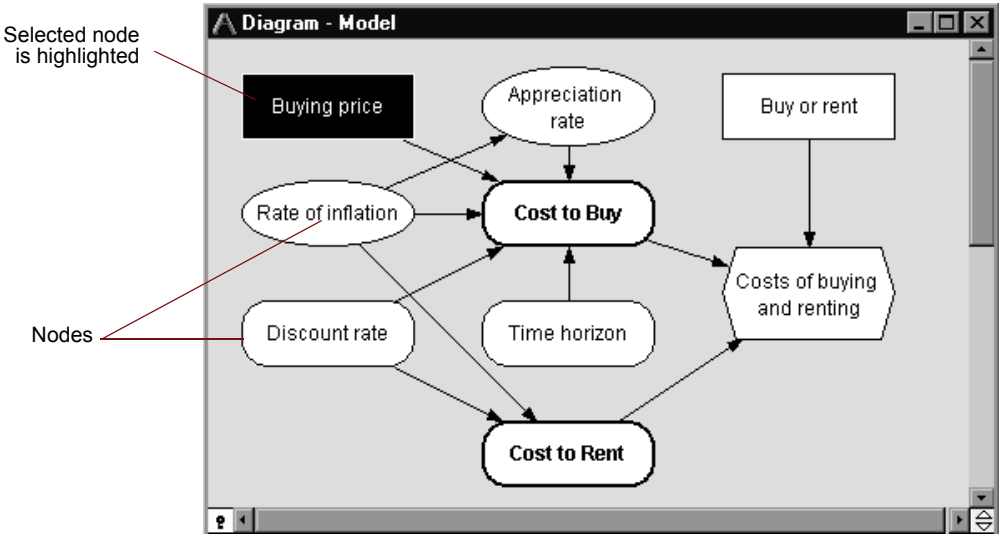
If the value of an output node has not yet been computed, the **Calc** button appears in the node. Click on the **Calc** button to compute and display the value. Chapter 2, “Viewing Results”, describes how to interpret and redisplay results.

Opening model details

To see the structure of the model, double-click on the details (rounded, thick-outline) node. A diagram window showing an Influence Diagram will display (see “Influence diagram window” on page 28).

Influence diagram window

When you open a model detail window, or a model without input and output nodes, Analytica displays a Diagram window for the model. The Diagram window depicts the model as an ***Influence Diagram***. An Influence Diagram is an intuitive graphical view of the structure of a model, consisting of nodes and arrows. Each node depicts a Variable or a module.



A ***Variable*** is any Object that has a value or can be evaluated. Nodes with thin outlines depict Variables. A ***module***, with a thick outline, contains its own Influence Diagram.

The arrows in a Diagram window depict the ***influences*** among the Variables. An influence arrow from one Variable to another means that the value of the first Variable directly affects the value (or probability distribution) of the second Variable.

For example, in the preceding diagram, the arrow from *Buying price* to *Cost to Buy* means that the price of the house affects the

overall cost of purchasing it. A higher price means a greater cost, given a fixed *Rate of inflation* and *Discount rate*. The Influence Diagram shows the essential qualitative structure of the model, unobscured by details of the numbers or mathematical formulas that may underlie that structure.

For more on using Influence Diagrams to build clear models, see Chapter 6, “Creating Lucid Influence Diagrams”.

Opening details from a diagram

To see more details of a model, double-click on nodes in the Diagram window:

- Double-click on a Variable (thin outline) node to open its Object window. See “The Object window” on page 32.
- Double-click on a module (thick outline) node to see its Diagram window, showing the next level of detail of the model.

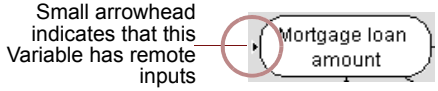
Going to the parent diagram

To see the diagram that contains the active module or Variable, click on the **Parent Diagram** button in the tool palette. The module or Variable will be highlighted in the parent diagram.

Analytica Note: *If the active diagram is of the top model, it has no parent diagram, and the Parent Diagram button is grayed out.*

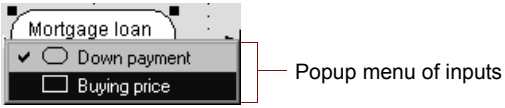
Finding remote inputs and outputs

When a Variable depends on a remote Variable—that is, a Variable in another module that is not visible—a small arrowhead appears to the left of the node. If a Variable has a remote output, a small arrowhead appears to the right of the node.



To go to the Diagram window containing a remote Variable:

1. Click on the small arrowhead. A popup menu appears listing all inputs (or outputs), including those that are not remote.



2. Click on the desired input (or output). The Diagram window containing the remote Variable opens, and the remote Variable's node is highlighted.

Viewing results

Click on a Variable node to select it; it becomes highlighted. Then click on the Result button in the tool palette to show the value of the Variable as a table or graph in a Result window. Chapter 2, "Viewing Results," discusses how to interpret and redisplay results.

Analytica Note: *If the value has not already been computed, you may need to wait while the result calculates, and the waiting cursor ( ) appears.*

Node types

Each node shape in a diagram represents a different class of objects. Here are the classes and their corresponding node shapes:

Decision

A rectangular node depicts a ***decision Variable***—that is, a quantity that the decision maker can control directly. For example, whether or not you take an umbrella to work is your decision. If you are bidding on a contract, how much you bid is your decision.

Variable

A rounded, thin-outline node depicts a ***general Variable***—that is, a quantity whose class is not determined more precisely, or a quantity that the decision maker cannot affect directly and that is not defined as probabilistic. Use a general Variable initially if you're not sure what kind of Variable you'll need, then change the node class later, if appropriate.

Chance

An oval node depicts a ***chance Variable***—that is, a Variable that is uncertain and that the decision maker cannot control directly. A chance Variable is usually defined by a probability distribution. For example, whether or not it rains is a chance Variable (unless you are a rain god). And whether or not your bid is the winning bid

is a chance Variable in your model, although it is a decision Variable for the person or organization requesting the bid.

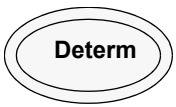
A hexagonal node depicts an ***objective Variable***—that is, a quantity that evaluates the relative desirability of possible outcomes of combinations of decision and chance Variables. Most models should contain a single objective node, although the objective can comprise several sub-objectives.

A rounded, thick-outline node depicts a ***module***—that is, a collection of nodes organized as a separate diagram. Modules can themselves contain nested modules.

A parallelogram-shaped node depicts an ***index Variable***. An index is used to define a dimension of an array. For example, *Year* is an index for an array containing the U.S. GNP for the past 20 years. Or *Nation name* is an index for an array of GNPs for a collection of nations (see “Introduction to arrays” on page 207). Index values appear in the row and column headers of a table, and in the *x* axis and key of a graph.

A trapezoid-shaped node depicts a ***constant***—that is, a Variable whose value is fixed. A constant has no inputs and is not computed. Examples of numerical constants are the atomic weight of oxygen or the number of feet in a kilometer. It is good practice to define such values as constants, so you can refer to them by name; otherwise, you must type their numerical values into each expression that includes them and search for the values when you need to change them.

A node resembling an arrow tail pointing right depicts a ***function***. You can define functions to augment the functions provided in Analytica; see Chapter 19, “Building Functions and Libraries”.



An oval node with a double rounded outline depicts a ***determ (deterministic) Variable***—that is, a Variable whose value cannot be directly controlled by the decision maker, and that is not uncertain or probabilistic. This node class is not included in the node palette to encourage use of the general Variable.

A gray rectangular node depicts a ***button Object***—that is, a graphical interface Object that triggers a script when the button is pressed. The button’s script is written in a programming language called Typescript. Typescript is a language different from the expression syntax used in definitions. Typescript is explained in the *Analytica Scripting Reference*, distributed with ADE.

Selecting nodes

To perform an operation on a diagram, you must first select a node (or a set of nodes), then select the operation to perform. There are various ways to select nodes (similar to selecting icons in the Finder):

To select one node

Click once on the node to select it. The selected node is highlighted.

You can also press the *tab* key to select one node at a time.

To select multiple nodes

Select one node, then click on another node while holding down the *shift* key. This operation adds the new node to the set of selected nodes, highlighting them all. By repeating this process, you can select as many nodes as you wish.

If the nodes are close together, you can also select them by dragging a selection rectangle around them.

To deselect one node

Click on a selected node while holding down the *shift* key. This operation removes the node from the selection, leaving the remaining nodes selected.

To deselect all nodes

Click on the background of the diagram to deselect all nodes.

The Object window

The ***Object window*** shows the attributes that together specify a node. For a Variable, as shown in the following figure, these attributes include the class, identifier (a brief, unique name), title, units, description, definition, inputs, and outputs. See the Glossary for descriptions of the attributes.

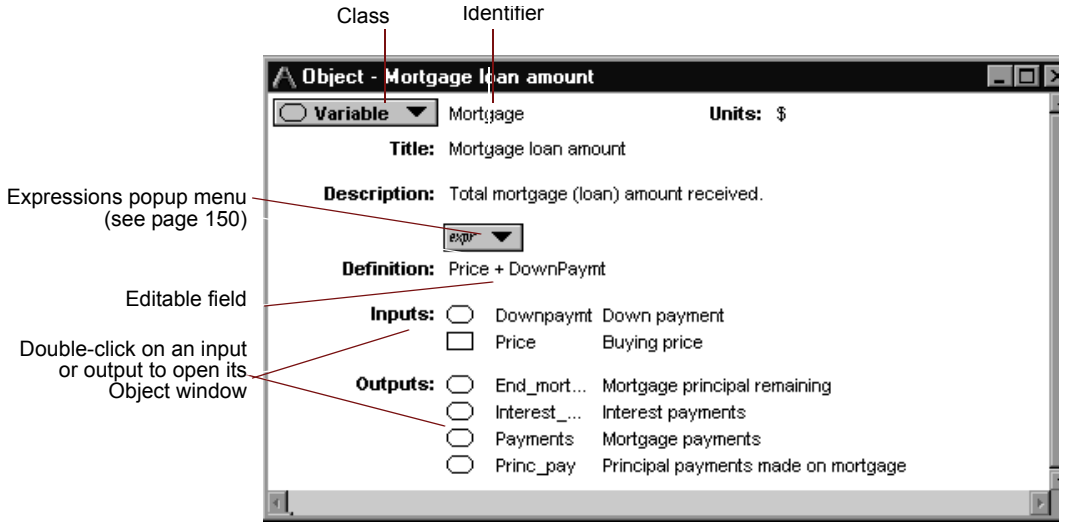
Opening an Object window

There are several ways to open the Object window for an Object titled *X:*

- Double-click on *X*'s node in its Diagram window (see page 28).
- Select *X* in its Diagram window and click on the Object button ( ) in the tool palette.
- Double-click on the entry for *X* in the Outline window (see "The Outline window" on page 397).
- If a Result window for *X* is displayed, click on the Object button ( ) in the tool palette.
- Double-click on the entry for *X* in the Inputs or Outputs field of another Variable.

Examining inputs and outputs

When you are looking at an Object window for a Variable, you can easily view the Object window for any of the Variable's inputs or outputs. Double-click on a node symbol, identifier, or title of a Variable in the list of inputs or outputs.



Returning to the parent diagram

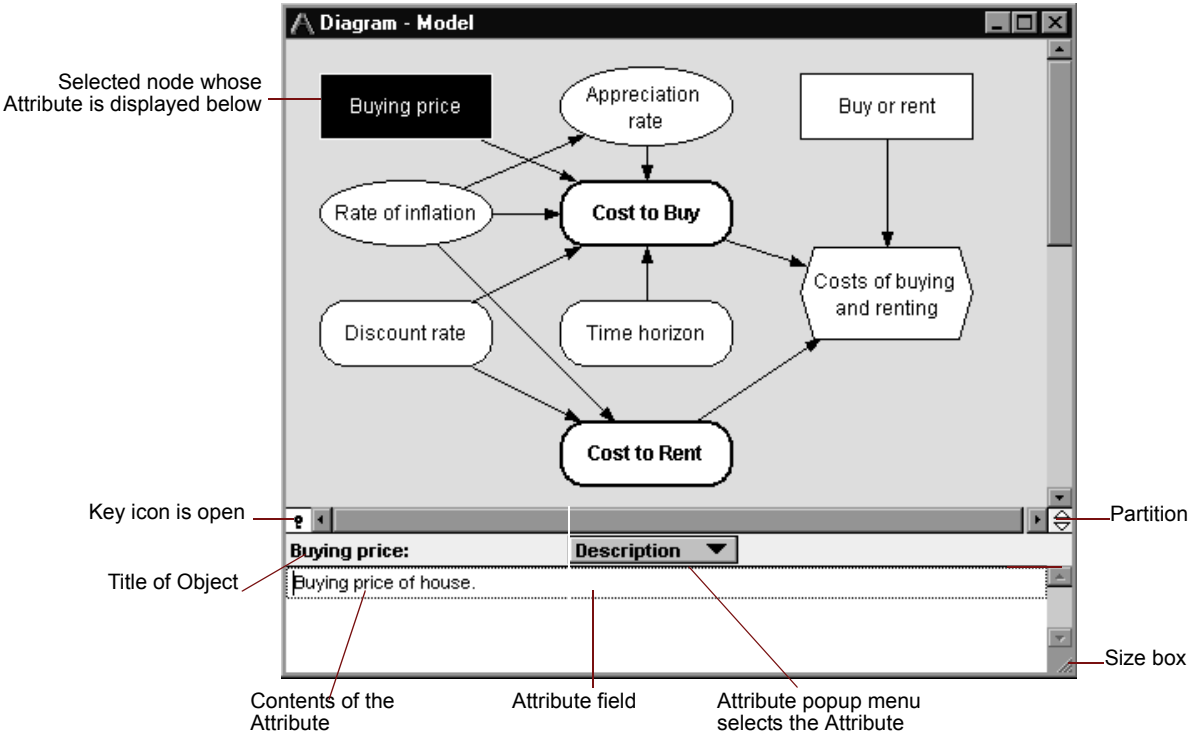
Click on the Parent Diagram button in the tool palette to see the diagram that contains this node, with the node highlighted.

Displaying additional Attributes

- To display the value of a Variable and its inputs, see “Showing mid values” on page 36.
- To display additional attributes and create new attributes, see “Managing attributes” on page 400.

The Attribute panel

The ***Attribute panel*** provides an alternative way to view attributes of a node. The ***Attribute panel*** appears as an extension below a Diagram window.



Displaying the Attribute

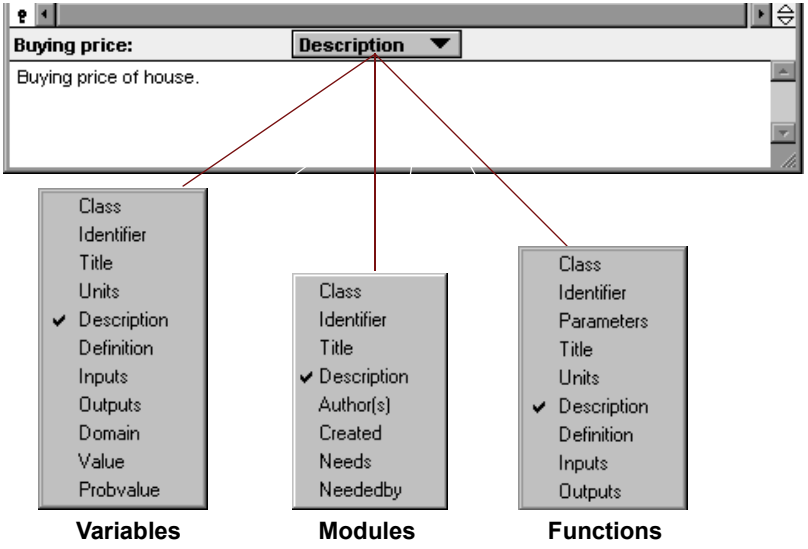
1. Click on the Key icon ( ) to display the Attribute panel.
2. Select a node to examine by clicking on it in the diagram.

Analytica Note: *If multiple nodes are selected, click on the diagram background to deselect them, then click on the node you wish to examine.*

3. To examine a different Attribute, click on the Attribute popup menu, and select the desired Attribute.
4. To examine the same Attribute for another node, select that node in the Diagram window. If no node is selected (you have clicked on the background of the diagram), the corresponding Attribute for the module is displayed.

The Attribute popup menu

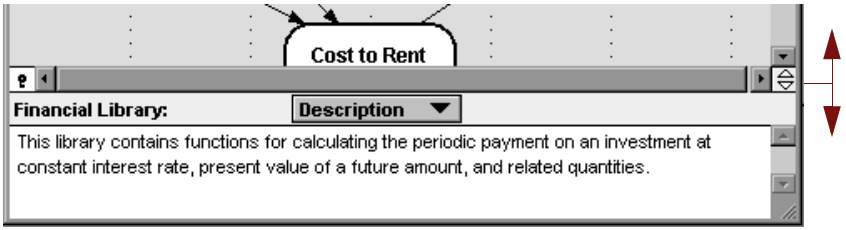
When the Attribute panel is displayed, the Attribute popup menu is located at the top center of the Attribute panel. Use this popup menu to select another Attribute to view. Variables, modules, and functions have different sets of attributes, as shown here:



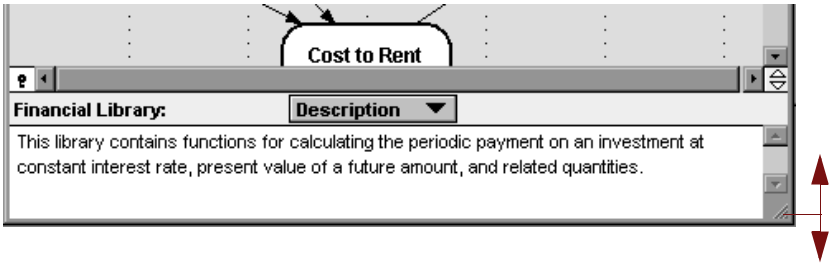
See the Glossary for descriptions of these attributes. To display other attributes or to add new ones, see “Managing attributes” on page 400.

Changing the panel size

To change the height of the diagram in relation to the diagram’s Attribute panel, drag the partition up or down.



To change the size and shape of the Attribute panel, drag the size box up or down; the height of the Diagram panel remains fixed.



To change the width and shape of the Diagram and Attribute panels, drag the size box right or left.

Closing the Attribute panel

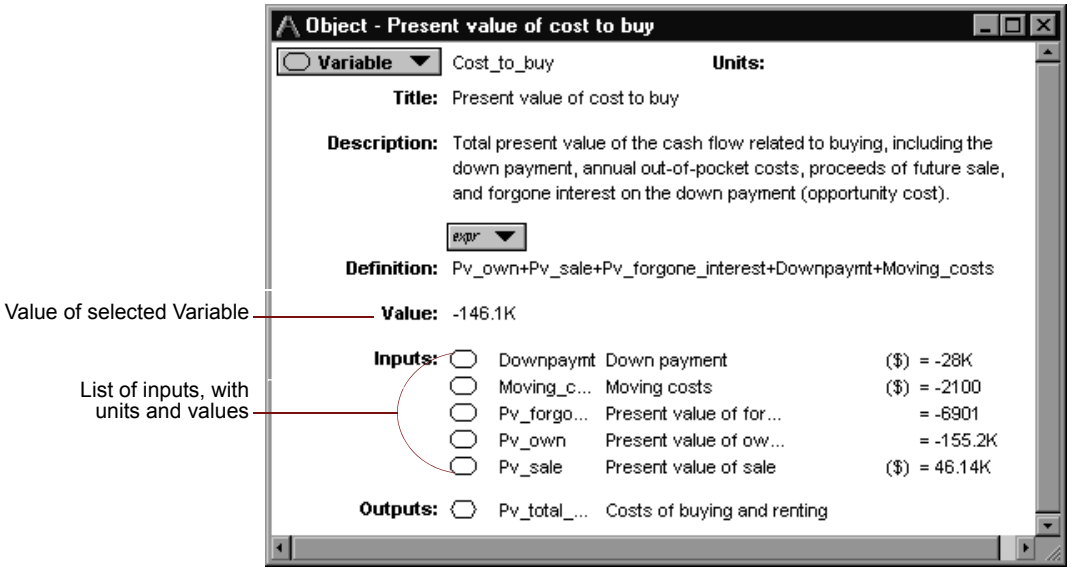
To close the Attribute panel, click on the Key icon (🔑).

Showing mid values

The ***mid value*** or deterministic result of a Variable is computed by holding each uncertain (probabilistic) input at a single, central value. The mid value for a probability distribution is its median. The mid value of a Variable is computed by using the mid value of

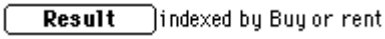
each input. If all inputs are certain (non probabilistic), the calculated value is still referred to as the mid value in Analytica.

To show the mid values of Variables in the value Attribute, select **Show with Values** from the **Object** menu. Mid values that are single values will display in Object windows and the Attribute panel. You can display these values to check that a calculation is performing correctly, or to find errors in the model.



Array values

If the mid value of the selected Variable is an array, rather than a single number, the **Result** button appears.



To display the array, click on the **Result** button. (This is equivalent to clicking on the Result button on the tool palette, described in “The tool palette” on page 24.) A Result window opens, showing the values either as a graph or as a table. For information about the Result window, see Chapter 2, “Viewing Results.”

Values of inputs

When **Show with Values** is turned on, the list of inputs displays one of the following for each input:

- The mid value, if it is a a single number (or text) and has been computed.
- **Calc** if it has not been computed. To calculate and view the value, select the input as the Variable you are examining and display its Object window or value Attribute.
- **Result** if the mid value is an array. To display it, select the input node as the Variable you are examining. The value Attribute displays the **Result** button; click on the **Result** button to see the values.

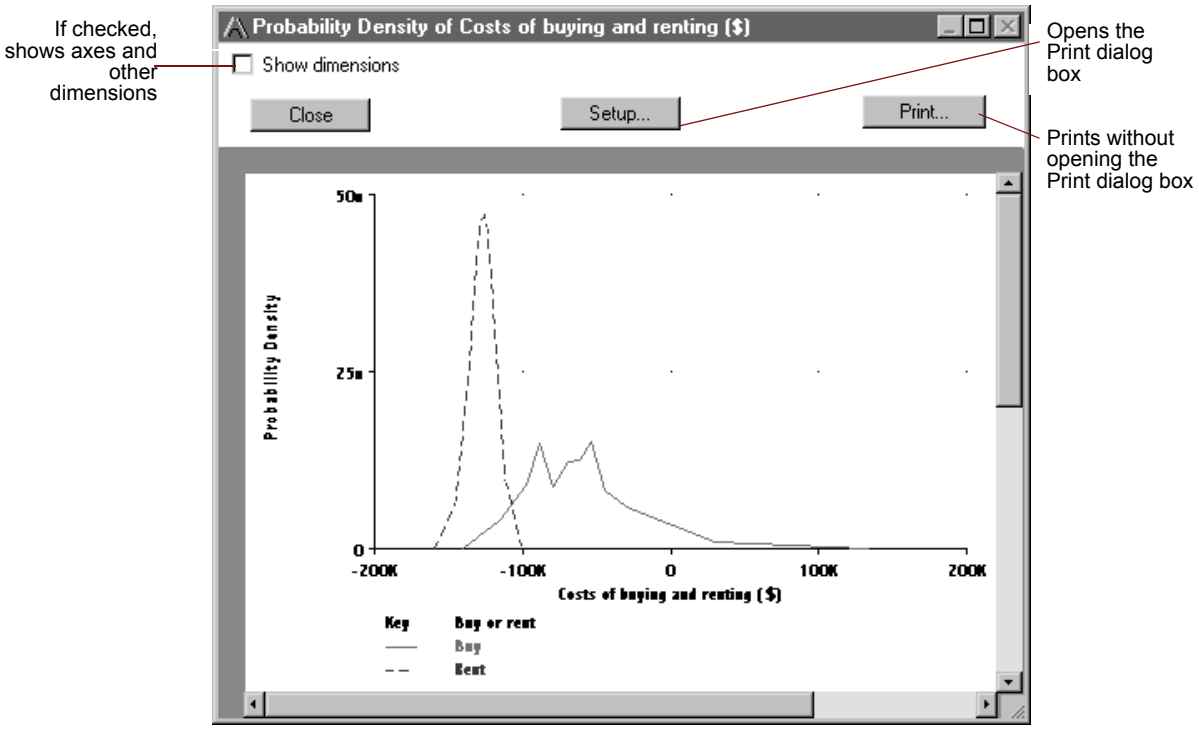
Printing

To print the contents of any window (Diagram, Outline, Object, Result Table, or aResult Graph window), activate the window and select the **Print...** command from the **File** menu. To set printing options such as page orientation, paper size, or scaling, use the **Print Setup...** command on the **File** menu. Any print settings that you specify are associated only with window that was active when you selected the **Print Setup...** command.

Previewing page breaks before printing

When you select the **Print preview** command on the **File** menu, a Preview window appears before printing. This window shows what will be printed with an indication of where page breaks will occur. Print settings such as scaling can be conveniently adjusted by selecting the **Setup...** button until the desired page breaks have been obtained. When previewing a result table or graph, an option for showing or hiding the index Variable titles can be toggled.

When viewing a diagram, outline, or Object window, page breaks can be viewed while working by enabling **Show Page Breaks** on the **Window** menu.



Scaling printouts

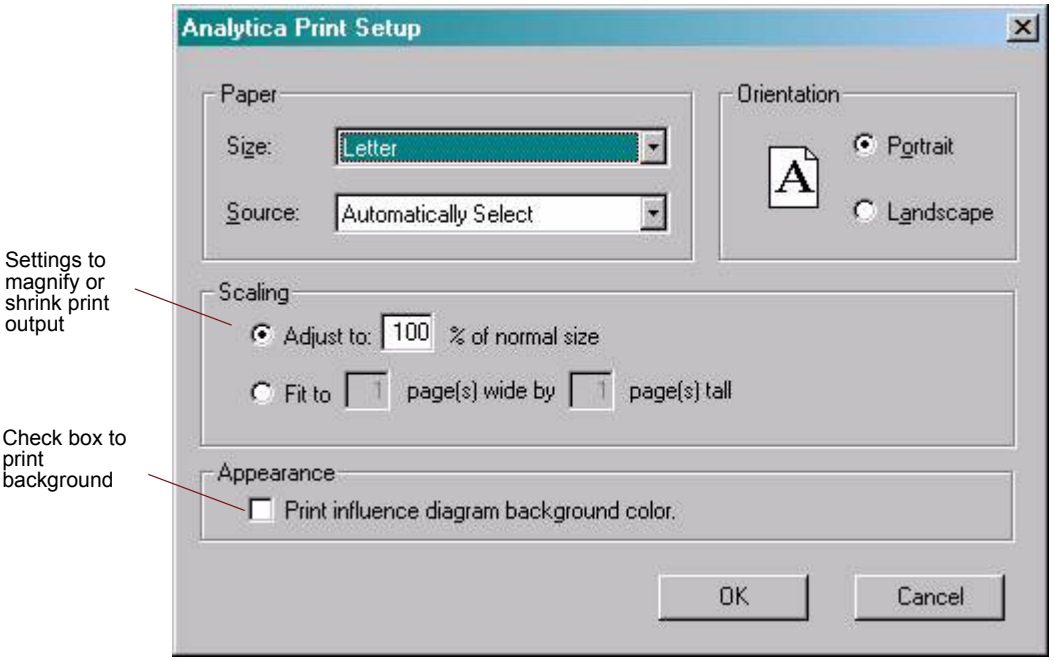
You can adjust the magnification of your printouts using the **Print Setup...** command on the **File** menu, or by using the **Setup...** button on the Print Preview window. You can specify magnification in two ways:

- Adjust to *p%* of normal size.

  p< 100% shrinks the output (fits more on a page).
  p>100% enlarges the output

- Fit to *n* page(s) wide by *m* page(s) tall.

  Shrinks the output if necessary to fit on a maximum of *n* pages wide by *m* pages tall. The output is never enlarged to fill out the specified number of pages. Also, the aspect ratio is preserved, so the actual number of pages printed may be less than *n* x *m.*

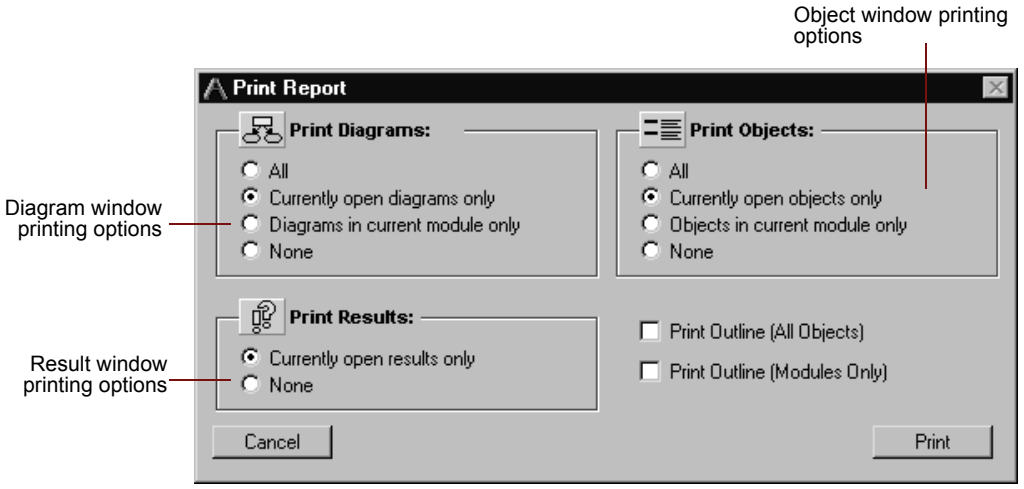


Printing the background

There is a check box on the **Print Setup...** window for controlling whether a diagram's background color is printed. By not printing the background color, one can save on ink or toner. Whether the background is printed or not is controlled by the **Print influence diagram background color** check box. By default, it does not print the background.

Printing multiple windows

To print the contents of several windows at one time, use the **Print Report** command in the **File** menu. Each window that is printed uses the print settings that have been specified for that window.



In addition to the Diagram, Result, and Object window options, there are two check boxes:

Print Outline (All Objects)

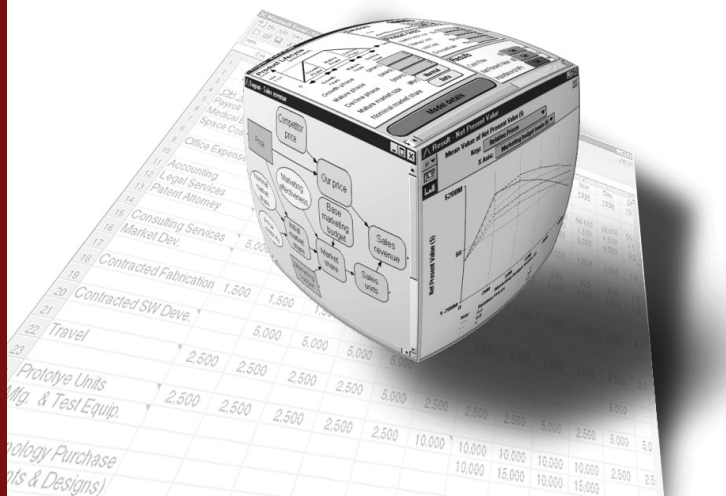
If checked, this option prints an outline of all of the objects in the model. It prints a list of all nodes by title, indented to show their position in the module hierarchy.

Print Outline (Modules Only)

If checked, this option prints the model hierarchy as an indented list containing only the modules.

Chapter 2

Viewing Results



In this Chapter

This chapter shows you how to:

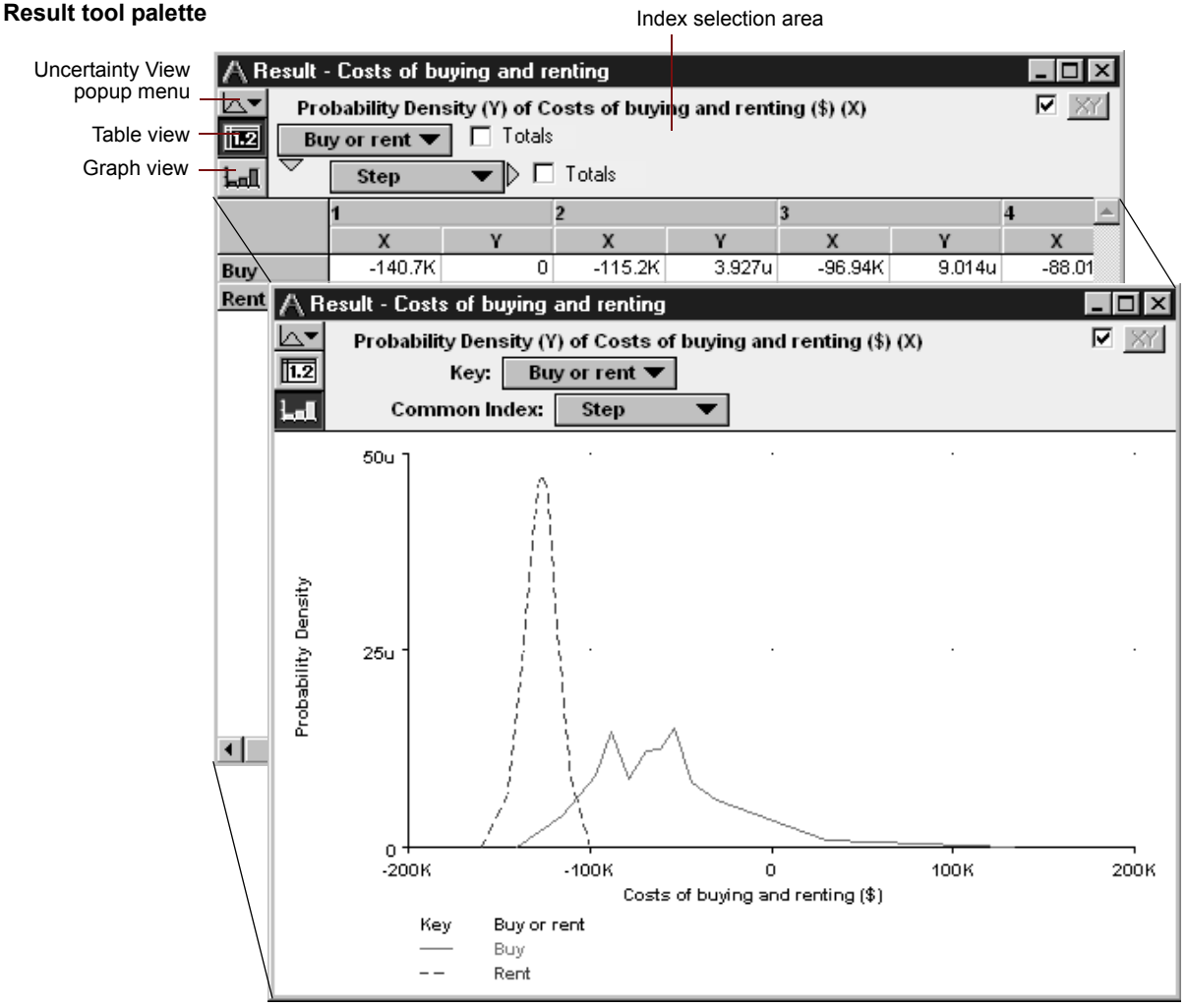
- Interpret Result windows
- View results as graphs and as tables
- View results that have more than two dimensions

2: Results

This chapter describes the elements of Result windows, how to view results as graphs and as tables, and how to view results that have more than two dimensions. It also discusses how to select a method for displaying uncertainty about probabilistic values.

The result window

Analytica computes the value of a Variable from that Variable's definition and displays the value in a Result window. If the value is probabilistic or an array, or both, it is displayed in a Result window as either a table or graph. The following figure shows a Result window of an array with the graph superimposed on a table.



Opening a result window

To open a result window for a Variable in an Influence Diagram, select the Variable and do any of the following:

- Click on the Result button ( ).
- Select **Show Result** from the **Result** menu.

- Select an uncertainty view option (such as Mid Value or Mean Value) from the **Result** menu.
- Select **Value** from the Attribute panel's popup menu.
- Select **Probvalue** from the Attribute panel's popup menu.
- Press Ctrl-R.

To open a Result window for an output node, click on the **Calc** or **Result** button.

The result tool palette

The ***Result tool palette***, in the upper left corner of the Result window, contains three items:

- The Uncertainty View popup menu. Pressing this popup menu brings up a menu of options for viewing the result data (see page 53).

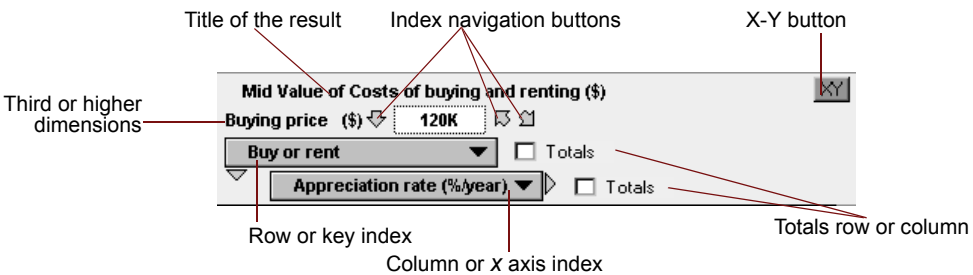
- The Table View button. Clicking this button displays the results as a table (see page 49).

- The Graph View button. Clicking this button displays the results as a graph (see page 51).

Toggle between the table and graph views using the Table View and Graph View buttons.

Index selection area

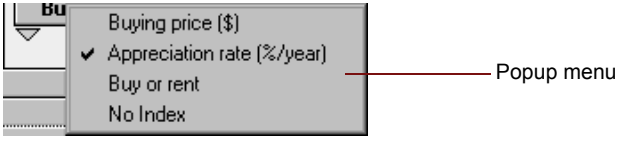
The top portion of a Result window is the ***index selection area***, which identifies the rows and columns of a table, or the *x*-axis and key of a graph. Buttons and popup menus allow you to rearrange the table or graph by interchanging indexes.



The index selection area contains these items (example Variables and indexes in the following text refer to the figure above):

- The title of the result, including the active uncertainty view, title of the Variable and units, if any; here, it is *Mid Value of Costs of buying and renting ($)*.
- Index navigation buttons, if the result has three or more dimensions (see the next page).
- The third or higher dimensions, if any (*Buying price* is a third dimension).
- The row or key index (*Buy or rent*).
- The column or *x*-axis index (*Appreciation rate (%/year)*).
- The totals check boxes. These control whether numeric row or column totals are displayed in a result table. Also when this box is checked for a given index, the index slice will default to the "Totals" option if this index is transposed to a third or higher dimension.

  Click on the popup menu (indicated by the down arrow (▼) for either index (row or column) to select a different index (or "No Index").



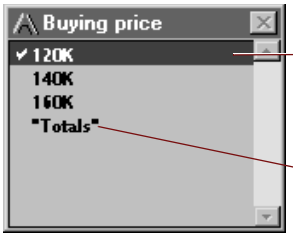
- The X-Y button (see “X-Y results” on page 355).

Index navigation buttons

When the result has three or more dimensions, not all values can display on the table or graph. For each index, or dimension, beyond the second, a set of navigation buttons appears in the index selection area.

Use the index navigation buttons to move among index values for the third (or higher) dimension of results:

- Click the left arrow ( ) to select the previous index value.
- Click the right arrow ( ) to select the next index value.
- Click the down arrow ( ) to open a dialog box showing all the values for this index.



Dialog box listing all values for the selected index (*Buying Price*)

Selected value is 120K
Click on a value to select it and to close the dialog box

Totals option. When selected, the view displays numeric totals across this index (*Buying price*).

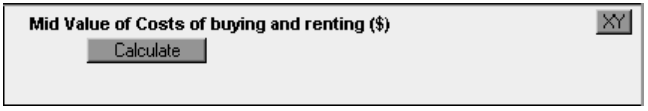
The default view

When you first display a Result window, Analytica displays the result as a graph, if possible, and otherwise as a table. You can change the default option with the "Default result view" setting in the Preferences dialog box (see “Preferences dialog box” on page 88).

When you display the Result window again, the view to be displayed is recalled from earlier in the session or from the previously saved session.

Recomputing results

When a Result window is open, and then the value of an input to the evaluated Variable is changed, a **Calculate** button appears in the index selection area.



Click on the button to recalculate the result.

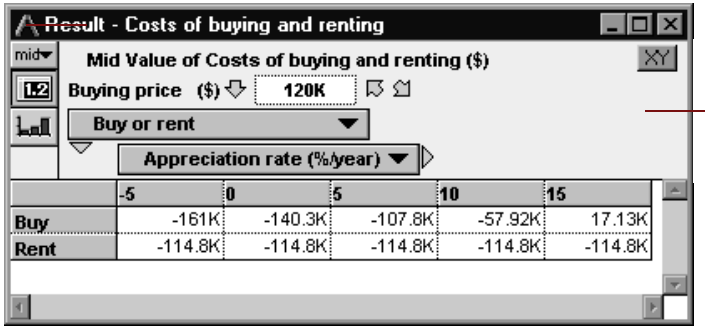
Viewing a result as a table

Displaying a table

If a graph is displayed, change it to the corresponding table by clicking on the Table View button ().

Display of values

The table view displays one or two dimensions of a Variable.



Result Tool palette (see page 47)

Index selection area (see page 47

Three-dimensional table

The display options depend on the number of dimensions in the Variable.

One dimension

The index is displayed vertically (there are no options).

Two dimensions

You can choose which index is displayed horizontally using the column index popup menu (▶), and which index is displayed vertically using the row index popup menu (▼).

Three or more dimensions

You can select a two-dimensional view using the row and column popup menus. Select specific values for the third or higher dimensions using the index navigation buttons.

Formatting numbers

You can specify the number format for a table's contents, using the Number Format dialog box.

To display the Number Format dialog box:

1. Select the row(s), column(s), or cell that you wish to format.
2. Choose **Number Format** from the **Result** menu or press Ctrl-B.

See “Number Format dialog box” on page 135 for details on the number format options.

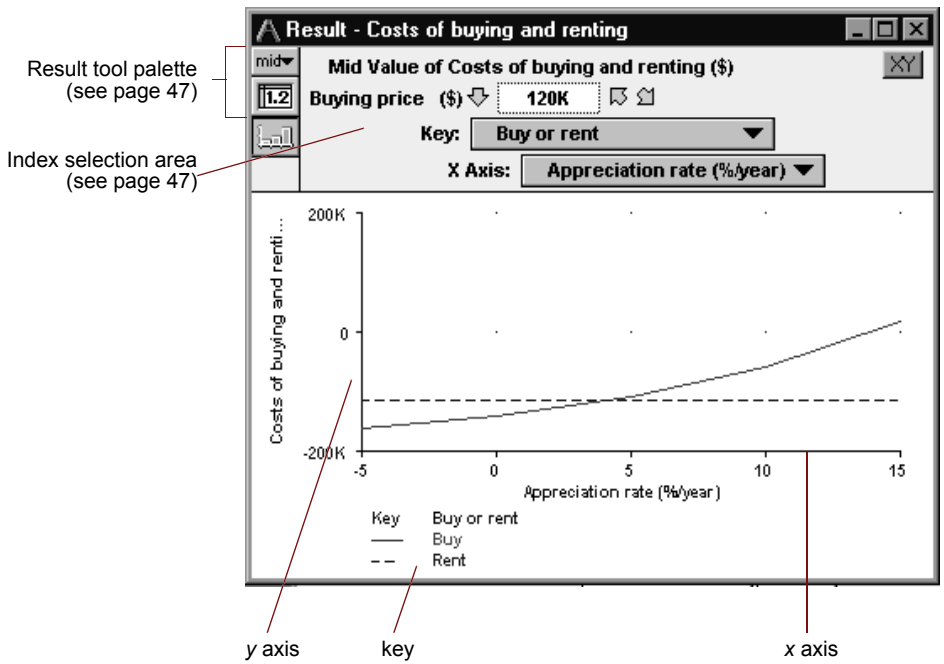
Viewing a result as a graph

Displaying a graph

If a table is displayed, change it to the corresponding graph by clicking on the Graph View button ( ).

Display of values

A ***graph*** displays the values from an array.



The vertical *y* axis shows the Variable’s values.

The display options depend on the number of dimensions in the Variable.

One dimension

The values of the dimension are shown horizontally, along

the *x* axis (there are no options).

Two or more dimensions

You can choose which index is displayed along the *x* axis by using the *x* axis popup menu, and which index produces multiple curves by using the key index popup menu.

The ***key*** shows the value of the index Variable that corresponds to each curve, indicated by pattern or color.

Select specific values for the third or higher dimensions using the index navigation buttons.

Changing graph ranges and styles

You can override the default ranges and styles for a graph, or you can change the default for all graphs, using the Graph Setup dialog box.

To display the Graph Setup dialog box, do one of the following:

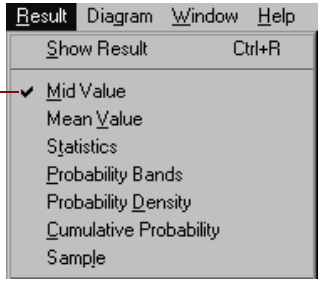
- Select **Graph Setup** from the **Result** menu.
- Double-click anywhere on a graph in the Result window.

See “Graph Setup dialog box” on page 129 for details on the Graph Setup options.

Uncertainty view options

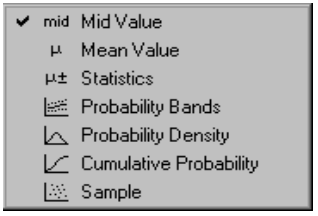
The value of a Variable in an Analytica model can be either ***certain*** (deterministic) or ***uncertain*** (probabilistic). An uncertain value can be viewed in several different ways. Select the uncertainty view in either:

- The **Result** menu.
- The Uncertainty View popup menu (top left corner of the Result window).



Currently selected uncertainty view option

Result menu uncertainty view options



Uncertainty View popup menu

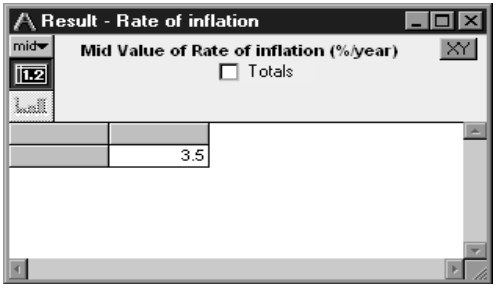
The check mark indicates the currently selected uncertainty view option. If the active window is not a Result window, selecting an uncertainty view option opens a Result window for the selected Variable.

The uncertainty view options are described briefly here. For more information on these options, see their entries in the Glossary and consult any standard statistics textbook.

The following examples use the Variable *Typical Uncertainty*, which is defined as a normal distribution having a mean of 50 and a standard deviation of 30.

Mid Value

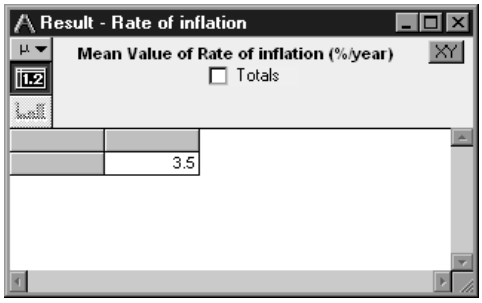
The mid value is the deterministic value, computed by holding probability distributions at their median values. This value is computed very quickly compared to the uncertainty values. The mid value is the only option available for a certain (non probabilistic) Variable.



Mean Value

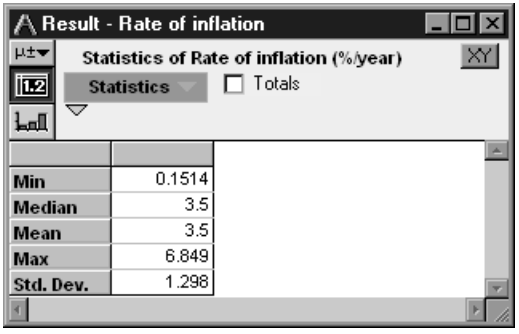
The mean value is an estimate of the expected value of the uncertain value. For a symmetrical distribution, such as a

normal distribution, the mean is the same as the median (mid) value.



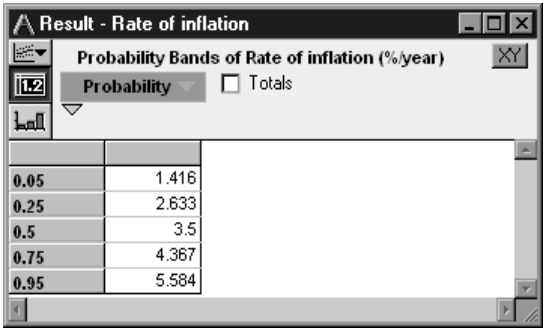
Statistics

Statistics for the uncertain value, such as mean and standard deviation, are provided in a table. Select the statistics to be calculated using the Uncertainty Setup dialog (see “Statistics option” on page 296).



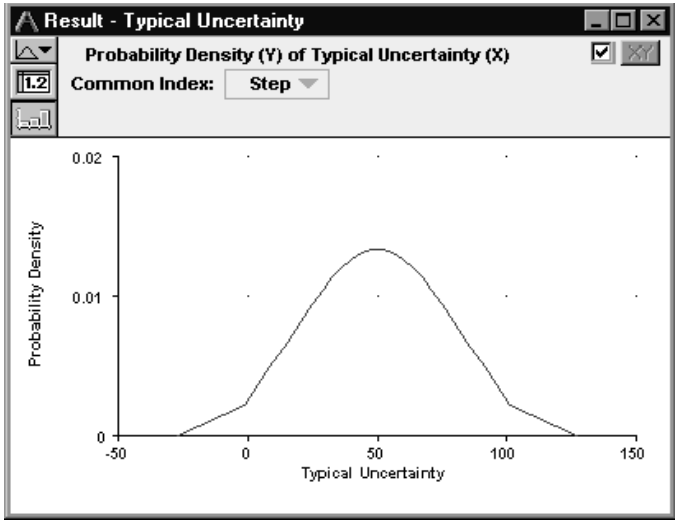
Probability Bands

Probability bands are specified at given percentile values. Select the probability bands you wish to show using the Uncertainty Setup dialog (see “Probability Bands option” on page 296).



Probability Density or Probability Mass

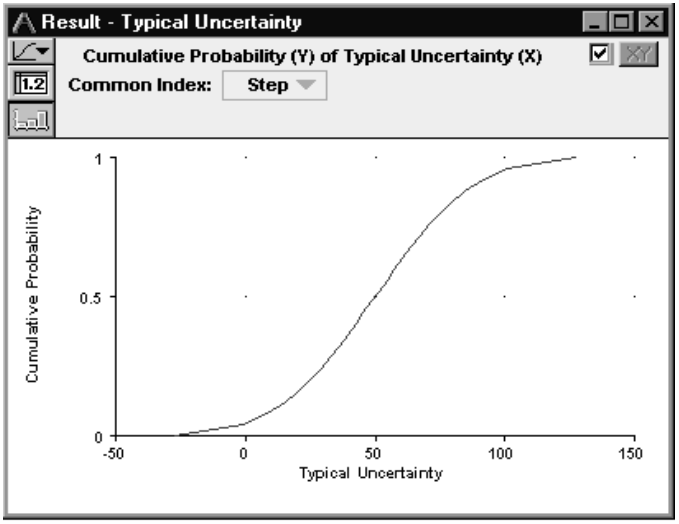
If the quantity is a continuous probability distribution, Analytica displays a probability density function. The horizontal (*x*) axis plots possible values of the uncertain quantity. The height of the curve (probability density) is proportional to the likelihood that the quantity will have the *x* value. The highest point on the curve is the most likely value (the mode). Where the curve is at zero height or invisible, there is zero probability that the quantity will have that value. (For a discrete probability distribution, Analytica graphs the probability mass).



The "common index" of Step is a counter to relate the Variable's values to the probability density; at the first and last value of Step, the probability density is zero.

Cumulative probability

The cumulative probability distribution plots the possible values of the uncertainty quantity along the horizontal (*x*) axis. The height of the graph at each value of *x* shows the probability that the quantity will be less than or equal to that *x* value. The cumulative distribution ranges from a probability of 0 on the left to probability of 1 on the right, without decreasing. The steeper the curve, the more likely the quantity will have a value in that region.

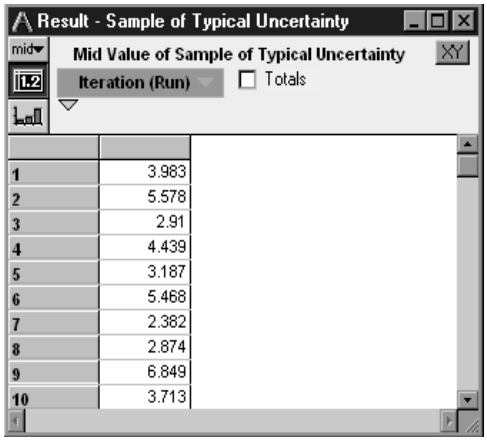


The "common index" of Step is a counter to relate the Variable's values to the cumulative probability; at the first value of Step, the cumulative probability is zero and at the last value of Step, the cumulative probability is 1.

Sample

A sample is an array of the random sample of values generated by the sampling process. The sample is the underlying representation for an uncertain quantity.

All other representations of uncertainty are estimated from the sample. The precision of the estimates depends on the sample size and the sampling method (see "Selecting the sample size" on page 499 for selecting the sample size and "Uncertainty Setup dialog box" on page 291 for setting the sample size).

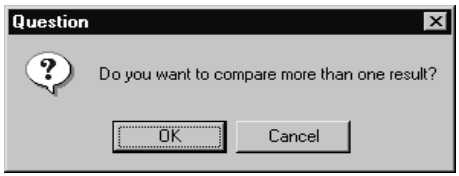


Analytica Note: *Continuous distributions, such as Normal(0,1), have real-valued samples and are described by probability density functions (PDFs) and cumulative density functions (CDFs). Discrete distributions (e.g., ChanceDist) have a finite set of possible values for each sample point and are described by their probability mass and cumulative mass. There can be an ambiguity when the samples from a distribution are numeric; in that situation, it is not always clear whether the distribution is discrete or continuous, and thus whether the result should be graphed as a probability distribution function or a probability mass.*

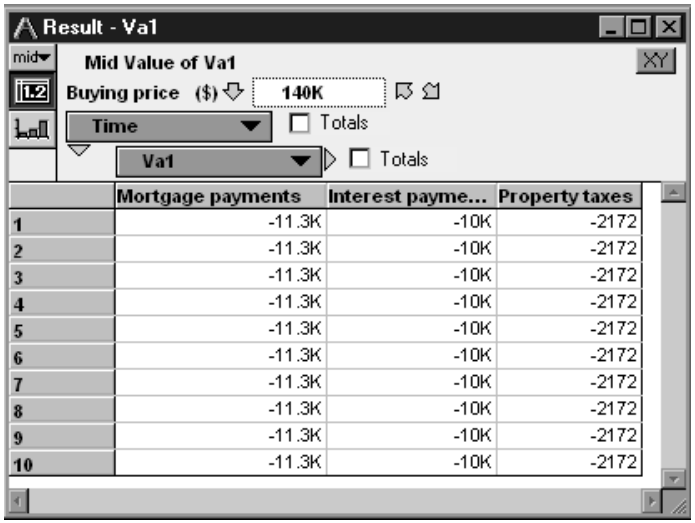
Analytica 3.1 makes more intelligent guesses about whether a sample is from a discrete or continuous distribution. So, for example, it will usually correctly induce that the samples from a Poisson distribution are discrete, even though they are numeric. When displaying a probability mass or cumulative mass, the solid bar graph style is used, and when displaying a continuous density (PDF or CDF), the histograph style is used. If Analytica guesses incorrectly, you can override this by selecting solid bar (if you want to view discrete mass) or histogram (if you want continuous density) graph style.

Comparing results

To directly compare the values of two or more Variables in one table or graph, select all the Variables together and open a Result window (see page 46). A dialog box asks for confirmation.



Analytica creates a new node with a default title and displays the values in one table or graph.

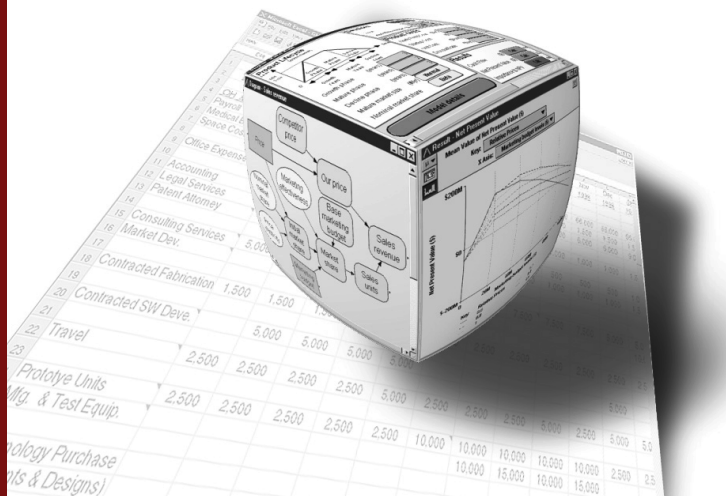


	Mortgage payments	Interest payme...	Property taxes
1	-11.3K	-10K	-2172
2	-11.3K	-10K	-2172
3	-11.3K	-10K	-2172
4	-11.3K	-10K	-2172
5	-11.3K	-10K	-2172
6	-11.3K	-10K	-2172
7	-11.3K	-10K	-2172
8	-11.3K	-10K	-2172
9	-11.3K	-10K	-2172
10	-11.3K	-10K	-2172

In this example, we have compared the changing values of three Variables—*Mortgage payments*, *Interest payments*, and *Property taxes*—over ten time periods.

Chapter 3

Analyzing Model Behavior



In this Chapter

This chapter shows you how to perform a parametric analysis on a model by

- Selecting Variables as parameters
- Specifying alternative values for the parameters
- Examining the results

3: Analyzing model behavior

A potent source of insight into a model is examining the behavior of its outputs as you systematically vary one or more of its inputs. This technique is called ***model behavior analysis***. Each input that you vary systematically is called a ***parameter,*** and so this technique is also known as ***parametric analysis***. Since you can view this as exploring hypothetical scenarios, it is also called ***scenario*** or ***what-if analysis.*** Analytica makes it simple to analyze model behavior in this way. All you have to do is to assign a list of alternative values to each input parameter. When you view the result of any output, Analytica computes and displays a table or graph showing how the output values vary for all combinations of the values for each input.

This chapter describes how to select Variables as parameters, how to specify alternative values for the parameters, and how to examine the results.

Varying input parameters

The first step in analyzing model behavior is to select one or more input Variables as parameters and to assign each parameter a list of possible values.

Which inputs to vary

You can vary any numerical input Variable of your model, including decisions and chance Variables. Often you will want to vary each decision Variable to see which value gives the best results according to the objectives. You may also want to vary some chance Variables to see how they affect the results. It is often best to look first at the decision or chance Variables that you expect to have the largest effect on the model outputs. In complicated models, you may want to start with an importance analysis, to identify which chance Variables are likely to be most important. (see Chapter 15, “Sensitivity and Uncertainty Analysis”). You can then select the most important Variables as the parameters to vary to analyze model behavior.

How many values to assign

Usually it is best to assign a list of three alternative values to each parameter—a low, medium, and high value. In some cases, two values may be sufficient. If you have a special interest in a particular parameter (for example, if you suspect it may have a strongly nonlinear effect) you may want to assign more than three values to examine in more detail the model behavior as the parameter varies. Naturally, the computation time increases with the number of values.

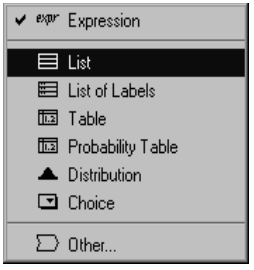
Creating a list

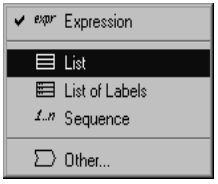
To create a list of values for a Variable, change its definition following these steps:

1. Select the Variable by clicking on its node in the Influence Diagram.

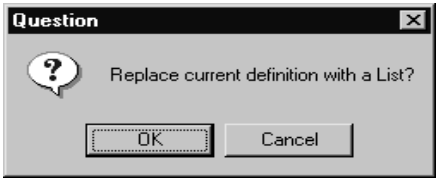
2. Display the Variable's definition by clicking on the Definition button in the tools palette.

3. Click on the Expressions popup menu above the definition and select the **List** option. (Do *not* select the **List of Labels** option.)

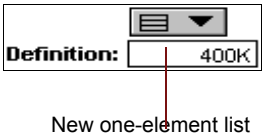




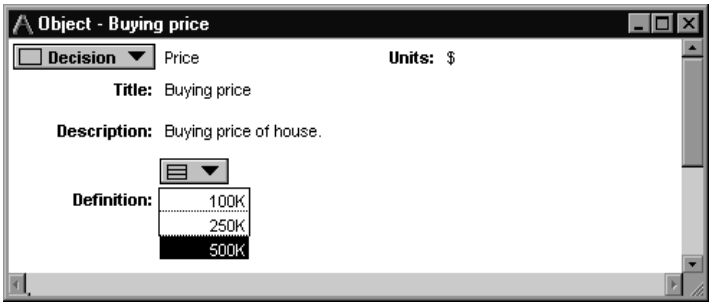
4. A dialog box asks for confirmation. Click **OK**.



Analytica displays a list with one element, containing the old definition of the Variable.



5. Select the element by clicking on it.
6. Type in the lowest value for the Variable.
7. Press *Enter* and type in the next value.
8. Repeat step 7 until you have all the values you want.



Analytica Note: *After you have entered two or more values into a list, the next item receives a default value. For example, if the last two values are 10 and 20, Analytica offers 30 as the next value.*

For details on how to edit a list, see “Editing a list” on page 221.

If you want to create a list with a large number of evenly spaced values, use the `Sequence()` function (see page 223).

How many inputs to vary

Typically you should start a model behavior analysis by varying just one input Variable, the one you expect to be most important. Vary additional Variables one at a time, in order of their expected importance. If a Variable turns out to have little effect, you may restore it to its original value or probability distribution. If you have many inputs whose effects on model behavior you would like to explore, vary just a few at a time, rather than trying to vary them all simultaneously.

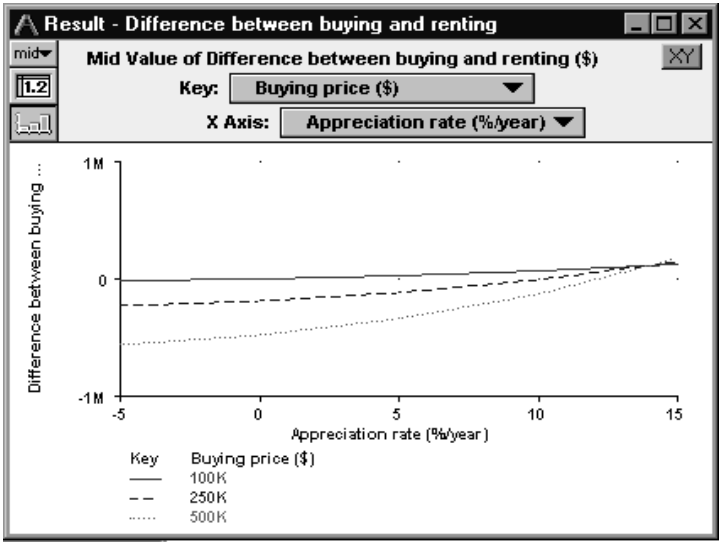
Each parameter that you vary becomes a new dimension of your output result array. The computation time and memory needed increase roughly exponentially as you add parameters. Moreover, you may find it hard to interpret an array with more than three or four dimensions. Remember that the goal is to obtain insight into what affects the model behavior and how.

Analyzing model behavior results

Once you have assigned a list to one or more inputs, you can examine their effect by viewing the result on an output Variable. If your model has an objective, you might start by looking at that Variable.

1. Select the Variable you wish to view by clicking on its node in the diagram.

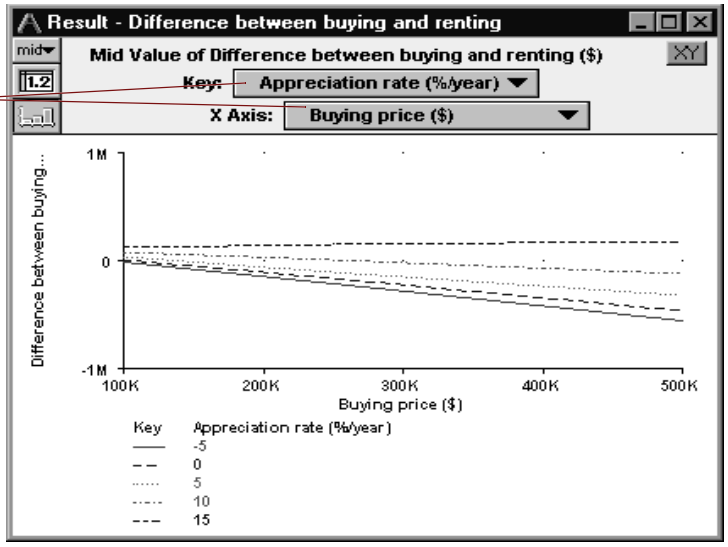
2. View the result by clicking on the result button in the tool palette. The result displays as a table or graph.



The result is an array with a dimension for each input parameter that you have varied (in this example, *Buying price* and *Appreciation rate*). If an input parameter does not appear as a dimension of the result, it implies that the result Variable does not depend on the input. The result may also have other dimensions that are not input parameters you have varied—for example, *Time* for a dynamic model.

It is generally easiest to look first at the result graph to see the model's general behavior. You need to look only at the result table if you want to see the precise numerical values. If you are varying more than one input parameter, try rearranging the dimensions (see "Index selection area" on page 47) to get additional insights into model behavior.

Result graph with dimensions reversed



Understanding unexpected behavior

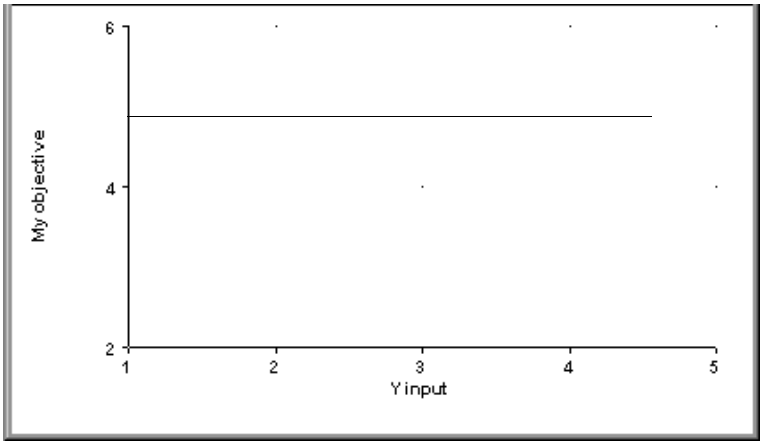
If you find the model's behavior unexpected or inexplicable, you may want to look more deeply into how the behavior arises. An easy way to do this is simply to look at the results for other Variables between the input(s) and the output(s) in which you're interested. You can work forwards from an input towards the output, or backwards from the output towards the inputs. Look at the behavior of each intermediate Variable, and see if you can understand why the inputs affect it the way they do.

Typically, the reason for unexpected behavior will quickly become clear to you. It may be that some intermediate relationship has an effect different from what you expected. It may turn out that there is an error in a definition. In either case, this kind of exploration can be very revealing about the model. You may end up improving the model or gaining a deeper understanding of the system it represents.

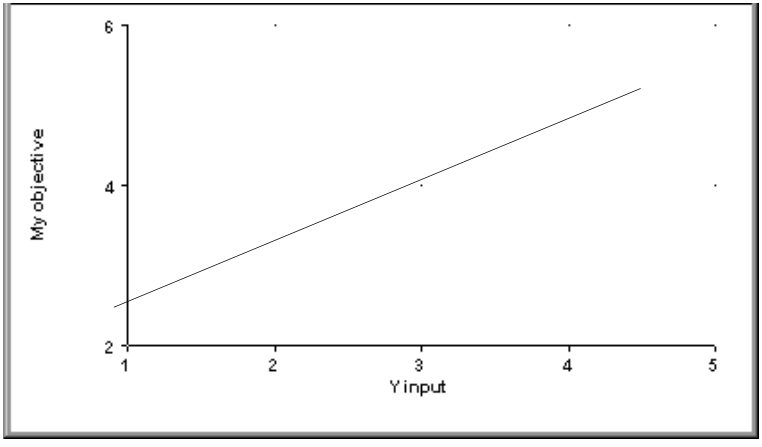
Understanding model behavior

By examining result graphs, you can learn if each input affects the output, if the effect is linear or non-linear, and if there are interactions among inputs in their effect on the output. Below are some typical graph patterns and their qualitative interpretations.

- A horizontal line shows that changes in the input over the specified range have no effect on the output.

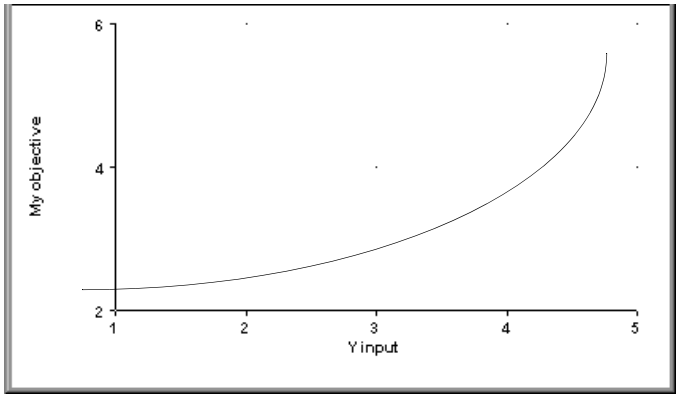


- A straight line shows that the output depends linearly on the input—provided that you have specified more than two different values for the input.



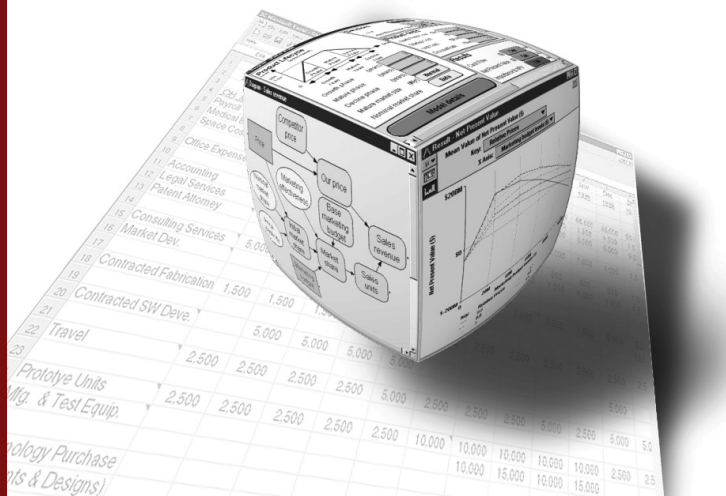
- A bent or curved line shows that there is a nonlinear dependence. (If you have only two values for the input, the

graph will be a straight line even if there is a nonlinear dependence.)



Chapter 4

Creating and Editing a Model



In this Chapter

This chapter shows you how to:

- Create a new model
- Save changes
- Create and edit nodes
- Draw arrows
- Make aliases

4: Creating and editing a model

This chapter introduces you to the elements for building Influence Diagrams. It describes how to create a new model and save changes, how to create and edit nodes, and how to draw arrows and make aliases.

Creating and saving a model

You can create new models in Analytica, as well as edit your models and save the changes that you make.

Creating a new model

To create a model, do one of the following:

- Start Analytica. The program begins with a new, untitled model open.
- If Analytica is already running, click on **File** and select **New Model** in the pull-down menu.

  If an existing model is currently open, Analytica does one of the following:

  - If the model is unchanged, Analytica closes it.
  - If the model has been changed, Analytica displays a dialog box that allows you to
    - save the model before closing it
    - close it without saving
    - or cancel the action.

Saving a model

To save changes to the model, select the **Save** command from the **File** menu (Ctrl-S).

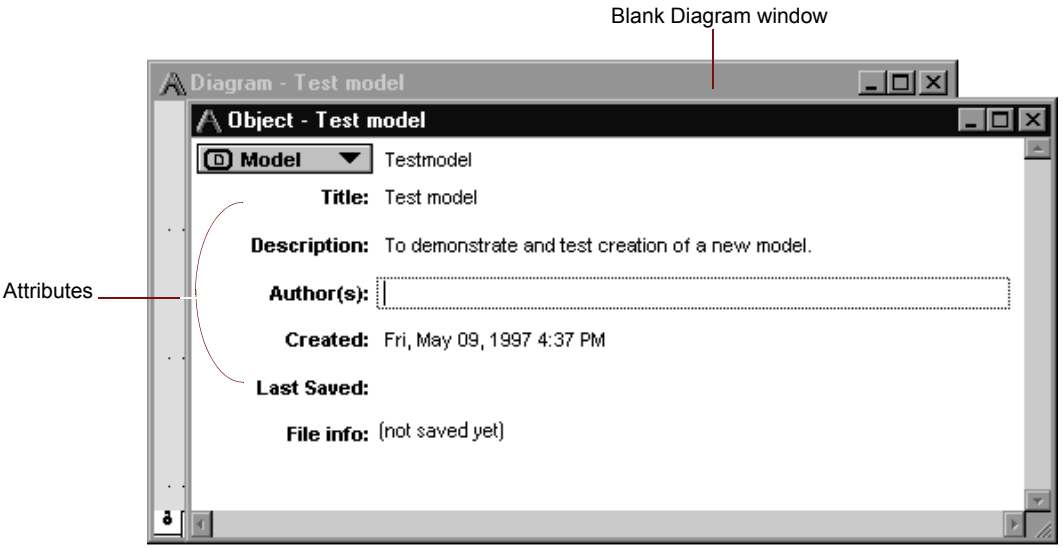
To save the model file under a new name, select the **Save As** or **Save a Copy In** command from the **File** menu. After you use the **Save As** command, selecting the **Save** command will save the model with the new name. After you use the **Save a Copy In**

command, selecting the **Save** command will save the model with its original name.

The model Object window

The model Object window shows information about the model, such as the author(s), and creation and save dates; it also includes space for a description of the model's purpose.

When you create a model, an Object window is displayed for the new model, initially untitled, with the fields shown in the following figure. Enter information as appropriate.



See the Glossary for descriptions of the Attributes.

After entering information into the model Object window, bring the Diagram window to the top in any of three ways:

- Click on the Parent Diagram button.
- Click anywhere in the Diagram window behind the Object window.
- Click on the Object window's Close box.

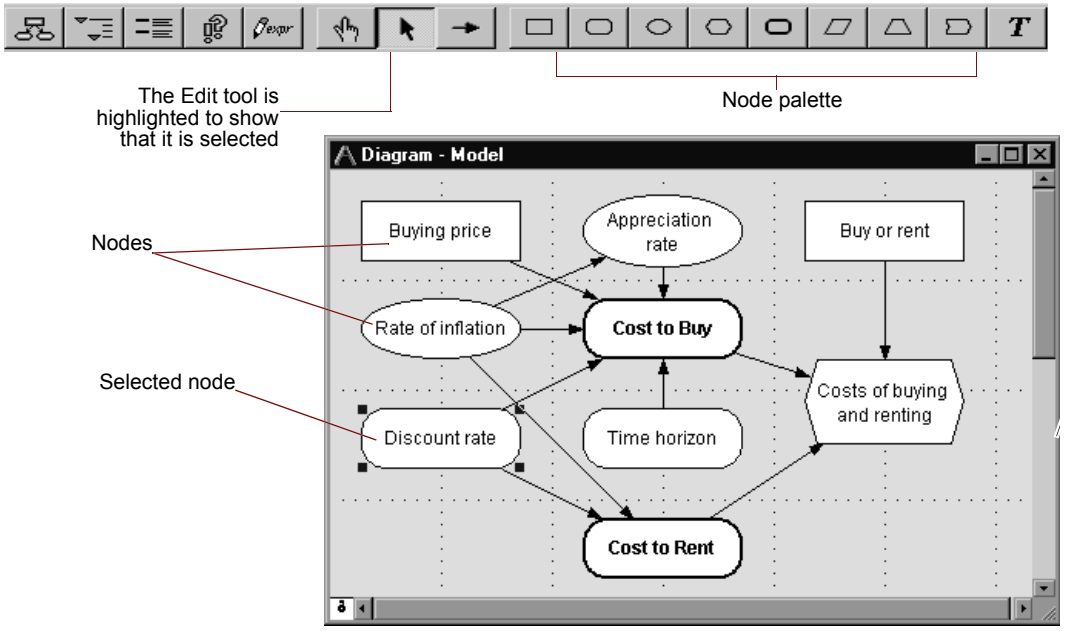
You can now draw a diagram for the new model (see "Creating and editing nodes in a diagram" on page 73).

Creating and editing nodes in a diagram

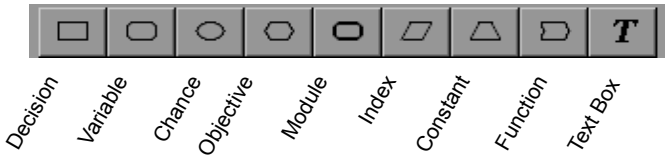
To create new nodes, or move or modify existing nodes, the Edit tool must be selected.

When a Diagram window for a new model is first opened, the Edit tool is selected by default. When a Diagram window for an existing model is first opened, the Browse tool is selected (see “Browsing the window” on page 27), so you can examine, but not change, the diagram.

To begin editing a diagram, click on the Edit tool ( ), if it is not selected.



When you are in Edit mode or Arrow mode, the ***node palette*** appears at the right of the tool palette.



The node palette is displayed when either the Edit tool or Arrow tool is selected.

For details about the node classes in the node palette, see “Node types” on page 30.

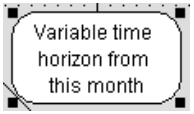
Creating a node

To create a new node, press on the appropriate icon in the node palette, then drag the outline into the diagram. After placing the node in the diagram, use the keyboard to enter its title.

Editing a node title

To edit the title of a node, first select the node, then click on that node’s text field. Pause between the mouse click to select the node and the mouse click to select the text; otherwise, your action may be interpreted as a double-click, opening the node’s Object window. The node’s appearance changes to match the illustration of the node on the left, below. When you have finished typing, press *Alt-Enter* (or *Enter* on the numeric keypad) to accept the title change and resize the node to fit the text.

You can edit the title when the node looks like this

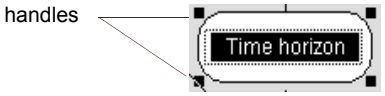


The node is resized to fit the text

When a node’s title is first created, its ***identifier*** is created from the first 20 characters of the title (underscores replace spaces). The node’s identifier is the name used to refer to this node in the mathematical definition of other nodes. The node keeps this identifier until it is explicitly edited, unless the Change Identifier preference is set (see page 90).

Selecting nodes

To select a node, single-click on it. Handles indicate that you have selected the node. To deselect a selected node, click anywhere outside of it.



To select or deselect multiple nodes, Shift-click. You can also select a group of nodes by dragging a rectangle around them. Move the cursor to a corner of the diagram (not in a node), press the mouse button, and drag the mouse to draw a rectangle. When you release the button, all the nodes completely inside the rectangle are selected.

Working with nodes

You can manipulate the nodes in a diagram in a variety of ways.

Moving a node

To move a node, press the mouse while the cursor is inside the node but not on a handle, then drag the node.

You can also move a selected node using the arrow keys (up, down, left, right).

Moving a node into a module

To move a node into a module in the diagram, drag the node onto the module until the module becomes highlighted. When you release the mouse button, the node goes into the module.

Alternatively, double-click on the module to open its diagram window. Move the module diagram window so both it and the node to be moved are visible. Then drag the node onto the module diagram window.

Changing the size of a node

To change the size of a node, drag a handle until the node is the size you desire.

By default, a node is resized keeping the center in place (that is, all four corners expand or contract). This helps to keep nodes on the grid and lined up with each other. To turn off the default so one corner at a time can be resized, uncheck the **Resize Centered** option in the **Diagram** menu.

Deleting a node

To delete a node, first select it. Then, choose **Clear** from the **Edit** menu, or press the *Delete* key. You will be prompted to confirm your intentions before the node is deleted.

Copying and pasting nodes

To create nodes that have substantial information in common, you can use the standard Copy and Paste commands. Initially, the copies are identical except for their identifiers (which have numbers appended to them to make them unique).

Cutting a node	Select **Cut** from the **Edit** menu(Ctrl-X).
Copying a node	Select **Copy** from the **Edit** menu (Ctrl-C).
Pasting a node	Select **Paste** from the **Edit** menu (Ctrl-V).

Duplicating nodes

To create two sets of nodes that have substantial information in common, create the first set. Select the nodes, then choose **Duplicate Nodes** from the **Edit** menu. This is equivalent to using Copy and Paste, without writing to the clipboard.

Aligning to the grid

When the grid is on (the default), each node that you create or move is centered on a grid point. This default makes it easier for you to position nodes so that arrows are exactly horizontal or vertical when nodes are aligned vertically or horizontally.

To re-center nodes, select **Align Selection to Grid** from the **Diagram** menu (Ctrl-J).

To turn the grid off in edit mode, uncheck **Snap to Grid** from the **Diagram** menu. When the grid is off in edit mode, the grid is still visible; you can move the nodes pixel by pixel.

Adjusting node Z-order

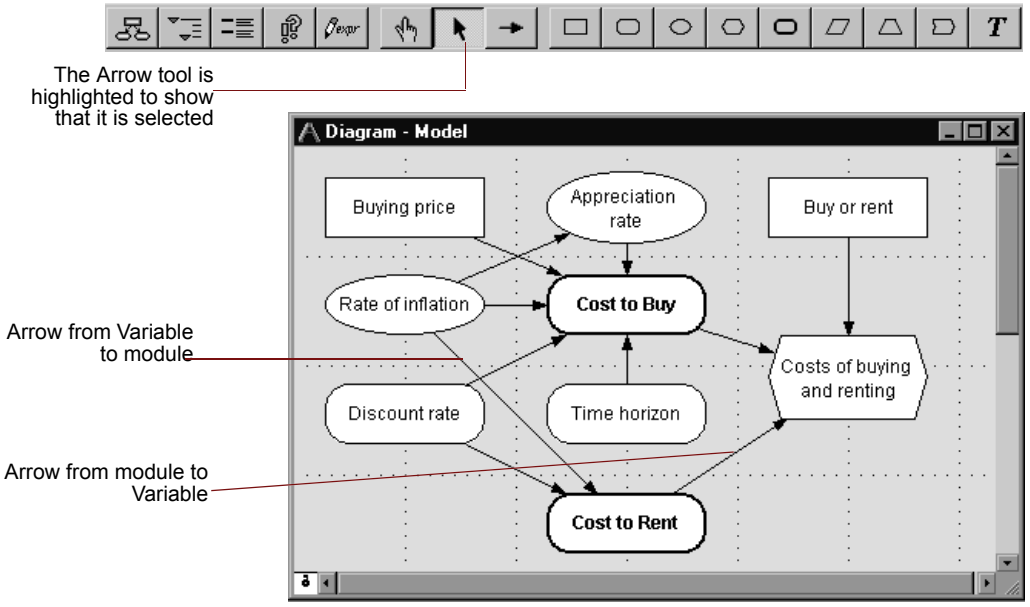
The node Z-order specifies what diagram elements will display on top of others if the items overlap. By defaults, text and picture nodes are behind arrows, and arrows are behind nodes. The Z-order can be changed by selecting a node or nodes and using the **Send to Back** and **Bring to Front** commands, which are found only on the right mouse button menu.

Drawing arrows in a diagram window

Use the Arrow tool to draw or remove arrows (influences) between Variable nodes.

Arrow tool

The Arrow tool must be selected before you can manipulate arrows. To select the Arrow tool, click on the arrow button () in the floating tool palette. Notice that the cursor changes to an arrow.



Arrows and nodes

Arrow from Variable node to Variable node

Indicates that the target Variable depends on the origin Variable.

Arrow from Variable node to module node

Indicates that at least one Variable in the target module depends on the origin Variable.

Arrow from module node to Variable node

Indicates that the target Variable depends on at least one Variable in the origin module.

Arrow from module node to module node

Indicates that the target module contains at least one Variable that depends on at least one Variable in the origin module.

Double-headed arrow between module nodes

Indicates that each module contains at least one Variable that depends on at least one Variable in the other module.

Small arrowhead to the right or left of a Variable node

Indicates that the Variable has a remote input or output—a Variable that is not inside the displayed Variable's module (see "Finding remote inputs and outputs" on page 29).

Creating and removing arrows

To draw an arrow, first be sure the arrow tool () is selected.

1. Drag from the origin node (it becomes highlighted) to the destination node (which also becomes highlighted).
2. When you release the mouse button, the arrow is drawn.

To draw multiple arrows to a single destination node, select all the origin nodes. Then drag from any origin node to the destination node.

To remove an arrow, do one of the following:

- Select the arrow, then press the *Backspace* or *Delete* key.
- Repeat the process of drawing an arrow from the origin node to the destination node.

Analytica Note: *An arrow is also drawn whenever the identifier of a Variable is added to the definition of a Variable (see "Creating or editing a definition" on page 145).*

Model changes when creating an arrow

Creating an arrow between two nodes changes the model. The change depends on the classes of the two nodes.

Arrow between two Variable nodes

When you draw an arrow from a Variable node *A* to another Variable node *B*, *A* becomes an input of *B*. You can then use

this input in creating or editing the definition of *B* (See Chapter 8, “Creating and Editing Definitions”).

Arrow between a Variable node and a module node

When you draw an arrow from a Variable into a module, or from a module to a Variable, Analytica creates an alias of the Variable inside the module (see “Alias nodes” on page 82). You can then open the module and draw arrows between the alias and other Variables in the module.

Arrow between two module nodes

When you draw an arrow from one module node to another, Analytica creates a new Variable node in the first module and an alias of that Variable in the second module (see “Alias nodes” on page 82). You can then open each module and draw arrows between the new nodes and other Variables in the module.

Model changes when deleting an arrow

If *B* already has a definition that includes *A* and you delete the arrow from *A* to *B*, Analytica removes *A* from *B*’s definition. For example, if *A* is an index and *B* is defined as a table, Analytica removes *A* as an index of *B*.

In some cases, removing *A* does not leave a valid definition for *B* (for example, when *A* is part of a mathematical expression such as `1/A`). Under these circumstances, *A* is replaced with the keyword `Expr` and the entire expression is surrounded with `FunctionOf()`. This notation indicates that the definition is invalid and must be edited.

For example, suppose you had a definition that was *X*+*Y* and then deleted *X*. The definition would be replaced with `FunctionOf( expr + Y )`

Cyclic dependency

A ***cyclic dependency*** occurs when a Variable depends on itself directly or indirectly so that the arrows form a directed circular path.

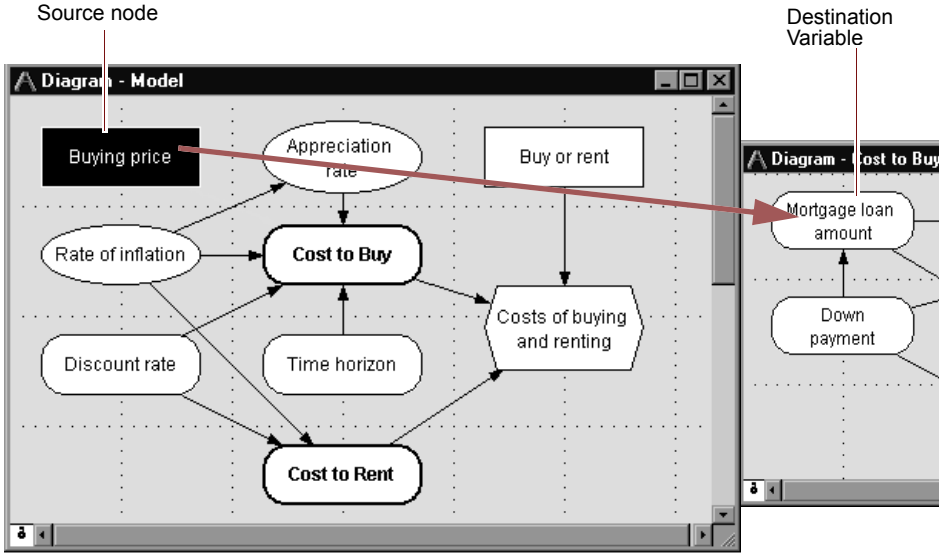
A cyclic dependency is permitted only in a dynamic model (see Chapter 16, “Modeling Changes over Time”), provided that the Variable depends on its value in an earlier time period.

Arrows between Variables in different modules

There are two direct methods and two indirect methods for drawing an arrow between two Variables in different modules. The following examples demonstrate the direct methods of drawing an arrow from the Variable *Buying price* to the Variable *Mortgage loan amount* in another module (see the following figure).

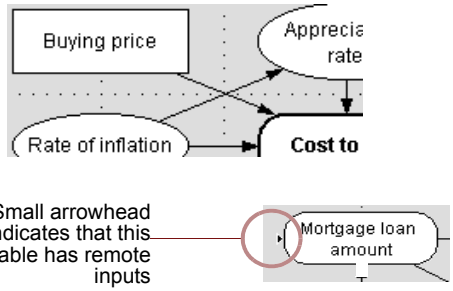
Drawing arrows across windows

1. Open both module Diagram windows and bring to the top the module Diagram window containing the origin Variable, *Buying price*.
2. Position the Diagram windows so the Variable *Mortgage loan amount* is exposed in the window underneath.
3. Draw an arrow from the origin Variable (*Buying price) to* the second Variable (*Mortgage loan amount*).



Result

An arrow points from *Buying price* to the *Cost to Buy* module; a small arrowhead points into *Mortgage loan amount* to indicate that a source node is in a different diagram.



Moving, drawing, and moving back

1. Select *Mortgage loan amount*, then choose **Move Into Parent** from the **Diagram** menu to move the Variable into the parent diagram.
2. Draw an arrow from *Buying price* to *Mortgage loan amount*.
3. Move *Mortgage loan amount* back into its module by dragging it onto the *Cost to Buy* module node.

Indirectly drawing an arrow

The two indirect methods of drawing an arrow between two Variables in different modules are:

- Edit the definition of the target Variable, entering the identifier of the input Variable directly (see “Creating or editing a definition” on page 145). The arrow appears when the definition is accepted.
- Use an alias node (see “Alias nodes” on page 82).

Alias nodes

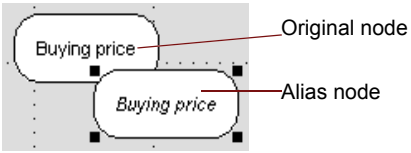
An ***alias*** is a copy of a node, referring to the same Variable or module as the original node. You can use an alias node to display the same Variable in more than one module diagram. For example, often the inputs to a Variable are in one module, while its outputs are in other modules. To display an input Variable’s node in the modules containing the outputs, create an alias in each of those modules.

A Variable or module can have only a single original node. You can create an unlimited number of alias nodes for any original node.

Create an alias in any of the three following ways.

Use the Make Alias command

Select the original node. Then choose the **Make Alias** option from the **Object** menu(Ctrl-M). The alias node appears next to the original node. You can then move it into another module by dragging it.



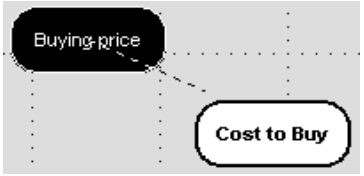
Use this method to make an alias of a module, if you want to show a module node in more than one diagram.

Draw arrow between Variable and module

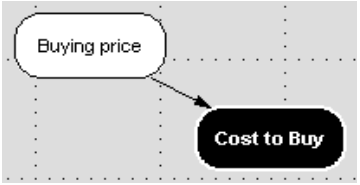
Draw an arrow from the original node to a module node, or from the module node to the original node. An alias for the original node appears in the module.

Example

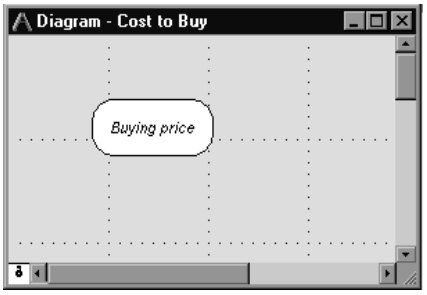
Draw an arrow from a Variable (*Buying price*) to a module (*Cost to Buy*).



An arrow is displayed from the *Buying price* Variable to the *Cost to Buy* module.

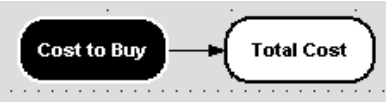


The new alias node appears in the diagram for the *Cost to Buy* module.

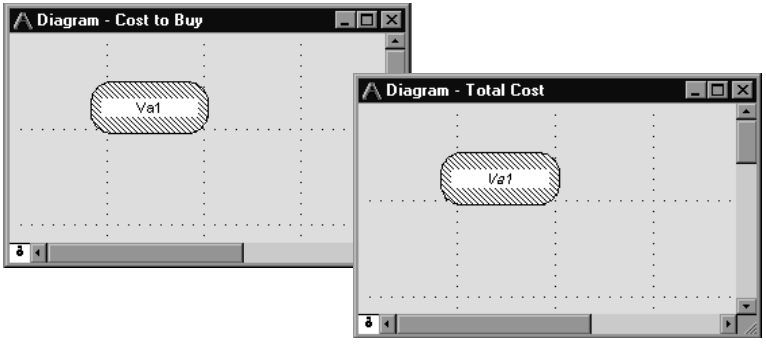


Draw arrow between two modules

Draw an arrow from one module node (*Cost to Buy*) to another module node (*Total Cost*).



Analytica creates a new Variable node with a default title, such as *Va1*, in the first module, and creates an alias of *Va1* in the second module.



Alias nodes

An alias looks and behaves similar to the original node that it is derived from, except that its title is in italics.

Analytica Note: *An alias of a module does not display any input or output arrows.*

You can treat an alias node just as if it were the original node. Click once on an alias to select it (for example, to display its result). Double-click on an alias to open the Object window for the original node.

To use a Variable with an alias as an input to another node, draw an arrow either from the original node or from its alias.

To create a new input to a Variable with an alias, draw an arrow either to the Variable or to its alias. An arrow to an alias of a Variable creates a corresponding arrow to its original node in its diagram, if the original node is in the same module as the new input or an alias of the new input.

Modifying an alias node

When you create an alias node, it looks just like its original node, including its node shape, color, label font, and icons. The only difference is that the title is in italics.

If you edit the title of an alias, the title of the original node changes to match the alias. Conversely, if you edit the title of the original node, the alias's title changes to match the original.

To change the appearance of an alias node alone, use the **Set Node Style** option from the **Diagram** menu (see "Node Style dialog box" on page 123). If you use the Node Style dialog box to change the appearance of an alias node, its original node does not change. Similarly, using the Node Style dialog box to change the appearance of an original node does not affect any of its previously created aliases.

Editing Attributes

You can edit a user-modifiable Attribute either in the Attribute panel (see "The Attribute panel" on page 34) or in the Object window (see "The Object window" on page 32). To change a node's class, see "Changing the class of a node" on page 86.

To edit an Attribute:

1. If in the Attribute panel, select the Attribute in the Attribute popup menu.

2. Click in the Attribute field. If a gray outline appears around the Attribute and the cursor is blinking, the Attribute is user-modifiable.
3. Edit the Attribute using the standard text-editing methods.

The edited text is stored when you click anywhere outside the Attribute field, or when you press *Alt-Enter*.

Attribute changes

Any changes to Attributes propagate to all other displayed windows. For example, if you change the title of an Object, the new title is displayed in that Object's diagram. If you change the definition of a Variable, the arrows are redrawn to reflect changes in dependencies.

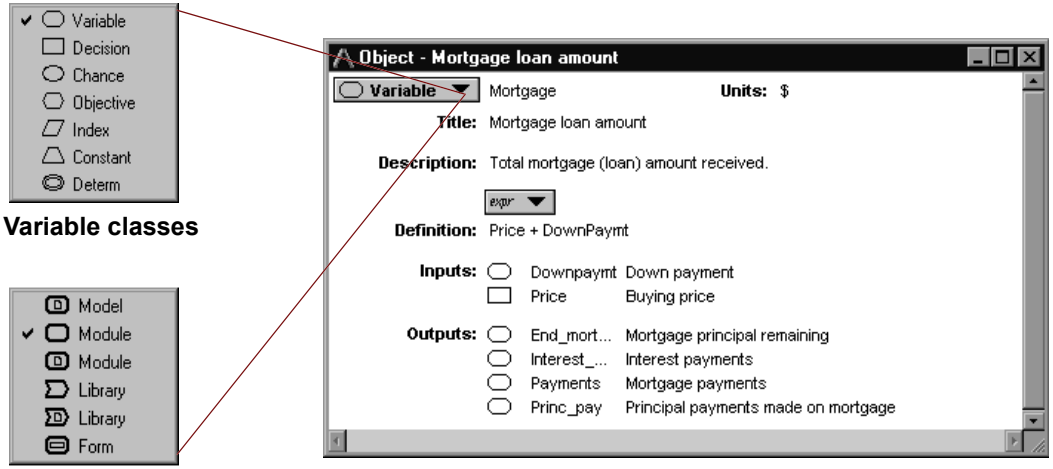
Cancel and Undo

While you are editing an Attribute, you can cancel and revert to the previous value at any time by pressing the *Esc* (escape) key. If you have finished entering the value of the Attribute, but now want to revert to the previous value of the Attribute, select **Undo** from the **Edit** menu(Ctrl-Z).

Changing the class of a node

Use the Class popup menu to change the class of a node. This menu appears:

- In the top left corner of the Object window.
- In the Attribute panel when **Class** is the selected Attribute.



The contents of the Class popup menu depend on whether the node is a Variable or a module (see the preceding figure).

Analytica Note: *You cannot change a Variable into a module, or vice versa. You also cannot change a function into a Variable or module, or vice versa.*

To change a node's class, press on the Class popup menu and select another class.

Module Classes

The Variable classes are described in the section "Node types" on page 30. The module classes are described below.

Model

A module or hierarchy of linked modules that you work on during a session with Analytica. A model is saved in a file (an Analytica document) between sessions. Only a model saves preferences (see "Preferences dialog box" on page 88) and uncertainty options (see "Uncertainty Setup dialog box" on page 291).

Module

A collection of nodes that are displayed in a single diagram. A module is depicted on its parent diagram as a rounded node

with a thick outline.

Filed module

A module whose contents are saved in a file separate from the model that contains it. A filed module can be shared among several models, without having to make a copy for each model. See “Using filed modules and libraries” on page 404

Library

A module that contains functions and/or Variables. User libraries appear in the **Definition** menu below the system function libraries, giving easy access to their contents. See “Libraries” on page 429

Filed library

A library whose contents are saved in a file separate from the model that contains it. A filed library can be shared among several models, without having to make a copy for each model. See “Using filed modules and libraries” on page 404

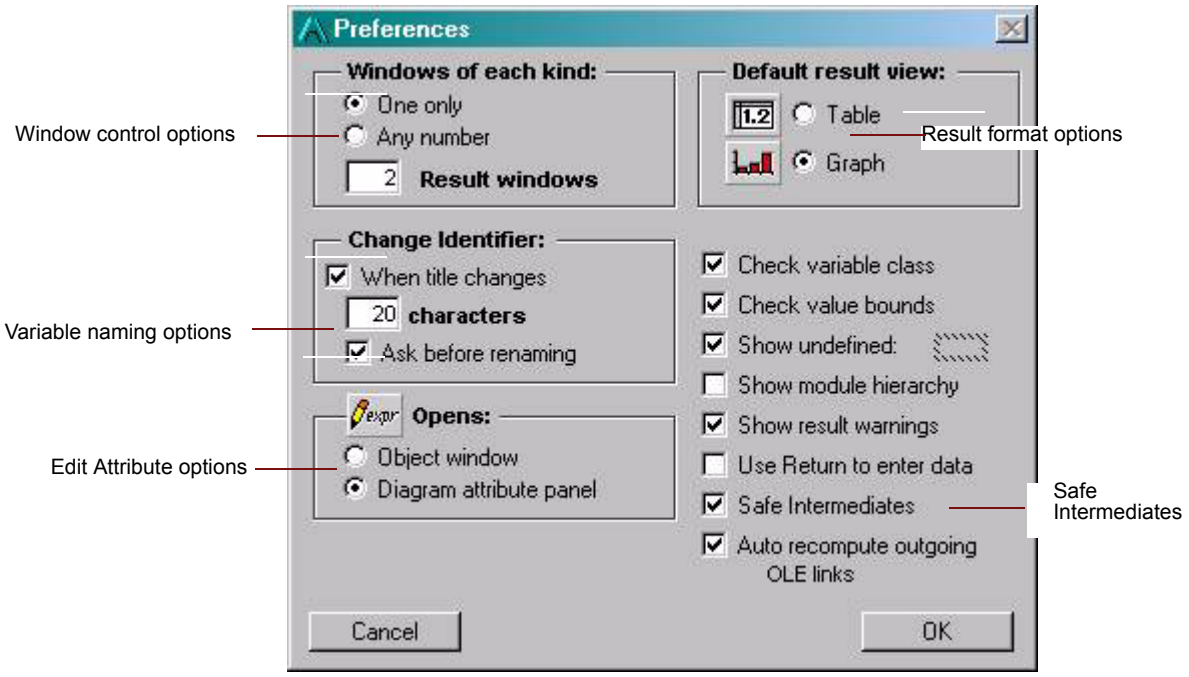
Form

A module that creates an input node alias or an output node alias when you draw an arrow from a node to the form. See Chapter 9, “Creating Models Used by Others”.

Preferences dialog box

Use the Preferences dialog box to inspect and set a variety of preferences for the operation of Analytica. All preference settings are saved with a model of class Model.

To display the Preferences dialog box, select **Preferences** from the **Edit** menu.



Windows of each kind

Use the options in this box to control how many windows of various kinds are displayed at once (see “Managing windows” on page 412).

One only

Check this box to close an existing window (if there is one) whenever you open a new window.

Any number

Check this box to keep all windows open until you explicitly close them.

Result windows

Enter a value in this field to indicate the number of Result windows that you can keep open simultaneously. The default

(and minimum) number is 2; the maximum number is 20.

Change identifier

Use the options in this box to control the changing of identifiers. See “Creating and editing nodes in a diagram” on page 73 for a description of how identifiers are initially assigned.

When title changes

Check this box to change a Variable’s identifier whenever you change its title. Analytica uses up to the number of specified characters (20 by default, range from 2 to 20), replacing spaces and returns with an underscore character (_), and omitting anything between parentheses.

If the box is not checked, the identifier is changed only when you explicitly edit it.

Ask before renaming

Check this box to see a confirmation dialog box before automatic changing of a Variable’s identifier.

Opens

Use the options in this box to specify the window that displays when you select the edit definition button (Ctrl-E). When you are prompted (for example, via an error message) to edit a definition, and you click on OK, this preference setting determines the window to display.

Object window

Select this option to open the Object window and select the definition text.

Diagram Attribute panel

Select this option to open the Attribute panel on the appropriate Diagram window and select the definition text.

Default result view

Use the options in this box to control how data appear in Result windows the first time a result is calculated for a node (see “2: Results” on page 45).


Select this option to display a table.


Select this option to display a graph.

Checkboxes

Use these checkboxes to control various aspects of how Analytica looks and behaves.

Check Variable class

If checked, a warning displays for the following inconsistencies between a Variable’s class and definition:

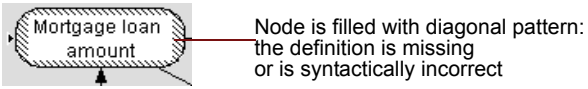
- A non-chance Variable whose definition includes a probability distribution.
- A constant whose definition is dependent on any other Variables. However, a constant defined as a table may have indexes as inputs.
- An index Variable defined as a single value, array, or any function other than `Sequence()`.

Check value bounds

If checked, the check Attributes are computed. See “Checking the validity of a Variable’s values” on page 156

Show undefined

If checked, nodes without a valid definition display with a cross-hatch pattern.



Show module hierarchy

If checked, a bar at the top of each Diagram window indicates

the hierarchy depth of the active module. See “Show module hierarchy preference” on page 396

Show result warnings

If checked, when a warning condition is encountered during result evaluation, evaluation is interrupted, and a warning message displays for action. If unchecked, when a warning condition is encountered during result evaluation, no warning message is displayed, and evaluation continues.

Use Return to enter data

A standard windows keyboard has a Return key located on the alphanumeric section of the keyboard, and a separate Enter key located on the numeric keyboard. When this checkbox is unchecked (the default), the return key starts a new line in a multi-lined text field (such as a definition) while the Enter key or Alt-Return signal that the data entry is complete. When this checkbox is checked, these are reversed, with Enter or Alt-Return starting a new line and Return completing the entry of data.

Safe Intermediates

In the most general case, Intelligent Array Abstraction requires an extra internal, but somewhat costly, step during evaluation to make sure all intermediate arrays are fully rectangular. Skipping this step seldom has an impact on the final result, but can speed things up dramatically for certain models, especially those using dynamic simulation extensively. Unfortunately, in the very rare cases where it does make a difference, skipping the step can lead in incorrect results.

You can control this with the **Safe Intermediates** checkbox. If there is a check in the **Safe Intermediates** checkbox, Analytica will make sure all intermediate arrays are fully rectangular. Removing the check will speed up performance, but might cause incorrect results. By default, Analytica 3.0 uses the safe but slower setting.

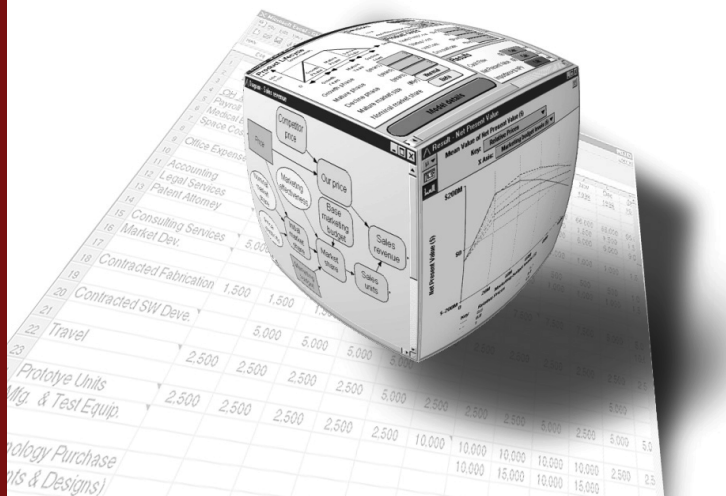
Auto recompute outgoing OLE links

If you have used the OLE linking feature to linked an Analytica result in your model to an external application, this checkbox controls whether this data is automatically

recomputed and updated whenever the result, or anything the result depend on, changes. When making several changes in computationally intensive models, it is often convenient to turn this checkbox off so that large recomputations after each small change. See OLE linking in “17: Importing, exporting, & OLE linking data” on page 375.

Chapter 5

Building Effective Models



In this Chapter

This chapter shows you how to build models that are

- Focused
- Simple
- Clear
- Comprehensible
- Correct

5: Building effective models

Creating useful models is a challenging activity, even for experienced modelers; effective use of Influence Diagrams can make the process substantially easier and clearer. This chapter provides tips and guidelines from master modelers (including Newton and Einstein) on how to build a model that is effective, one that focuses on what matters, and that is simple, clear, comprehensible, and correct. The key is to start simple and progressively refine and extend the model where tests of initial versions suggest it will be most important.

Most of the material in this chapter, unlike the other chapters in this *User Guide*, is not specific to Analytica. These guidelines are useful whether you are using Analytica, a spreadsheet, or any other modeling tool. However, Analytica makes it especially easy to follow these guidelines, using its hierarchical Influence Diagrams, uncertainty tools, and Intelligent Arrays.

These guidelines have been distilled from many years of experience by master modelers, using Analytica and a variety of other modeling software. However, they are general guidelines, not rules to be adhered to absolutely. We suggest you read this chapter early in your work with Analytica and revisit it from time to time as you gain experience.

Creating a model

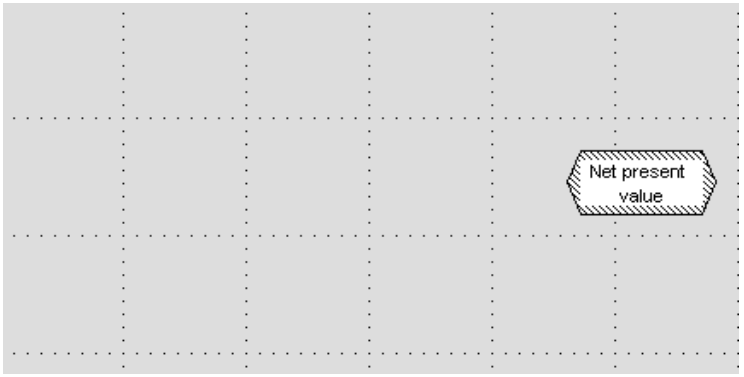
Below are general guidelines to help you build models that provide the greatest value with the least effort.

Identify the objectives

What are the objectives of the decision maker? Sometimes the objective is simply to maximize expected monetary profit. More often there are a variety of other objectives, such as maximizing safety, convenience, reliability, social welfare, or environmental health, depending on the domain and the decision maker. Utility theory and multiattribute decision analysis provide an array of methods to help structure and quantify objectives in the form of utility. Whatever approach you take, it is important to represent the objectives in an explicit and quantifiable form if the objectives

are to be the basis for recommending one decision option over another.

It is a useful convention to put the objective Variable or Variables (hexagonal nodes) on the right of the diagram window, leaving space on the left side for the rest of the diagram.



The most common mistake in specifying objectives is to select objectives that are too narrow, by concentrating on the most easily quantifiable objective—typically, near-term monetary costs—and to forget about the other, less tangible objectives. For example:

- When buying software you may want to consider the usability and reliability of different software packages, not just cost and performance.
- In pricing a product, you may want to consider the long-term effects of increased market share in developing new customers and markets and not just short-term revenues.
- In selecting a medical treatment, you may want to consider the quality of life if you survive the treatment, and not just the probability of survival.

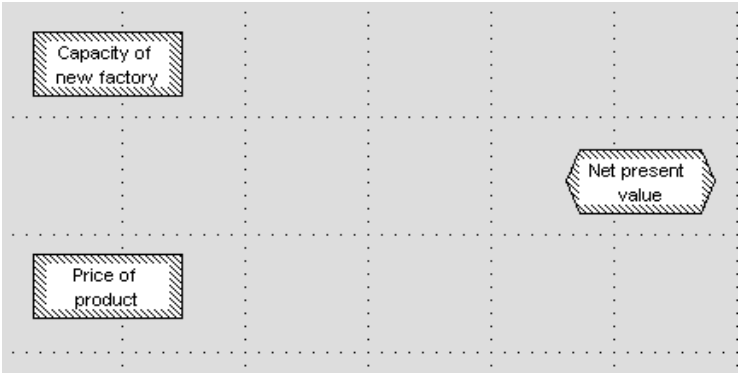
For an excellent guide on how to identify and structure objectives, see *Value-Focused Thinking* by Ralph Keeney.

Identify the decisions

The purpose of modeling is usually to help you (or your colleagues, organization, or clients) discover which decision options will best meet your (or their) objectives. You should aim, therefore, to include the decisions and objectives explicitly in your model.

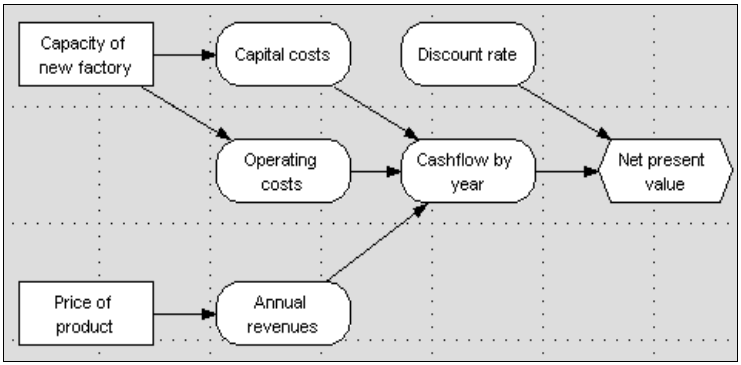
A decision Variable is one that the decision maker can affect directly—which computer to buy, how much to bid on the contract, which medical treatment to choose, when to start construction, and so on. Occasionally, people want to build a model just for the sake of furthering understanding, without explicitly considering any decisions. Most often, however, the ultimate purpose is to make a better decision. In those cases, the decision Variables are where you should start your model.

When starting a new Influence Diagram, put the decision Variables—as rectangular nodes—on the left of the diagram window, leaving space for the rest of the Influence Diagram to the right.



Link the decisions to the objectives

The decisions and objectives are the starting and ending points of your model. Once you have identified them, you have reduced the diagram construction to the process of creating the links between the decisions and objectives, via intermediate Variables. You may wish to work forward from the decisions, or backward from the objectives. Some people find it easiest to alternate, working inward from the left and the right until they can link everything up in the middle.



It helps to identify the decisions and objectives early during model construction, to maintain focus on what matters. There may be a bewildering variety of Variables in the situation that may seem to be of potential relevance. But, you only need to worry about Variables that influence how the decisions might affect the objectives. You can ignore any Variable that has no effect on the objectives.

Focus on identifying the Variables that make clear distinctions—Variables whose interpretations won't change with time or viewer. Extra effort here will be repaid in model accuracy and cogency.

Move from the qualitative to the quantitative

An Influence Diagram is a purely qualitative representation of a model. It shows the Variables and their dependencies. It is usually best to draw in most or all of the first version of your model just as an Influence Diagram, or hierarchy of diagrams, before trying to quantify the values and relationships between the Variables. In this way, you can concentrate on the essential qualitative issues of what Variables to include, before having to worry about the details of how to quantify the relationships.

When the model is intended to reflect the views and knowledge of a group of people, it is especially valuable to start by drawing up Influence Diagrams as a group. A small group can sit around the computer screen; for a larger group, it is best if you have the means to project the image onto a large screen, so that the entire group can see and comment on the diagram as they create it. The ability to focus on the qualitative structure initially lets you involve early in the process participants who might not have the time or interest to be involved in the detailed quantitative analysis. With this approach, you can often obtain valuable insights

and early buy-in to the modeling process from key people who would not otherwise be available.

Keep it simple

> "A theory should be as simple as possible, but no simpler."
> Albert Einstein

Perhaps the most common mistake in modeling is to try to build a model that is too complicated or that is complicated in the wrong ways. Just because the situation you are modeling is complicated doesn't necessarily mean your model should be complicated. Every model is unavoidably a simplification of reality; otherwise it would not be a model. The question is not whether your model should be a simplification, but rather how simple it should be. A large model requires more effort to build, takes longer to execute, is harder to test, and is more difficult to understand than a smaller model. And it may not even be more accurate.

Reuse and adapt existing models

> "If I have seen further than [others] it is by standing upon the shoulders of Giants."
> Sir Isaac Newton

Building a new model from scratch can be a challenge. If you can find an existing model for a problem similar to the one you are now facing, it is usually much easier to start with the existing model and adapt it to the new application. In some cases, you may find parts or modules of existing models that you can extract and combine to address a new problem.

To find a suitable model to adapt, you can start by looking through the example models distributed with Analytica. If there is an Analytica users' group in your own organization, it may collect a model library of classes of problems of interest to your organization.

Aim for clarity and insight

The goal of building a model is to obtain clarity about the situation, about which decision options will best further your objectives, and why. If you are already clear about what decision to make, you don't need to build a model, unless, perhaps, you are trying to clarify the situation and explain the recommended decisions for others. Either way, your goal is greater clarity. This goal

is another reason to aim for simplicity. Large and complicated models are harder to understand and explain.

Testing and debugging a model

Even with Analytica, it is rare to create the first draft of a model without mistakes. For example, on your first try, definitions may not express what you really intended. It is important to test and evaluate your model to make sure it is expresses what you have in mind. Analytica is designed specifically to make it as easy as possible to scrutinize model structures and dependencies, to explore model implications and behaviors, and to understand the reasons for them. Accordingly, it is relatively easy to debug models once you have identified potential problems.

Test as you build

With Analytica, you can evaluate any Variable once you have provided a definition for the Variable and all the Variables on which it depends, even if many other Variables in the model remain to be defined. We recommend that you evaluate each Variable as soon as you can, immediately after you have provided definitions for the relevant parts of the model. In this way, you'll discover problems as soon as possible after specifying the definitions that may have caused them. You can then try to identify the cause and fix the problem while the definitions are still fresh in your memory. Moreover, you'll be less likely to repeat the mistake in other parts of the model.

If you wait until you believe you have completed the model before testing it, it may contain several errors that interact in confusing ways. Then you'll have to search through much larger sections of the model to track them down. But if you have already tested the model components independently, you'll have already removed most of the errors, and it will usually be much easier to track down any that remain.

Test the model against reality

The best way to check that your model is well-specified is to compare its predictions against past empirical observations. For example, if you're trying to predict future changes in the composition of acid rain, you should try to compare its "predictions" for past years for which you have empirical observations. Or, if

you're trying to forecast the future profitability of an existing enterprise, you should first calibrate your model for past years for which accounting data are available.

Test the model against other models

Often you don't have the luxury of empirical measurements or data for the system of interest. In some cases, you're building a new model to replace an old model that is out-of-date, too limited, or not probabilistic. In these cases, it is usually wise to start by reimplementing a version of the old model, before updating and extending it. You can then compare the new model against the old one to check for discrepancies. Of course, differences may be due to errors in the new model or the old model. Once you have resolved any discrepancies, you can be confident that you are building on a foundation that you understand.

If the model is hard to test against reality in advance of using it, and if the consequences of mistakes could be catastrophic, you can borrow a technique that NASA uses widely for the space program. You can get two independent modelers (or two modeling teams) each to build their own model, and then check the models against each other. It is important that the modelers be independent, and not discuss their work ahead of time, to reduce the chance that they will both make the same mistake. For a sponsor of models for critical applications in public or private policy, this multiple model approach can be very effective and insightful. The competition keeps the modelers on their toes. Comparing the models' structure and behavior often leads to valuable insights.

Have other people review your model

It's often very helpful to have outside reviewers scrutinize your model. Experts with different views and experiences may have valuable comments and suggestions for improving it. One of the advantages of using Analytica over conventional modeling environments is that it's usually possible for an expert in the domain to review the model directly, without additional paper documentation. The reviewer can scrutinize the diagrams, the Variables, their definitions, and the behavior of the model electronically. You can share models electronically on diskette, over a network, or by electronic mail.

Test model behavior and sensitivities

Many problems become immediately obvious when you look at a result—for example, if it has the wrong sign, the wrong order of magnitude, or the wrong dimensions, or if Analytica flags an evaluation error. Other problems, of course, are not immediately obvious—for example, if the value is wrong by only a few percentage points. For more thorough testing, it is often helpful to analyze the model behavior by specifying a list of alternative values for one or two key inputs (see Chapter 3, “Analyzing Model Behavior”), and to perform sensitivity analysis (see Chapter 15, “Sensitivity and Uncertainty Analysis”). If the model behaves in an unexpected way, this may be a sign of some mistake in the specification. For example, suppose that you are planning to borrow money to buy a new computer, and the net value increases with the interest rate on the loan; you might suspect a problem in the model.

Celebrate and learn from unexpected behavior

If analyzing the behavior or sensitivities of your model creates unexpected results, there are logically two possibilities:

- Your model contains an error, in that it does not correctly express what you intended.
- Your expectations about how the model should behave were wrong.

You should first check the model carefully to make sure it contains no errors, and does indeed express what you intended. Explore the model to try to figure out how it generates the unexpected results. If after thorough exploration you can find no mistake, and the model persists in its unexpected behavior, do not despair! It may be that your intuitions were wrong in the first place. This discovery should be a cause for celebration rather than disappointment. If models always behaved exactly as expected, there would be little reason to build them. The most valuable insights come from models that behave counter-intuitively. When you understand how their behavior arises, you can deepen your understanding and improve your intuition—which is, after all, a fundamental goal of modeling.

Document the model as you build it

Give your Variables and modules meaningful titles, so that others—or you, when you revisit the model a year later—can more

easily understand the model from looking at its Influence Diagrams. It's better to call your Variable *Net rental income* than *NRI23*.

It's also a good idea to document your model as you construct it by filling in the Description and Units attributes for each Variable and module. You may find that entering a line or three of description for each Variable explaining clearly what the Variable represents will help to keep you clear about the model. Entering units of measurement for each Variable can help you avoid simple mistakes in model specification. Avoid the temptation to put documentation off until the end of the project, when you may run out of time, or may have forgotten key aspects.

Most models, once built, spend the majority of their lives being used and modified by people other than their original author. Clear and thorough documentation pays continuing dividends; a model is incomplete without it.

Expanding your model

Extend the model by stages

The best way to develop a model of appropriate size is to start with a very simple model, and then to extend it in stages in those ways that appear to be most important. With this approach, you'll have a usable model early on. Moreover, you can analyze the sensitivities of the simple model to find out where the key uncertainties and gaps are, and use this to set priorities for expanding the model. If instead you try to create a large model from the start, you run the risk of running out of time or computer resources before you have anything usable. And you may end up putting much work into creating an elaborate module for an aspect of the problem that turns out to be of little importance.

Identify ways to improve the model

There are many ways to expand a model:

- Add Variables that you think will be important.
- Add objectives or criteria for evaluating outcomes.

- Expand the number of decision options specified for a decision Variable, or the number of possible outcomes for a discrete chance Variable.
- Expand a single decision into two or more sequential decisions, with the later decision being made after more information is revealed.
- For a dynamic model, expand the time horizon (say, from 10 years to 20 years) or reduce the time steps (say, from annual to quarterly time periods).
- Disaggregate a Variable by adding a dimension (say, projecting sales and costs by each division of the company instead of only for the company as a whole).

Before plunging in to one of these approaches to expanding a model, it's best to list the alternatives explicitly and think carefully about which is most likely to improve the model the most for the least effort. Where possible, perform experiments or sensitivity analysis to figure out how much effect alternative kinds of expansion may have.

Changing the size or numbers of dimensions of tables is a difficult and time-consuming task in conventional modeling environments. Analytica makes it relatively easy, since you only need to change those definitions that directly depend on the dimension (for example, the Edit Tables).

Discover what parts are important to guide expansion

A major advantage of starting with a simple model is that you use it to guide extensions in the ways that will be most valuable in improving the model's results. You can analyze the sensitivities of the simple model (for example, using importance analysis, as described in "Importance analysis" on page 343) to identify which sources of uncertainty contribute most to the uncertainty in the results. Typically, only a handful of Variables contribute the lion's share of the overall uncertainty. You can then concentrate your future modeling efforts on those Variables and avoid wasting your energy on Variables whose influence is trivial.

Early intuitions about what aspects of a model are important are frequently wrong, and the results of the sensitivity analysis may come as a surprise. Consequently, it's much safer to base model development on sensitivity analysis of simple models than to rely on your intuitions about where to spend your efforts in model construction.

Once you have identified the most important Variables in your simple model, there are several ways to reduce the uncertainty they contribute. You can refine the estimated probability distribution by consulting a better-informed expert, by analyzing more existing data, by collecting new data, or by developing a more elaborate model to calculate the Variable based on other available information.

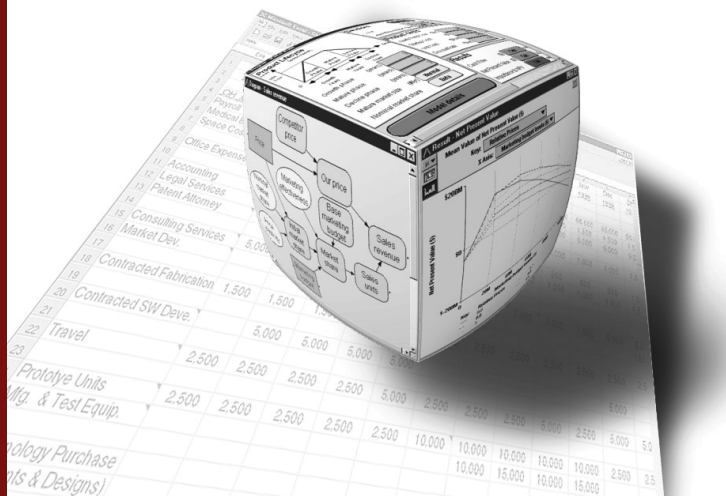
Simplify where possible

There's no reason that a model must grow successively more complex as you develop it. Sensitivity analysis may reveal that a Variable or submodel is just not very important to the results. In this case, consider eliminating it. You may find that some dimensions of tables are unimportant—for example, that there's little difference in the performance of different divisions. If so, again, consider aggregating over the divisions and eliminating that dimension from your model.

Simplifying a model has many benefits. It becomes easier to understand and explain, faster to run, and cheaper to maintain. These savings may afford you the opportunity to elaborate on more significant aspect of the model.

Chapter 6

Creating Lucid Influence Diagrams



In this Chapter

This chapter shows you how to:

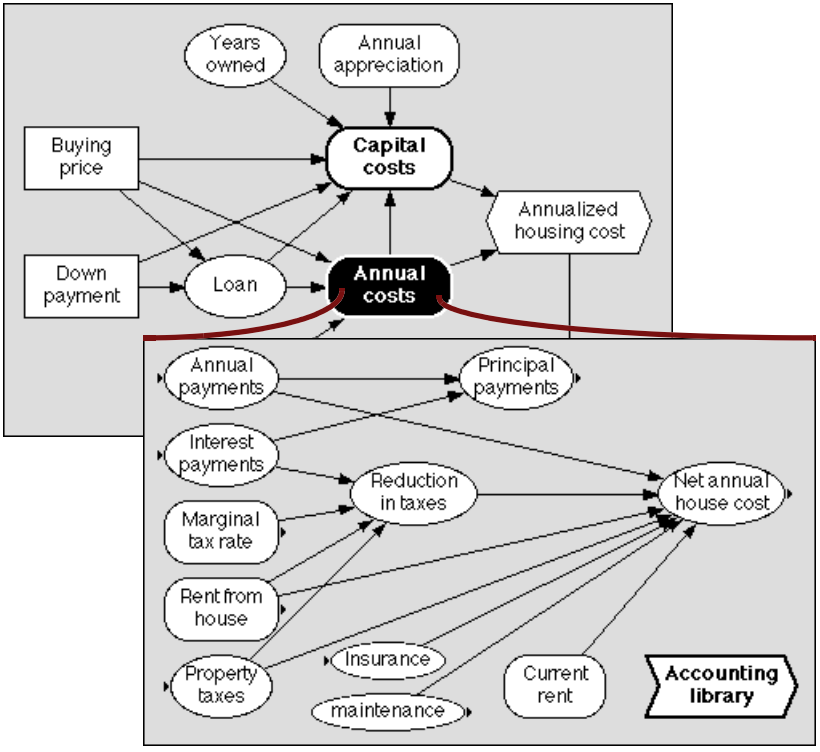
- Build Influence Diagrams
- Customize your diagrams

Chapter 6

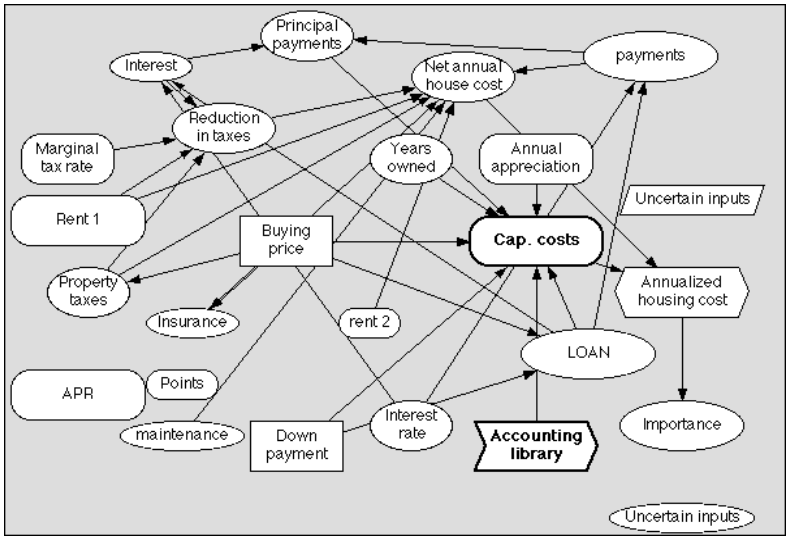
6: Creating lucid Influence Diagrams

This chapter presents guidelines for building Influence Diagrams in Analytica and explains how to customize your diagrams.

Hierarchical Influence Diagrams can provide an intuitive form to display the essential qualitative structure of a model with great clarity, uncluttered by the quantitative details.



It is also possible to create Influence Diagrams that are impenetrable spaghetti!

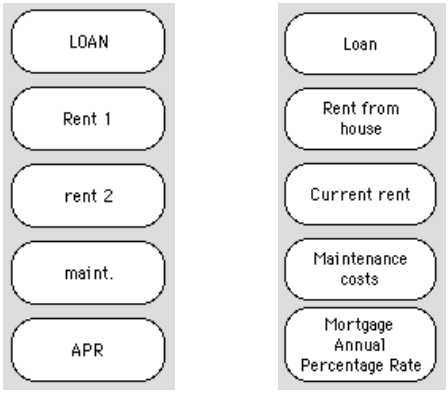


Guidelines for creating lucid and elegant diagrams

Where aesthetics are involved, rules cannot be hard and fast. You may want to adapt and modify these guidelines to suit your particular applications.

Use clear, meaningful node titles

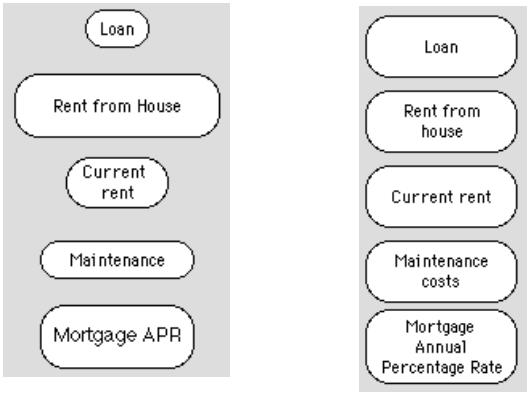
Aim to make each diagram stand by itself and be as comprehensible as possible. Each node title can contain up to 255 characters of any kind, including spaces. Use clear, concise language in titles, not private codes or names (as are often used for naming computer Variables). Mixed-case text (first letter uppercase and remaining letters lowercase) is clearer than all letters uppercase.



Poor Object titles Good Object titles

Use consistent node sizes

Diagrams usually look best if most of the Variable nodes are of the same size, rather than sized to fit their title text.



Inconsistent node sizes Consistent node sizes

Node sizes will be uniform if you set the default minimum node size in the Diagram Style dialog box (see page 121) large enough so that it will fit the full title for almost all of the nodes. The default minimum is used unless the text is too lengthy, in which case the node expands vertically to fit the text.

If you have nodes of several different sizes, you can make them more consistent by selecting **Adjust Size** (Ctrl-T) from the **Diagram** menu. All of the selected nodes are resized to the default

minimum node size, or the minimum size needed to enclose each node's title, whichever is larger.

You can also resize several nodes by the same amount simultaneously by following these steps:

1. Select the nodes to resize.
2. Resize one of the selected nodes by dragging one of its handles. All the other selected nodes are also resized.

Use small and large nodes sparingly

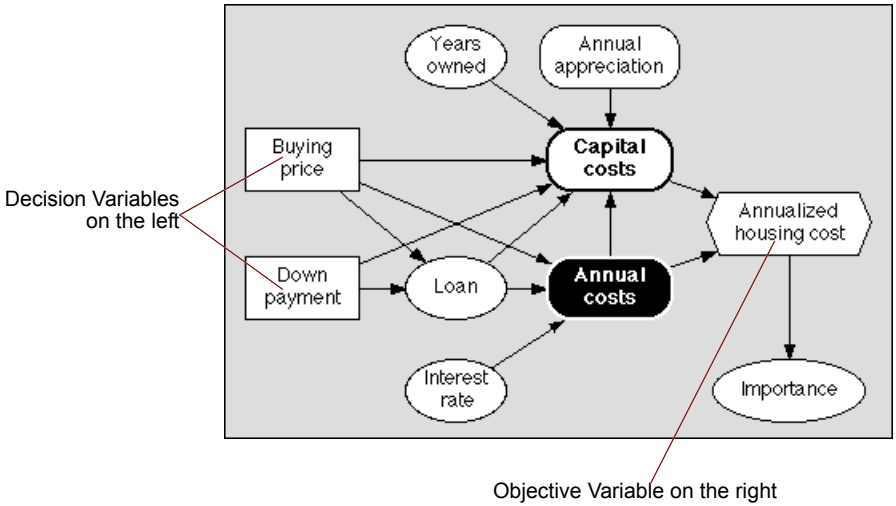
Sometimes it is more effective to make a few specialized nodes extra large or small. For example, start and end nodes, which may link to other models, often look best when they are very small. Conversely, you may want to make key input nodes containing large tables or model nodes containing the "guts" of a model unusually large to convey their importance.

Arrange nodes from left to right (or top to bottom)

People like to read diagrams, like text, from left to right, or top to bottom.[1] Try to put the decision node(s) on the left or top and the objective node(s) on the right or bottom of the diagram, with all of the other Variables or modules arranged between them.

You may want to allow a few arrows to go counter to the general flow in order to reduce crossing arrows, or overlaps. In dynamic models, there may be feedback loops (depicted with dashed arrows), which may appropriately go counter to the general flow.

1. For applications in Arabic, Hebrew, or other languages written from right to left, you may want to reverse this convention.



Tolerate spaghetti at first…

It is often hard to figure out a clear diagram arrangement in advance. It is usually easiest to start a new model using the largest Diagram window you can get. Click the maximize box to have the diagram fill your screen. You may want to create key decisions and other input nodes near the left or top of the window, and objectives or output nodes near the right or bottom of the window. Aside from that, create nodes wherever you like, without worrying too much about clarity.

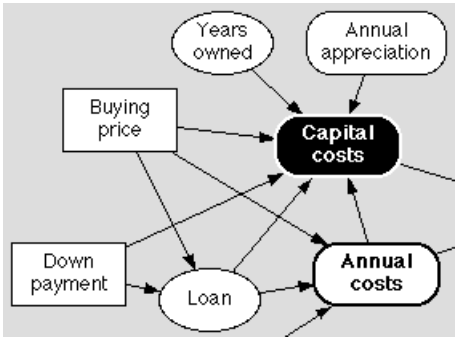
…reorganize later

When you start linking nodes, the diagram may start to look tangled. This is the time to start reorganizing the diagram to create some clarity. Try to move linked nodes together into a module. Develop vertical or horizontal lines of linked nodes. Accentuate symmetries, if you see them. Gradually, order will emerge.

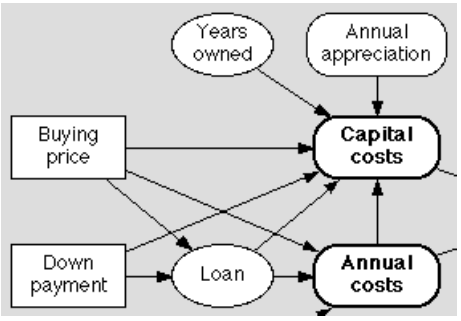
Align nodes horizontally or vertically

It usually looks best to align nodes with their centers on the same horizontal or vertical lines, so that many arrows are exactly horizontal or vertical. The square grid of 9x9 points underlying each diagram makes this easy. When **Resize Centered** is selected in

the **Diagram** menu (the default), each node is centered on a grid point.



Poor alignment



Good alignment

If nodes are not centered on a grid point, re-center them by following these steps:

1. Select all nodes in the diagram with the **Select All** (Ctrl-A) command from the **Edit** menu.
2. Select **Align Selection To Grid** from the **Diagram** menu.

Hide less important arrows

Sometimes nodes are so interrelated that it is hard or impossible to arrange a diagram to avoid arrows crossing each other or crossing nodes. It may be helpful to hide some arrows that show less important linkages. For example, indexes are often connected to many other Variables; therefore, hiding the arrows from indexes can greatly simplify a diagram.

You can hide all of the arrows linking indexes, functions, or modules, or the dashed feedback arrows in dynamic models, using the **Set Diagram Style** command from the **Diagram** menu (see page 121). You can also hide the input or output arrows from each node individually, using the **Set Node Style** command (see page 123).

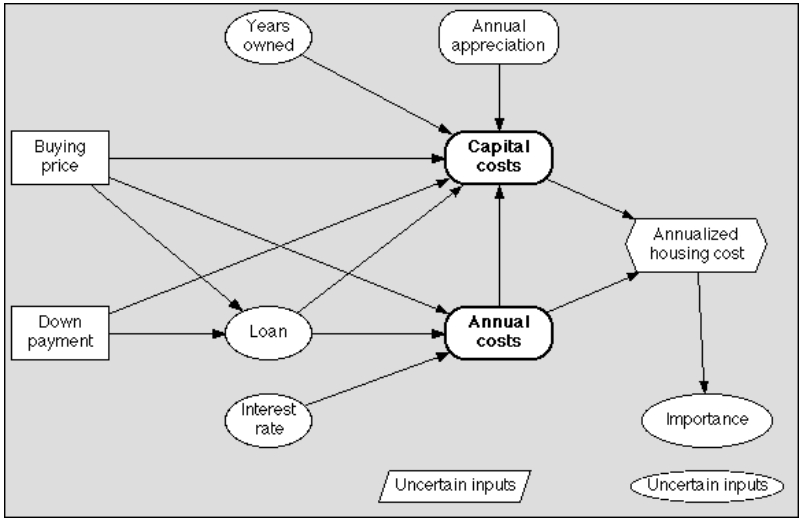
Keep diagrams compact

Screen space is valuable. To save space, keep nodes close together, leaving enough space between them for the arrows to be visible.

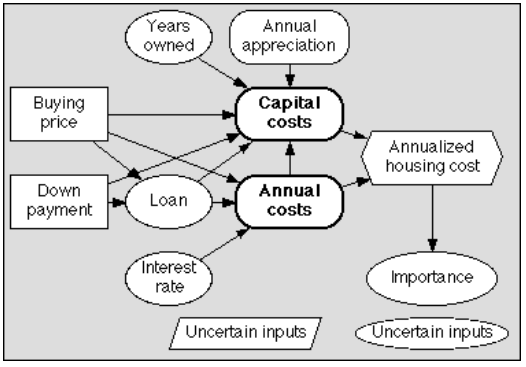
When first creating a diagram, use plenty of space. Your diagram window can be as large as your monitor screen. Using this space, find a clear arrangement, one that minimizes arrow crossing and avoids node overlaps.

After you have a clear arrangement, you can usually make the diagram more compact by moving the nodes closer together and moving the entire diagram closer to the upper left corner of the window. You can then reduce the window size to fit the diagram

A spread-out diagram



A compact diagram

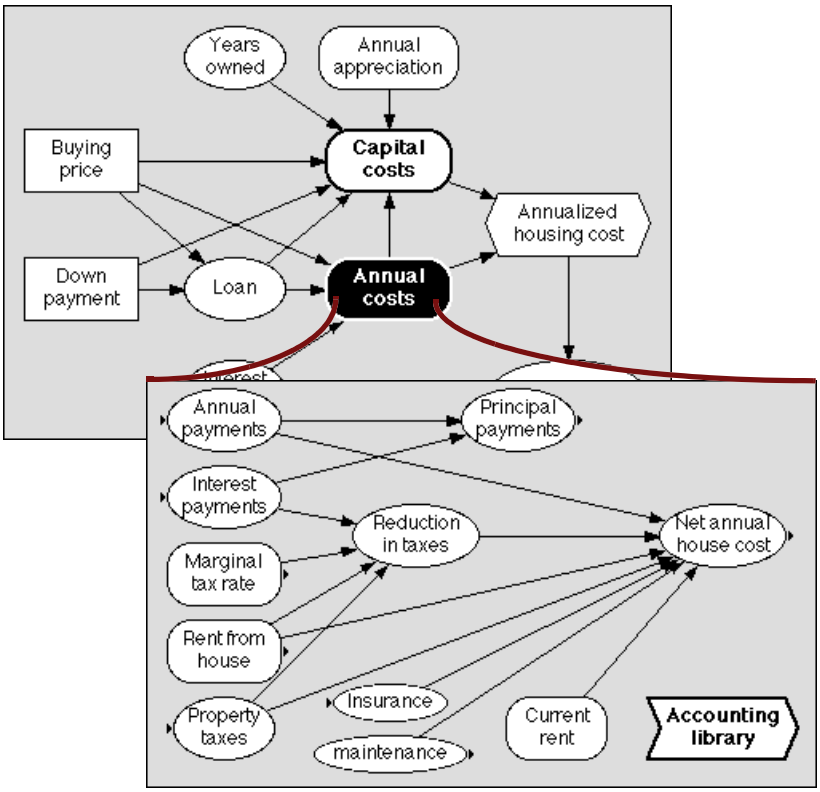


Organizing a module hierarchy

In addition to properly arranging the nodes in a single diagram, you can also improve the clarity of your models by using module hierarchies effectively.

Group related nodes in the same diagram

When assigning nodes to diagrams, the goal is to put groups of nodes with many links among them in the same diagram, and to separate them from other groups with which they have few or no links. For example, the diagram below shows that a group of nodes related to annual housing costs have been organized into the Annual costs module within the larger model.



Sometimes you have a good idea of how to group nodes before you create them. In such cases, it is easy to create the modules first, and then create and link the nodes in groups in each module.

In other cases, it may not be obvious what groupings will work best. It is then often best to create all the nodes in a single large diagram. After drawing all the arrows, you may have a confusing spaghetti diagram. At this point, try to move the nodes around to identify groups containing 5 to 15 nodes, with many links within each group and fewer links between groups. When you arrive at a

satisfactory grouping, create a module node for each group and move the group of Variables into its own module.

Use 5 to 15 nodes per diagram

In creating a hierarchy of diagrams of a model that contains 100 Variables, you could create a single module with 100 nodes, 10 modules with an average of 11 nodes each, 20 modules with 6 nodes each, or 50 modules with 3 nodes each.[2]

A module containing more than 15 nodes is often hard to decipher, unless there are very strong regularities in the structure. On the other hand, if the modules are small, averaging fewer than 5 nodes, you need so many modules that it is easy for users to get lost.

The range of 5 to 15 nodes per diagram is a good general goal. But don't feel too constrained by it if a few diagrams must be much smaller or larger than this range.

Contrast the module hierarchy in the illustration on page 118 with the spaghetti on page 111. The relationships among objects are much easier to see and understand in the model with 10 nodes in the top-level module and 12 nodes in the embedded module (page 118) than in the model with 25 top-level nodes (page 111).

Color in Influence Diagrams

Color can greatly improve the clarity and appeal of diagrams. The diagram's background and its nodes are all lightly colored by default. You can change the colors to meet your special needs.

Use colors judiciously

Selecting garish, uncoordinated colors can take attention away from the diagram. Light colors work best because the black arrows and text are easier to read over them. Analytica's default colors provide a light neutral color for the background and a slightly stronger color for the nodes.

2. Each module also creates a new node, so the total number of nodes is the number of Variables plus the number of modules.

Background color

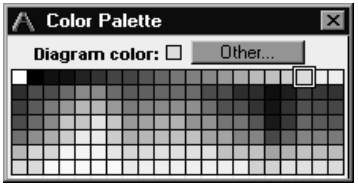
Light background colors work best so that the black arrows display clearly.

Node colors

If you wish to change the color of nodes, it is best to have all similar nodes be the same color. It generally looks messy to have nodes in many different colors.

Changing background or node colors

To change the color of the diagram background, or one or more nodes, select the Edit Tool and bring the diagram window to the front. Select **Show Color Palette** from the **Diagram** menu.



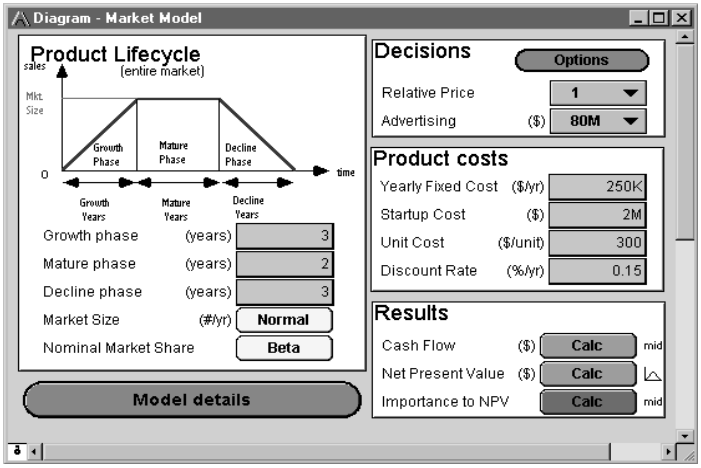
Select the node or nodes, or click in the diagram background to select the background for changing color. The current color displays in the single square at the top of the color palette. Click on a color square to select the new color.

For more color selections, click on the **Other** button to display a color wheel with the colors available on your monitor.

In the color wheel, select a color by clicking with the mouse pointer at the desired color.

Grouping nodes by color

Additional visual organization can be achieved by grouping related nodes in rectangular boxes of varying colors, as used in the following form.



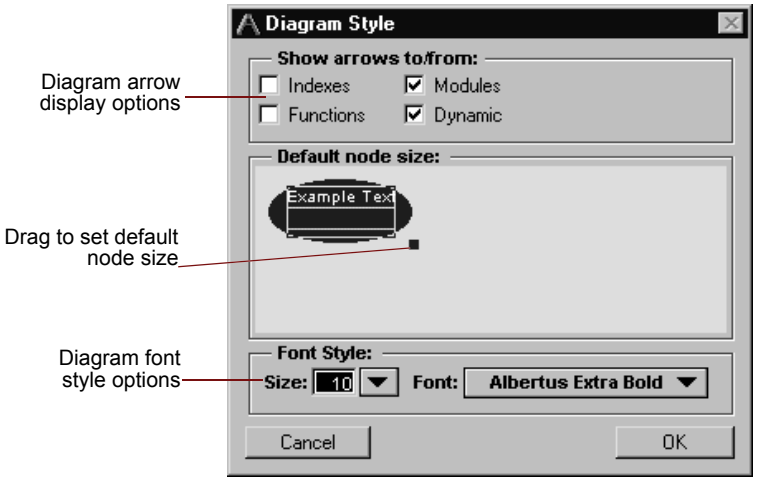
To create grouping rectangle, create a ***text node*** using the ***T*** button on the toolbar palette, leave the text blank or enter text as desired, and resize the node to the size of the group. When resizing, you may find it convenient to deselect the **Resized centered** option on the **Diagram** menu. With the node selected, check the **fill color** option on the **Set Node Style...** dialog from the **Diagram** menu, and use the color palette to choose the background color. Finally, if the rectangle is obscuring other items on the diagram, select **Send to Back** from the right mouse button menu.

If you plan to print your diagram on a black and white printer, you should select a color other than pure white, or include a node border for the group (from the **Set Node Style** dialog). Since Analytica suppresses the diagram background color when printing to a black and white printer, pure white groupings without borders will not show up on printouts.

Diagram Style dialog box

Use the Diagram Style dialog box to control various aspects of the diagram display: the default font size and typeface for the node labels, whether arrows are displayed for specified node classes, and the default node size.

To display the Diagram Style dialog box, select **Set Diagram Style...** from the **Diagram** menu.



Show arrows to/from

Use the options in this box to control various arrow displays.

Indexes	Turns on or off the display of arrows into and out of index Variables.
Functions	Turns on or off the display of arrows into and out of functions.
Modules	Turns on or off the display of arrows into and out of modules.
Dynamic	Shows and hides dynamic arrows (for Variables defined using the `Dynamic()` function, see page 362).

Default node size

Drag the handle in this box to set the default node size. When you create a new Variable or select the **Adjust Size** command from the **Diagram** menu, the node is made this size. When you change the title of a node, its size is adjusted to this size if the new title fits within it.

Font Style

Use the options in this box to set a default typeface and font (size) for all the nodes in the model.

Node Style dialog box

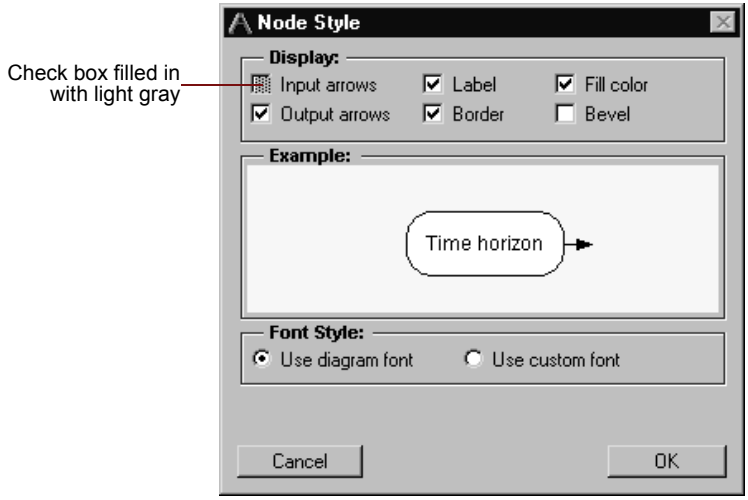
Use the Node Style dialog box to control the display of one or more nodes in a diagram.

You can specify the typeface and font (size), and whether to display the incoming arrows, outgoing arrows, the node outline, or the node label. The options for each node override the defaults specified for the entire diagram in the Diagram Style dialog box.

Changing the node style

To change the node style:

1. Select one or more nodes.
2. Choose **Set Node Style...** from the **Diagram** menu.



Display

Use the options in this box to control various display options:

Input arrows	Display arrows coming into a node.
Output arrows	Display arrows going out of a node.
Label	Display the node label (title or identifier).
Border	Display the node border.

Fill color	Display the node color. If unchecked, the node will appear transparent.
Bevel	Display the border beveling (3D button effect).

Analytica Note: *A check box filled in with light gray indicates that this option is not the same for all selected nodes. If you leave it unchanged (gray), each node keeps its current setting for this option. If you change this option (on or off), all nodes are changed to the new setting.*

Font Style

Use the options in this box to change the typeface and font (size) from the defaults (see page 122) to a custom style for the selected node(s).

Changing the size of the diagram

The diagram is preset to display and print in the orientation determined by the setting in the Page Setup dialog box. On your monitor, the size is shown by the extent of the colored background. You can change the size of the diagram in whole-page increments.

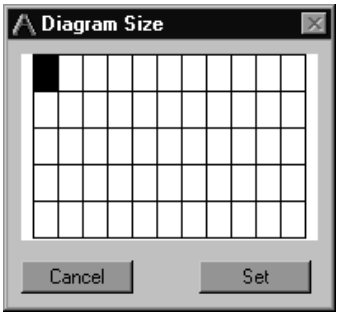
To change the size of the diagram:

1. Drag a node into the region beyond the current diagram extent.

   This causes the diagram to expand in whole page increments. If the Print Setup for the diagram is set to fit on *mxn* pages, this may simply change the location of the page breaks.

A second (obsolete) method for changing the size of a diagram is:

1. Bring the diagram window to the front.
2. Select **Set Diagram Size...** from the **Diagram** menu.



3. Specify the size diagram you want.

   Each rectangle in the grid represents a page. The size of the page is determined by the paper size and the Reduce or Enlarge percentage setting in Page Setup. To increase or decrease the size by whole-page increments, click the rectangle that you want to have as the bottom-right boundary of the diagram.

Analytica Note: *If you are decreasing the diagram size, you cannot remove pages that contain nodes.*

4. Click on **Set**.

When you save a model and later open it, the diagram size is reset to the number of pages holding the nodes.

Since the diagram extent automatically expands to be no smaller than the extent of existing nodes, use of the **Diagram Size** dialog to set the extent is only necessary if you wish to have the diagram extent larger than the extent of existing nodes. You should not explicitly set the diagram size with this method if you use or plan to use the Print Setup option **Fit on *mxn* pages**.

Taking screenshots of diagrams

This section contains some tips for taking good screenshots of Influence Diagrams and other Analytica windows for use in hard-copy documents.

Use Browse mode

When making screen captures of a Diagram window, be sure that the Browse mode () is selected rather than the Edit or Arrow

mode. The diagram is clearer in Browse mode, without the background grid visible.

Switch off cross-hatching

By default, the nodes of undefined Variables show a cross-hatched pattern around the title. To get rid of this pattern, deselect the **Show undefined** option in the Preferences dialog box (see “Preferences dialog box” on page 88).

Diagram colors

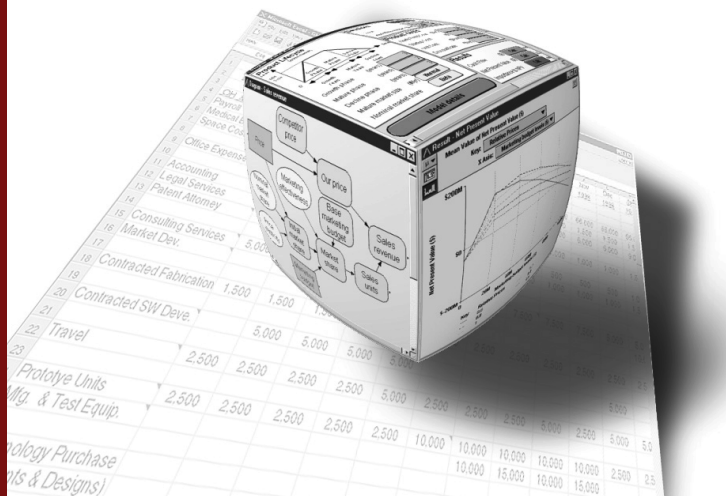
Use white for the background if you plan to print screenshots of the diagram on a black and white printer at less than 600 dpi (dots per inch). A light gray works well on a printed version if you have a 600 dpi or better printer.

Use a common level of reduction

When scaling down screenshots of windows, use a consistent reduction value. If your page setup precision bitmap alignment option is on, use a multiple of 25%; if the precision bitmap alignment option is off, use a multiple of 24%. Other reductions can create interference in printing and result in distorted screenshots.

Chapter 7

Formatting Graphs and Tables



In this Chapter

This chapter shows you how to control the display of results in graphs and tables.

7: Formatting graphs and tables

This chapter describes how to control the display of results in graphs and tables.

Graph Setup dialog box

Use the Graph Setup dialog box to select the graphing tool and control graphing options.

Display the Graph Setup dialog box in one of three ways:

- Select **Graph Setup** from the **Result** menu.
- Select **Graph Setup** from the right mouse button menu.
- Double-click on a graph in the Result window.

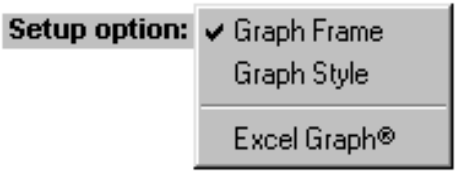
To set defaults for all new graphs, open the Graph Setup dialog box when no graph is the active window.

To establish settings for the graph of results of a specific Variable, open the Graph Setup dialog box when that graph is the active window.

The settings are saved when you save the model.

Setup option popup menu

Several options for viewing and changing settings in the Graph Setup dialog box are accessible using its **Setup option** popup menu.



Buttons

Set Default

Accepts all Graph Setup settings for the current and all future graphs, and closes the dialog box.

Cancel

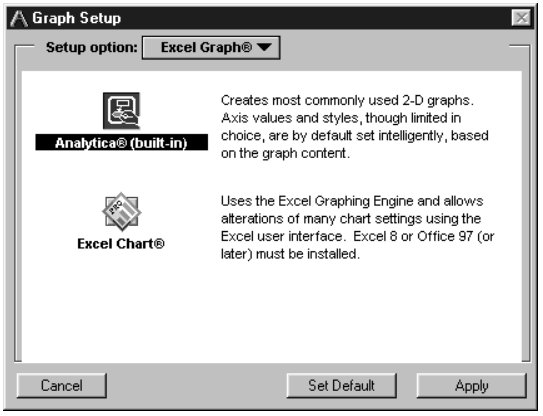
Leaves the Graph Setup settings unchanged, and closes the dialog box.

When you first open the Graph Setup dialog box for a model, the Graph Frame setup option displays.

Selecting the Graphing Tool

Analytica results may be displayed using either Analytica's built-in graphing engine, or Excel's graphing engine. To use Excel Graph you must have Microsoft Excel 8 (also known as Excel 97) or Excel 2000 installed on your computer (Excel is not included with Analytica).

To select the graphing tool, select **Excel Graph®** from the setup option pull-down menu.



Analytica® (built-in)

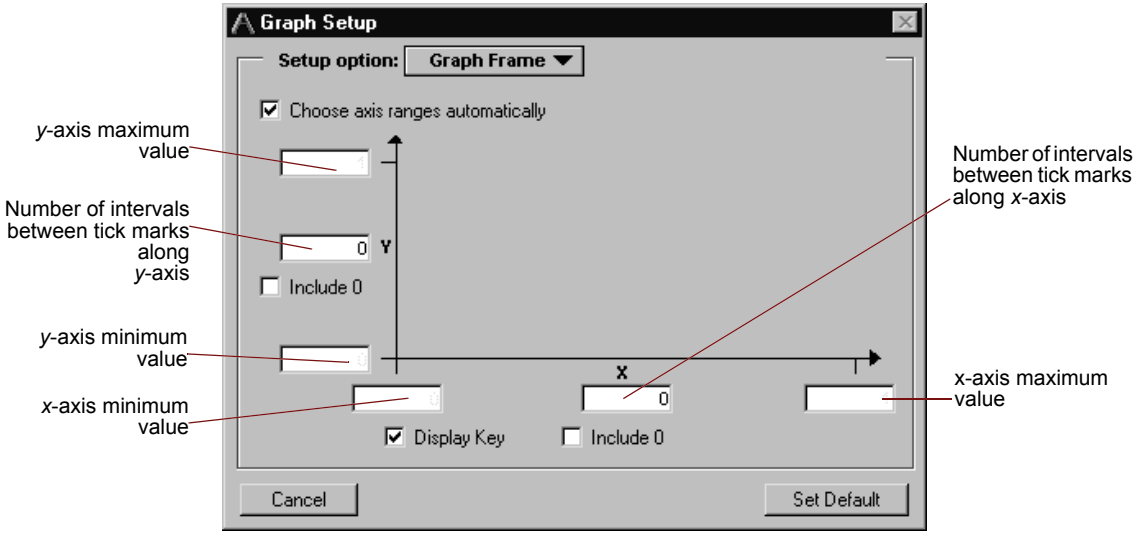
By default, Analytica's graphing tool is selected. If you previously had selected Excel Chart, use this option to choose Analytica's graphing tool.

Excel Chart®

Select this option to use Excel's graphing engine. Excel will launch when you select this icon, and will remain active as long as the graph is displayed. In future sessions, Excel graph will launch whenever a result graph for the evaluated Variable is set to use Excel graph. Once the graph is viewed, graph settings can be adjusted from the Excel window. Double click on the graph in Analytica to bring the Excel window to the foreground. See "Using Excel Graph with Analytica" on page 139

Graph Frame setup option

To change the graph frame in an Analytica graph, select the **Graph Frame** setup option from the popup menu.



Value entry boxes

Number of intervals between tick marks

If 0, Analytica chooses the number of intervals. If you enter a number, *n*, Analytica uses either *n* or *n*+1, depending on the minimum and maximum values.

Minimum/maximum value

After unchecking the Choose axis ranges automatically check box, you can enter the desired value.

Check boxes

Choose axis ranges automatically

Controls whether the ranges on the axes are set automatically. You must uncheck this box before you can edit the minimum and maximum fields for each axis. For bar graphs, you can change only the *y*-axis values.

You cannot uncheck this box to set defaults for all new graphs. You must uncheck it for *each* graph.

Display key

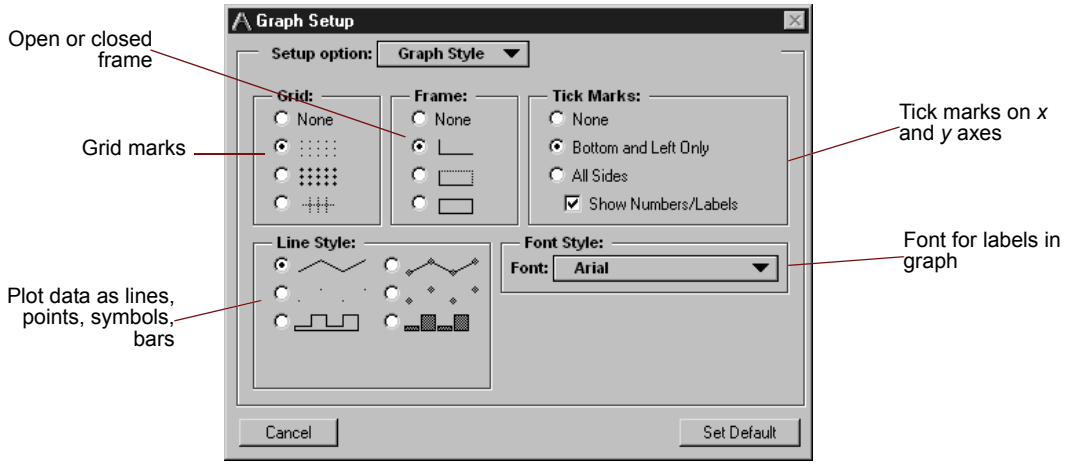
Shows the key (for a result of two or more dimensions).

Include 0

Includes 0 (the origin) on the given axis.

Graph Style setup option

To change the graph style in an Analytica graph, select the **Graph Style** setup option from the popup menu.



Grid

Controls whether a background grid displays, and if it is comprised of dots or lines.

Frame

Controls whether the graph displays the axes alone, or with a frame around the graph.

Tick marks

Controls how the tick marks appear along the axes.

None

Display no tick marks.

Bottom and Left Only

Display tick marks along the bottom and left hand axes.

All Sides

Display tick marks all around the frame.

Show Numbers/Labels

Display numbers or labels along the axes.

Line Style

Controls the style of the graph.

Line graph

Different line styles and colors are used for each key value.

Line and data markers

Different line styles, colors, and symbols are used for each key value. You can size the symbols.

Data markers (dots)

Useful with a large number of data points.

Data markers only

Different symbols are used for each key value; you can size the symbols.

Bar chart

Bars are of equal width and are center labeled on the horizontal axis. (Default for Probability Mass Function of a Probtable.)

Selecting bar chart will also force probability density function (PDF) or cumulative distribution function (CDF) views to treat the sample data as discrete, rather than continuous, and, therefore, will display a probability mass graph, rather than a probability density graph.

Overlap

Specifies the vertical spacing of bars within the same group. Positive values cause the bars within a group to overlap, negative values introduce space between bars of the same group.

Origin

Specifies the y-value for the base of the bars in the graph. The default is zero (*i.e.*, bars extend from the y-origin to the plotted value). The value in this field may be a number, or it may be an Analytica expression. If the result is dimensioned by the x-axis or key indexes, each bar can have its base positioned independently.

Histogram

Horizontal lines are plotted for each result point, positioned so that they extend to midway between the point and its neighbors on both sides. (This is the default style for probability density functions and cumulative distribution functions when viewed with equal x-axis steps.)

If your sample is numeric, and your Variable has no domain Attribute, selecting histogram will force the probability density function (PDF) and cumulative distribution function (CDF) views to treat the data as continuous.

Symbol Size

Enter the desired symbol size in points.
Minimum size: 4
Default size: 6
Maximum size: 36

Font Style

Sets the typeface for the graph. (Font size is determined by the window size and is adjusted when the window is resized.)

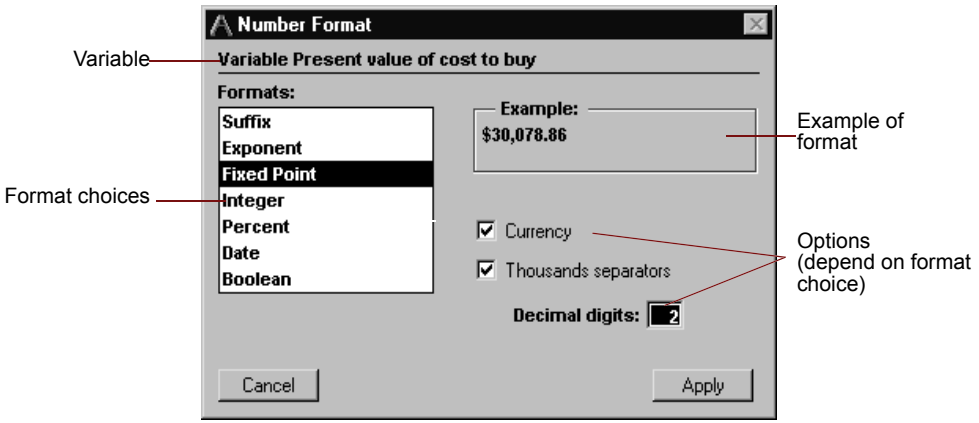
Number Format dialog box

Number formats can be specified for a table's contents, its row and column indexes, and for the *y* axis on a graph.

The number format for a Variable affects the display of all of its values everywhere they appear. For example, if you set the number format for an Index Variable in one Result window, the same number format is used if the Index Variable appears in another Result window.

To set the number format:

1. Open a Result window.
2. If the Result window is a table, select a row, column, or cell.
3. Choose **Number Format** from the **Result** menu or Ctrl-B to display the Number Format dialog box.



The top line shows the Variable to which the number format will be applied.

Formats

Choose from the following number formats:

Format	Description	Example
Suffix	the default (see the following table)	12.35K
Exponent	scientific exponential	1.235e04
Fixed Point	fixed decimal point	12345.68
Integer	fixed point with no decimals	12346
Percent	percentage	1234568%
Date	text date	12 Jan 93
Boolean	true or false	True

The suffix characters are:

Power of 10	Suffix	Prefix	Power of 10	Suffix	Prefix
3	K	Kilo	-2	%	percent
6	M	Mega or Million	-3	m	milli
9	G	Giga	-6	μ	micro (mu)

Power of 10	Suffix	Prefix	Power of 10	Suffix	Prefix
12	T	Tera or Trillion	-9	n	nano
15	Q	Quad	-12	p	pico
			-15	f	femto

Analytica Note: *If integer or fixed point is selected, a number larger than 10^9 displays in exponent format.*

Analytica for Windows uses the $ for a currency symbol, the comma for a thousands separators (‘,’), and a period (‘.’) for a decimal point.

Options

The options in the Number Format dialog box depend on the format selected.

The maximum number of digits or decimal digits is 15 (14 for fixed point and percent); the maximum number precision is 15 digits (9 for integers). The Suffix format shows a minimum of four significant digits.

Currency

Prepends a dollar sign (‘$’) when displaying numbers.

Decimal digits

If set to 1 or more, numbers are padded with zeros to fill out the specified number of digits after the decimal point.

Number of digits

For Exponent format, numbers are padded with zeros. For Suffix format, fewer digits can be displayed.

Date formats

These formats show a number as a date, computed as the number of days since January 1, 1904 (34,699 is January 1, 1999). The format used for the long and short formats can be set in the

Regional Setting Properties dialog box from the Windows control panel. If you select a format that includes the day of week, the day of week will be suppressed when data is copy and pasted or OLE linked to an external application (this allows applications such as spreadsheets to parse the dates).

Thousands separators

When selected, inserts commas between every third digit.

Analytica Note: *Number format affects how a definition is parsed, and affects how coersion from a number to text is performed. For example, if you set the Number format to "date", then you can enter a definition such as* `1/25/2005` *or* `25 Jan 2005`*. If the number format is not "date", then* `1/25/2005` *would be parsed as two successive divisions. If you have a coersion from a number to string, as in the expression* `Pi & ""`*, the resulting string will be created according to the number format for that node.*

Using multiple formats in a single result table

When viewing a result, the number format setting for a Variable applies to all values in the body of its result table. For most Variables, this is desirable since your Variables should contain only a single type of information (*e.g.*, only dollar amounts, or only percentages, but not both). However, occasionally you may want to bring together multiple pieces of information into a single "report," with each column (or row) formatted differently.

A report with multiple number formats can be created as follows:

1. Create a separate Variable for each column (or row) of the report. Set the number format for each of these Variables as you wish the corresponding column in the report to be formatted.
2. Create a node to represent the report. Define the node to be a list. In each cell of the list, type the identifier of the Variable containing the contents for that column or row of the report.
3. Display the result as a table.

When a Variable is defined as a list of identifiers, the result table uses the number format for the source Variable to format the column or row corresponding to that Variable. If the number format is

not set for the Variable corresponding to a column, the number format for the result being viewed is then used.

Using Excel Graph with Analytica

You may view result graphs using either Analytica's built-in graphing tool, or Microsoft Excel's graphing tool. When Excel Graph is used, Excel is launched and remains active in the background, while the graph itself appears within the Analytica result window. The use of Excel graph requires that Microsoft Excel 8 (or Microsoft Office 97), or later, be installed on your computer. Microsoft Excel is not included with Analytica.

It is generally more convenient to use Analytica's built-in graphing tool to display most results, especially during model development, since this avoids the overhead of launching Excel, uses less space in your model files, and makes it easier to switch between alternative result views. However, because Excel graph does offer additional graph types and formatting control not available in Analytica, utilizing an Excel graph can be very useful for creating highly specialized or presentation-quality graphs.

Switching a Result Graph to an Excel Graph

Starting with an Analytica result graph in view:

1. Double click on the graph.
2. When the Graph Setup box appears, select **Excel Graph** from the Setup option pull-down menu.
3. Click on the Excel Chart icon and click the **Apply** button.

Excel is launched, and the graph is drawn within the Analytica result window. To change graph settings, double click on the graph to bring Excel to the foreground, then use the menu options from Excel to select the graph type and formatting settings as desired.

Important notes about Excel Graph

Formatting settings

With an Excel Graph, various changes to the state of a result window can cause formatting settings to return to their default values or be lost. These modifications include:

- Certain changes in Variable's result values.
- Switching the graph view (*e.g.,* Probability Density Function view to Sample view).
- Altering axes order.

Any of these events will cause you to lose modifications you may have implemented to the series definitions (*i.e.,* X Values, Y Values or Series Names) since OLE will update your graphing data, thus overwriting your modifications.

There are other special cases when formatting settings can be reset. Switching the graph view or axes order may reset your axis labels. Also, if you change the graph view, the graph type is not preserved.

In general, the reformatting of graph settings is appropriate and should not be a significant issue. However, you may elect to link your model results directly into an Excel spreadsheet and then use the data.

Excel Graph as default will increase model file size

Using Excel Graph as the default is not recommended for medium to large size models because the overhead associated with Excel Graph will inflate the size of your model files (30KB per graph). Thus, when employing Excel Graph for a particular node you should choose the **Apply** button and not the **Set default** button from the Graph Setup dialog box.

In medium to large size models where you would like to have many nodes displaying their results using Excel Graph, it's wise to place 'copies' of these nodes in a saved module to save room. To exploit this strategy, do not make copies of the nodes using the **Copy** and **Paste** operations. Instead, create a new node with the same title and make its definition merely the identifier for the node of which you are making a copy.

Three-dimensional graphing

Excel Graph has the ability to display your model results as a three-dimensional surface. In these plots the "z" coordinate is the graph's key. To create three-dimensional plots of your model result when using Excel Graph, go to Excel's **Chart** menu, select **Chart Type** and choose the **Surface** option.

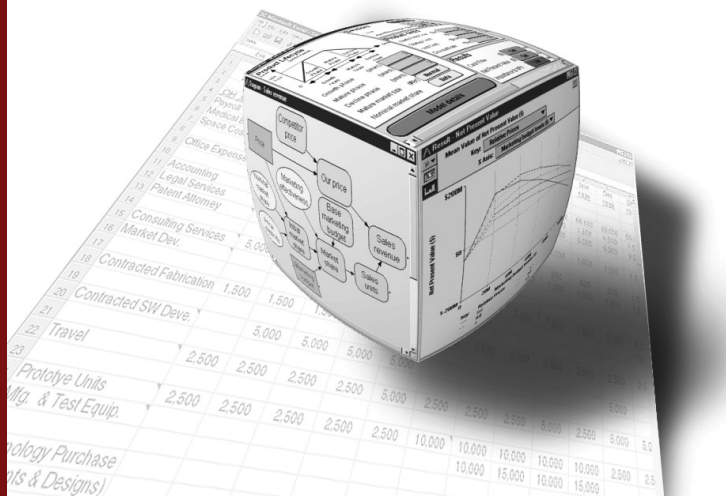
Keeping Excel open

You may want to avoid restarting Excel each time you graph Analytica model results with Excel Graph. When using Excel Graph from Analytica, Analytica will first try to use an existing instance of Excel, but if Excel is not running, Analytica will launch it. If Analytica launches Excel and only Analytica is using it, when the last result window containing an Excel Graph is closed, Analytica will close Excel.

To avoid restarting Excel repeatedly, launch Excel yourself before evaluating any nodes using Excel Graph. Analytica will use this instance of Excel, but will not close it.

Chapter 8

Creating and Editing Definitions



In this Chapter

This chapter shows you how to:

- Create definitions
- Edit definitions
- Use the Object Finder
- Check the validity of a Variable's value

8: Creating and editing definitions

This chapter introduces the tools for creating and editing powerful mathematical models by giving each Variable a formula that defines how to compute its value in its ***definition***. The definition of a Variable can be a simple number, text, a probability distribution, or a more complicated expression. It can also be a list or table of numbers or other expressions. Subsequent chapters present more details about using mathematical expressions, arrays, and probability distributions.

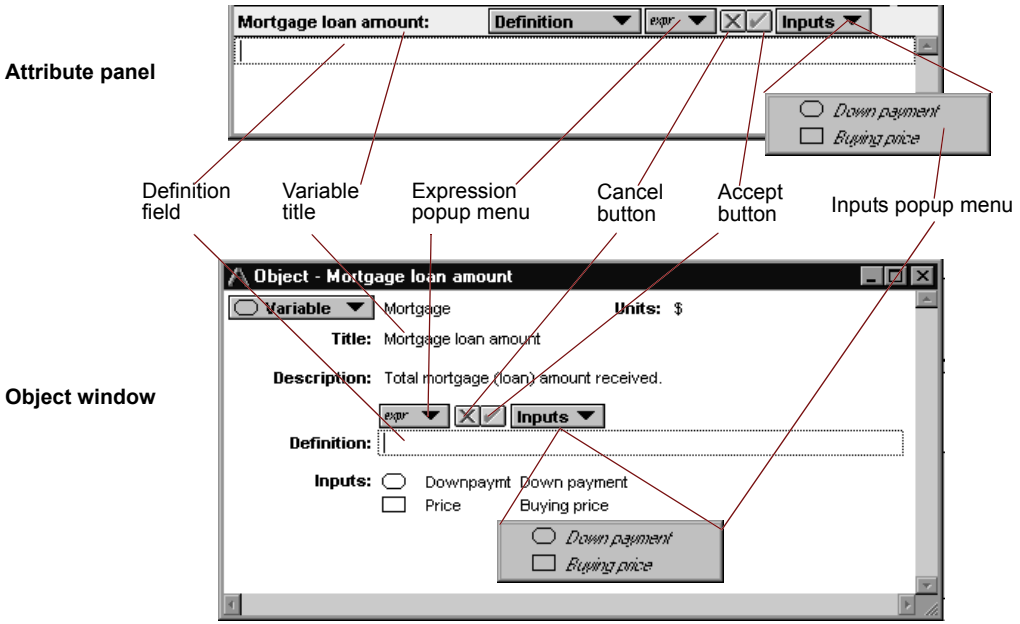
Creating or editing a definition

To create or edit the definition of a Variable, first be sure that the Edit tool () is selected. Select the Variable and do any of the following:

- Enter *Ctrl-E*.
- Click on **(**  **)** in the tool palette.
- Select **Edit Definition** from the **Definition** menu.
- Double-click on the Variable to open its Object window. Then click in the definition field.
- Click on the Key icon () to open the Attribute panel of the diagram. Select **Definition** from the Attribute popup menu. Then click in the definition field.

If the inputs to the Variable were specified by drawing arrows in the diagram, the definition initially looks like the illustration below. The definition field is blank and a popup menu listing the inputs to the Variable appears above the definition field. If the Variable has no inputs, the **Inputs** popup menu does not appear.

If you are editing in the attribute pane and you want to insert the identifier of a node *in the same diagram window*, you can click on the node while holding down the **ALT** key. This will insert the identifier of the node into the **attribute** field. It won't work if you want to click on something that is in a different window because the edit field looses focus before the mouse event reaches the node.



To edit a definition that is a simple number, text, or other expression:

1. Select the definition.
2. Edit it by typing, by deleting, or by using the standard text editing operators—that is, Copy (Ctrl-C), Cut (Ctrl-X), and Paste (Ctrl-V).

See Chapter 10, “Using Expressions”, for the syntax of numbers, operators, simple expressions, and mathematical functions.

You can change the definition to one of several commonly used expressions with the Expression popup menu (see “The Expression popup menu” on page 150).

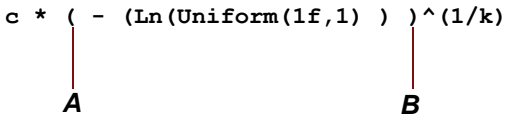
Special editing key combinations

A few special key combinations are quite useful when editing textual definitions. The arrow keys move one character or line at a time, and *Home* and *End* move to the beginning and end of the current line. Simultaneously depressing the *Ctrl* key with *Left* or *Right* moves to the beginning or end of the next word or identifier. If, in addition, the *Alt* key is depressed, the cursor is moved to the matching parenthesis when you are adjacent to a parenthesis. If

the *Shift* key is depressed during any of these cursor movements, the spanned text will be selected to be available for copy/paste operations, etc. A rapid method for selecting an identifier is to double click on the identifier using the mouse.

Parenthesis matching

Analytica expressions can often contain many levels of nested parentheses. The parenthesis matching keys make it easy to find the corresponding parenthesis in an expression. *Alt-Ctrl-Left* and *Alt-Ctrl-Right* moves from the outside of a parenthesis adjacent to the cursor to the outside of the corresponding parenthesis. For example, in the figure below, pressing *Alt-Ctrl-Right* when the cursor is at point *A* moves the cursor to point *B*. Subsequently pressing *Alt-Ctrl-Left* moves the cursor back to *A*.



Comments in definitions

It is wise to generously document your models. However, descriptions of Variables and algorithms are usually best placed in the **Description** Attribute and/or user-defined attributes for a Variable. However, comments intelligently embedded in a definition can also be very useful for improving readability of long expressions. Comments can also be used to disable portions of expressions while debugging.

Comments can occur at any point in a expression, provided they do not sever an identifier name. Comments begin and end with curly braces (*i.e.*, '{' and '}'), and may not be nested. During parsing and evaluation, everything between curly braces is ignored. Comments in the cells of an Edit Table are not preserved.

Identifiers

To refer to the value of another Variable, use its identifier. To place a Variable's identifier at the insertion point in the definition, do any of the following:

- If the Variable is an input, select it from the **Inputs** popup menu.

- Type in the Variable's identifier. To see all nodes in the active diagram labelled with their identifiers, select **Show By Identifier** from the **Object** menu (Ctrl-Y).
- Select **Paste Identifier** from the **Definition** menu and use the **Find** button or identifier menu items (see “Object Finder dialog box” on page 152).
- If the definition is being edited from the **Attribute** pane, you can insert the identifier of a variable in the same module window by holding down the *ALT* key and clicking on the node. The identifier of the clicked node will be inserted at the caret position. This shortcut isn't available from the **Object** Window or for nodes is different modules.

Functions

You can paste functions at the insertion point by doing either of the following:

- Select **Paste Identifier** from the **Definition** menu to open the Object Finder (see “Object Finder dialog box” on page 152).
- Select the function from its library in the **Definition** menu (see “Pasting from a library in the Definition menu” on page 155).

Automatic parentheses matching

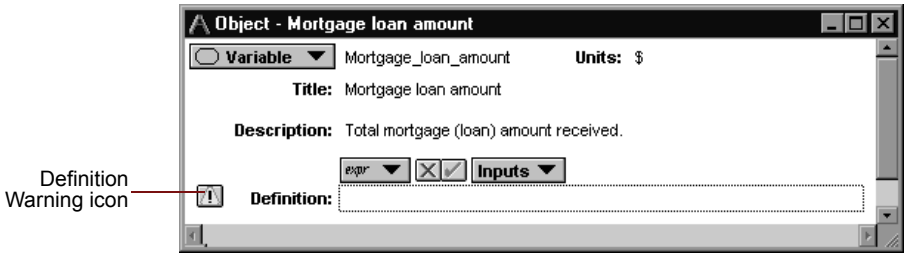
As you write or edit a definition, each time you enter a closing parens —parenthesis ")", square bracket "]", or comment bracket "}"—Analytica will show as **bold** it and its corresponding opening parens—"(", "[", or "{"—if there is one. This helps you see whether you have the right number and types of parentheses in complex expressions—without resorting to the dreaded parens counting.

Syntax check

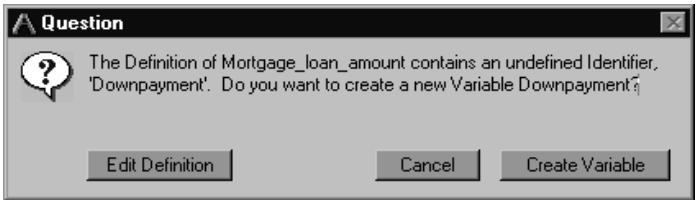
After entering or editing a definition, press Alt-*enter* or click on the accept button (☑) to perform a syntax check of the revised definition and accept the changes.

Click on the cancel button (☒) to cancel your changes.

The definition Warning icon (⚠) appears next to the definition if it is not syntactically correct. Click on the icon to see a message about what may be wrong.



A definition's syntax check may reveal syntax errors (see "Syntax error" on page 528). For example, if a definition contains text that is not an identifier, the following dialog box appears.



How a valid definition may change the diagram

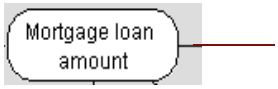
After you give a Variable a valid definition, the Influence Diagram containing that Variable might change.

Cross-hatching disappears

If the "Show Undefined" preference is selected (see "Preferences dialog box" on page 88), a node whose definition is missing or syntactically incorrect displays with a cross-hatch pattern.

For example:

After the definition is checked to be syntactically correct, the Variable's node in the Influence Diagram is clear.



Arrow updating

After checking syntax, Analytica makes sure the arrows going into this Variable (its inputs) properly reflect its definition.

- It draws an arrow from any other Variable mentioned in the definition.
- It removes an arrow from any Variable that is not mentioned in the definition.

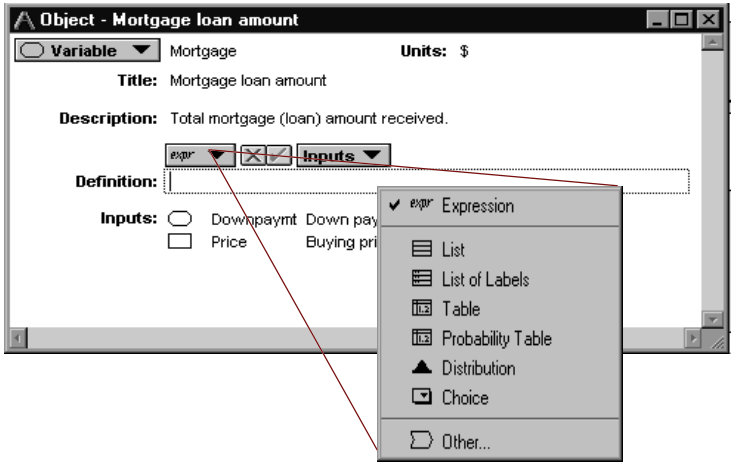
To avoid removing influence arrows while editing a definition, do not click on the check mark or press *Alt-Enter* to leave the definition. Instead:

- In the Object window, click on the close button.
- In the Attribute panel, select a different Attribute or select a different Variable in the diagram.

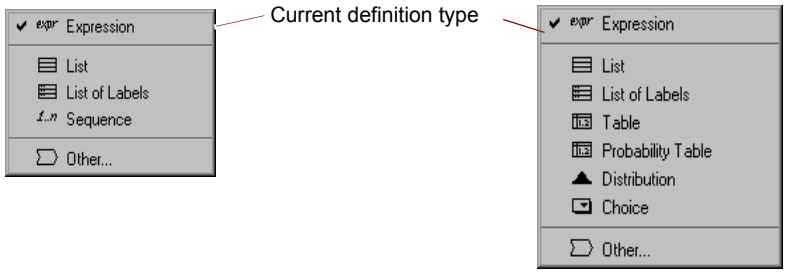
The Expression popup menu

Click on **expr** to see the Expression popup menu. The Expression popup menu shows the type of the definition, which is an empty expression in the following figure.

Expression popup menu



Use this popup menu to change the definition to one of several common kinds of expressions. The entries in this menu depend on the class of the node being defined.



Expression

Shows the definition as a mathematical expression, even if it was defined using the other expression types in this popup menu. See Chapter 10, “Using Expressions”.

List

Creates an ordered set of expressions or numbers. See “Creating an index” on page 215.

List of labels

Creates an ordered set of text labels. See “Creating an index” on page 215.

Sequence

Creates a list of numerical values. See "**`Sequence(Start,End, Stepsize)`**" on page 223.

Table

Creates an array of numbers or expressions. See “11: Arrays and indexes” on page 207.

Probability table

Creates an array defining probabilities (numbers or expressions) across the domain of a discrete (chance) Variable. See “Probability Tables” on page 304.

Distribution

Creates an uncertain definition by selecting a function from the Distribution system library. See “Defining a Variable as a distribution” on page 287.

Choice

Creates a popup menu for choosing one or all elements from a list. See “Creating a popup menu” on page 163.

Other

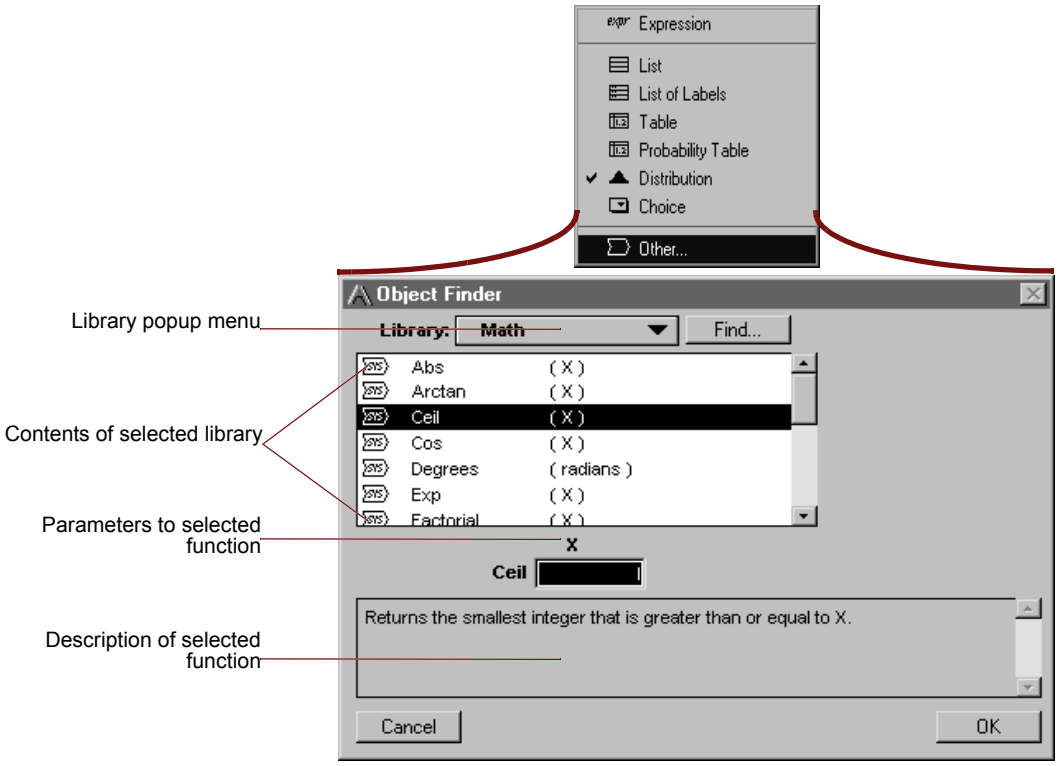
Opens the Object Finder dialog box, which is described in the next section. Changes the definition to the function or Variable that you select from the Object Finder.

Object Finder dialog box

Use the Object Finder dialog box to browse system functions, your own library functions, and all of a model’s identifiers and place any of these objects into a definition.

Open the Object Finder in either of the following ways:

- To insert the desired function or identifier at the insertion point in the definition, select **Paste Identifier** from the **Definition** menu.
- To replace the entire definition with the desired function or identifier, select **Other** from the Expression popup menu.

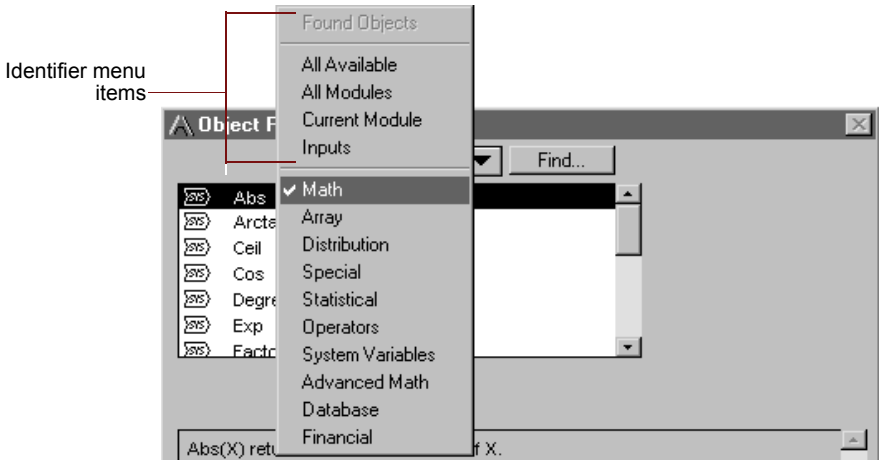


Use the **Library** popup menu to select a group of identifiers or a library.

- For identifiers, scroll to select:

Found Objects	Displays identifiers of objects found with the Find dialog. (See below.)
All Available	Displays all library functions and identifiers.
All Modules	Displays identifiers in all modules.
Current Module	Displays identifiers in the current module.
Inputs	Displays identifiers of the inputs to the selected node.

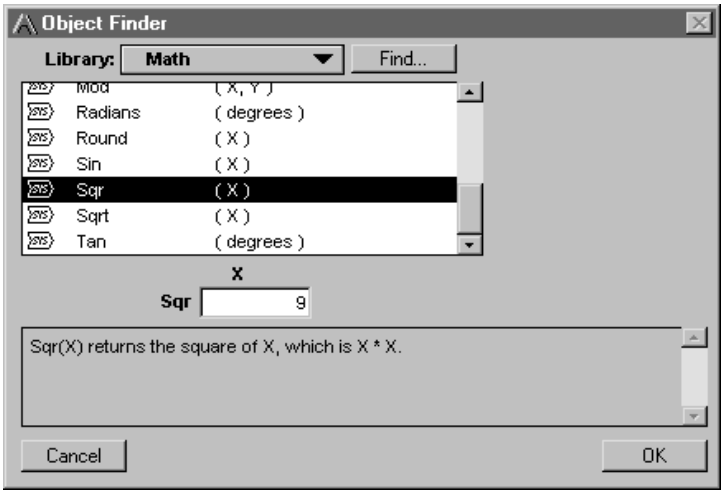
- For a library, the contents of the selected library are listed below the popup menu, showing the parameters if they are functions.



Use the **Find** button to search on the identifiers or titles of all Variables, modules, and functions.

Matching objects are listed in the Found Objects library.

To use a function, identifier, or system expression in a definition, select it. For a function, enter the required parameters in the parameter fields.



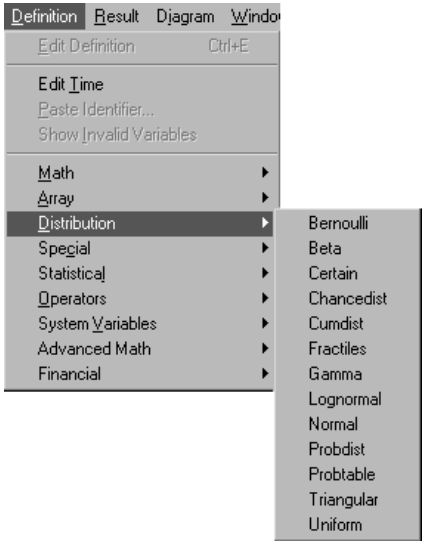
Click on **OK** to place the function, identifier, or expression in the definition.

Pasting from a library in the Definition menu

Use the **Definition** menu to quickly paste a function or system expression into a definition, when you do not need the Description of the function or expression.

While editing a definition:

1. Have the cursor at the point you want to insert a function or expression.
2. From the **Definition** menu, select the library and then the function or expression.



3. The function or expression is pasted into the definition.

4. Replace all parameters with input identifiers or expressions. Each parameter is enclosed in << >>. To replace it, select both << >> and its contents, then type or use the Inputs button or the **Paste Identifier** command to open the Object Finder dialog box.

Checking the validity of a Variable's values

You can create an automatic check on the validity of the value of a Variable using its Check Attribute. For example, to check that the value of Percent_damage is between 0 and 100, you give it a check of:

```
Percent_damage>=0 AND Percent_damage<=100
```

When the Variable is evaluated, and if the Check Attribute fails (evaluates to *False*), Analytica will give a warning and the opportunity to edit the definition.

There are two steps to using value checking:

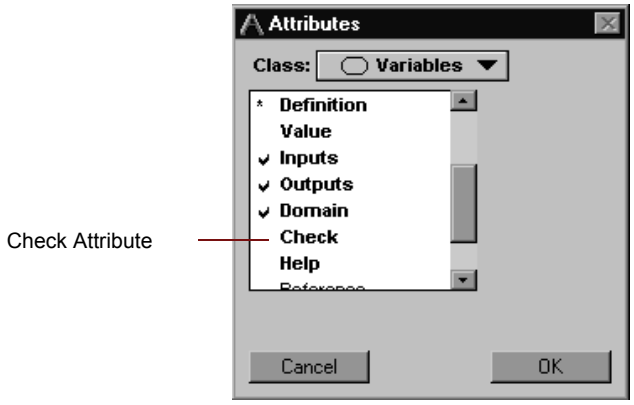
1. Display the Check Attribute.
2. Define checks for Variables.

Displaying the Check Attribute

If you want to use checking, first set the Check Attribute to be displayed in the Object window and Attribute view, since it is hidden by default.

To show the Check Attribute:

1. Select **Attributes** from the **Object** menu to open the Attributes dialog box. See "Managing attributes" on page 400.



2. Scroll down the Attribute list and find Check.
3. Click on Check once to select it, and a second time to add a check mark next to it. The check mark indicates that the Attribute is displayed in the Object window and in the Attribute popup menu.
4. Click on the **OK** button.

Now the Check Attribute appears in Object windows and in the Attribute popup menu in the Attribute panel below the diagram.

Defining the check

You can set the Check Attribute for any Variable. First open the Object window for the Variable, or show its Check Attribute in the Attribute view. Enter an expression in the Check Attribute field to constrain the value of a Variable. The expression should refer to this Variable by identifier or *Self*, and must be a Boolean (that is, evaluate to *True* or *False*). For example, to constrain the value for the lifetime of a car (*Lifetime*) to be greater than 0 and less than 12, define the check as:

Check: (Lifetime > 0) And (Lifetime < 12)

or

Check: (Self > 0) And (Self < 12)

If the check expression refers to another Variable, a dependency is created between the Variable being checked and the Variable included in the check expression. If the definition does not already refer to the other Variable, an arrow will be drawn between the two Variables.

Triggering a check

Analytica performs the check the first time it evaluates the checked Variable. Analytica evaluates a Variable the first time you ask to see its result, or the result of another Variable that depends on it. Analytica also performs the check on an input node immediately after you edit the input value (see "Using input nodes" on page 161).

If a check fails

If a check fails (the check evaluates to *False*), Analytica gives you the option of editing the Variable's definition, cancelling, or continuing. If you continue, the check will not be performed again unless you change the definition of the Variable or a Variable it depends on.

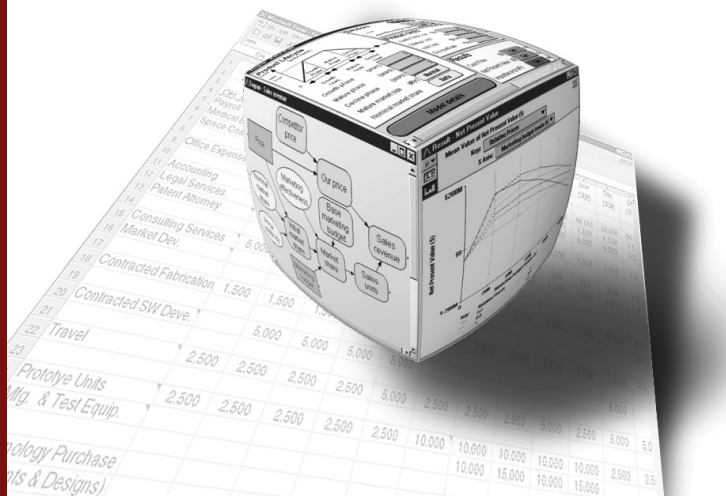
If you call the `Error()` function within the check, the message supplied as a parameter to `Error()` is displayed (rather than the default message produced by Check), and the same options are given.

Disabling value checking

You can disable all value checking by unchecking the **Check value bounds** check box in the Preferences dialog box (see page 88). This check box is checked by default.

Chapter 9

Creating Models Used by Others



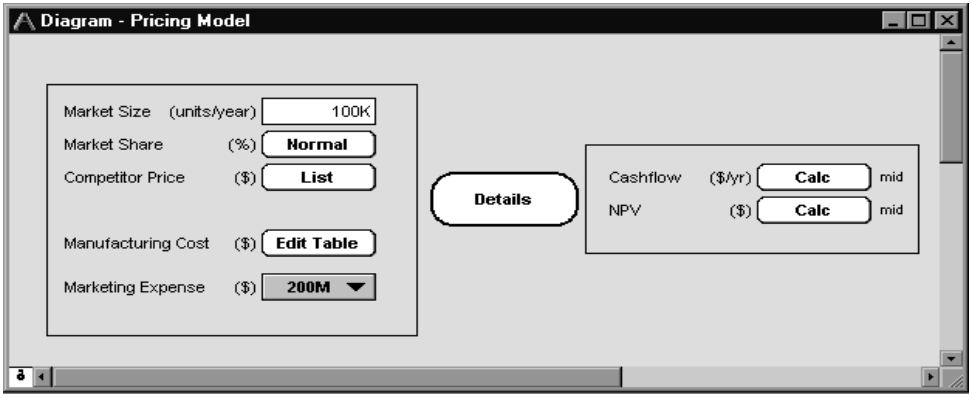
In this Chapter

This chapter shows you how to create a user interface for other users of your model.

9: Creating models used by others

You can use input and output nodes to create a simple user interface for other people who will use your Analytica model. Input nodes allow the user to see and change the values of Variables directly from Diagram windows. Similarly, with output nodes you can display selected output numbers in a diagram and open tables or graphs with a single click. Users of your model can then easily view and modify input Variables, and view the results, without navigating the details of the model, unless they wish to.

The diagram below contains input nodes on the left side and output nodes on the right side. The details of how the model computes the outputs from the inputs are available inside the "Details" module, for anyone who is interested.



Using input nodes

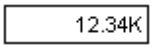
An ***input node*** lets you, or your end user, see and easily change the value of a Variable directly in the diagram, without opening an Attribute view or Object window (see “Browsing with input and output nodes” on page 26). In Browse mode you can change only the values and definitions of input nodes.

An input node is an alias of a Variable that you want to treat as an input to the model (see “Alias nodes” on page 84).

The type of definition of the original Variable determines the appearance of the input node (see “The Expression popup menu” on page 150). If you want your users to be able to change the

type of definition, instruct them on how to open an Attribute view or Object window and use the Expression popup menu.

Input field



A single number or text value (scalar) displays as an input field. You can have Analytica check if the input value is acceptable by using the check Attribute (see “Checking the validity of a Variable’s values” on page 156); the check is performed on input of a new value.

Input popup menu

A choice displays as an input popup menu. To create an input menu for an input node, see “Creating a popup menu” on page 163.

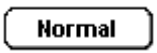
List

A list or list of labels displays as a **List** button (see “Creating an index” on page 215).

Edit table

An Edit Table displays as an **Edit Table** button (see “Viewing an array as an Edit table” on page 209).

Probability distribution



A probability distribution displays a button with the name of the distribution (see “13: Expressing uncertainty” on page 283).

Creating an input node

To create an input node from a Variable:

1. Select the Variable.
2. Select **Make Input Node** from the **Object** menu. The input node will appear in the same diagram next to the selected node.
3. Move the input node to the location you want.
4. Adjust the size of the node.

To make several input nodes at once, select the Variables and then choose **Make Input Node**.

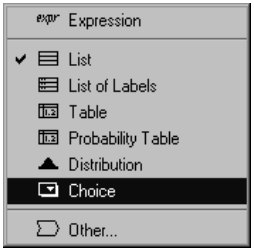
Creating a popup menu

For the classes of nodes that may be used for parametric analysis, such as decision and chance, the Expression popup menu includes the Choice option. The **Choice** option provides a way to offer the user a choice of selecting one or all values from a list.

Creating a menu from a list

If the original Variable is already defined as a list of numbers or labels, create a popup menu to select from the list as follows:

1. Show the definition of the Variable as a list, either in the Attribute view or the Object window.
2. Press the Expression popup menu and select the **Choice** option. Press OK to "Replace current definition with a Choice?"



3. The Object Finder dialog displays with parameter I=Self and n=0. Press OK.

The definition field of the original Variable now displays as a popup menu, and in browse mode, the input node displays as a popup menu. The original definition (list of numbers or labels) is now available as the ***domain*** of the Variable—the possible outcomes. In the expression view, the popup menu displays as the `Choice()` function (see page 256).

Analytica Note: *To define Var1 as a popup menu of another Variable Var2, that is defined as a list, select Choice from the Expression popup menu, and set the first parameter to I=*`Var2` *in the Object Finder dialog (see "*`Choice(I,n,inclAll)`*" on* page 256*).*

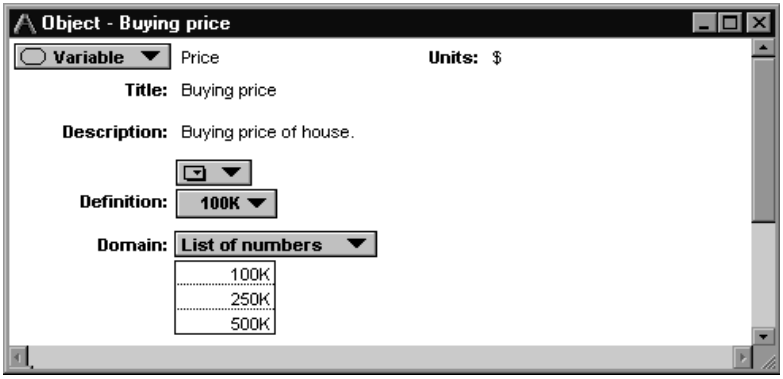
Analytica Note: *To hide the "All" option on the popup, enter inclAll=False as the third parameter in the Object Finder dialog.*

Creating a new definition

If a Variable has no previous definition, when you select **Choice** from the Expression popup menu, a domain (possible outcomes) of List of labels is created, with one element in the list.

To change the domain to List of numbers, press the Domain popup menu and select **List of numbers**.

Edit the list of values as you would edit a list of labels or list of numbers (see “Editing a list” on page 221). When you press *Alt-Enter*, the definition field becomes a popup menu of the domain values.



Analytica Note: *The values in the domain are evaluated deterministically.*

Using output nodes

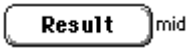
An ***output node*** gives you, or your end user, rapid access to a selected result in the model. You can use output nodes to focus attention on particular outputs of interest.

An output node displays a result value in the view style—table or graph, the indexes displayed, and the uncertainty view—last selected for display and saved with the model. It also shows the

uncertainty view icon (see “Uncertainty view options” on page 52).

61.73 mid

If the result is a single value (mid value or mean), it displays directly in the output field.



If the result is an array, the output node displays a Result button. Click on the button to display the table or graph.

After you display the table or graph, you can use the result tool palette to change the view.

If the value of an output has not yet been computed, the **Calc** button appears in the node. Click on the **Calc** button to compute and display the value.

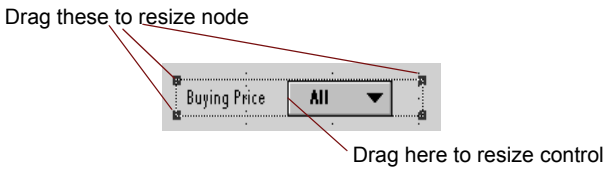
Creating an output node

To create an output node from a Variable:

1. In a diagram window, select the node of the Variable from which you wish to create an output node.
2. Select **Make Output Node** from the **Object** menu. The output node will appear in the diagram next to the selected node.
3. Move the output node to the location you want.
4. Adjust the size of the node.

The view style of the output result—table or graph—will be the format you last set for it (see “7: Formatting graphs and tables” on page 129).

Resizing controls



If you use a pull-down menu containing long text values, you may wish to widen the pull-down control as necessary to accommodate your longest text value. Input and output nodes contain text and graphics, in addition to the control itself. The node resizing handles that appear as small black squares at the corners of the node adjust the size of the bounding rectangle that holds all these

items, but does not change the width of the control itself. To change the width of a control (a pull-down menu, textedit box, or button), position the mouse over the left edge of the control, depress the mouse button and drag the mouse to the left or right.

Changing display style

The title and units of an input or output node are obtained from the original node. To edit them, edit the title and units of the original node (see “Editing Attributes” on page 85). If you edit the title or units of the original node, the input or output node's title or units changes to match the original.

By default, an input or output node shows its original node's title (label) in the original font, with no node outline or arrows. The node takes its color from its original node when the node is created. Later changes to the original node color do not change the color of the input or output node.

To change the appearance of an input or output node alone, use the **Set Node Style...** and **Show Color Palette** options from the **Diagram** menu (see “Node Style dialog box” on page 123 and “Changing background or node colors” on page 120). When you use these options to change the appearance of an input or output node, its original node does not change. Similarly, using these options to change the appearance of an original node does not affect its previously created input or output node.

Using form modules

It is often helpful to group input and output nodes into a single diagram for easy access by model users. The ***form module*** makes it easy for you to create input and output nodes in the form by drawing arrows between the form and Variables.

To create a form:

1. Make sure you are in a diagram window with the Edit tool selected.
2. Drag the module icon off the node palette and position it in the diagram.
3. Type in a title for the module—for example, Inputs.
4. Open the Attribute view at the bottom of the diagram window.

5. From the Attribute popup menu, select **Class**. A popup menu of available classes displays.

6. Select **Form** from the popup menu of classes.

Creating input and output nodes in a form module

An input or output node is an alias to another Variable in the model. Creating an input or output node is similar to creating an alias (see “Alias nodes” on page 82). To create a set of input and/or output nodes in the form module:

1. Adjust the diagram(s) on your screen so the form node and the source Variables for the input or output nodes are all visible (they can be in the same or different diagram windows).
2. In the tool palette, click on the arrow button ( ).
3. For input nodes, draw an arrow from the form node to each Variable. Analytica creates an input node for each Variable inside the form module.
4. For output nodes, select the Variables and draw arrows from the Variables to the form node. Analytica creates an output node for each selected Variable inside the form module.
5. When you have finished creating input and output nodes, double-click on the form node to open its diagram window.
6. In the tool palette, click on the edit button ( ).
7. Rearrange and resize the input and output nodes for clarity. It is usually clearest to put the input nodes down the left side and the output nodes down the right side.

A form module is like any other module, except when you draw arrows to or from the form module. So you can also create nodes that are not inputs or outputs and modules inside a form. If you have too many nodes to fit comfortably in a single diagram, you can create additional modules (which need not be forms) to enclose related groups of inputs and outputs.

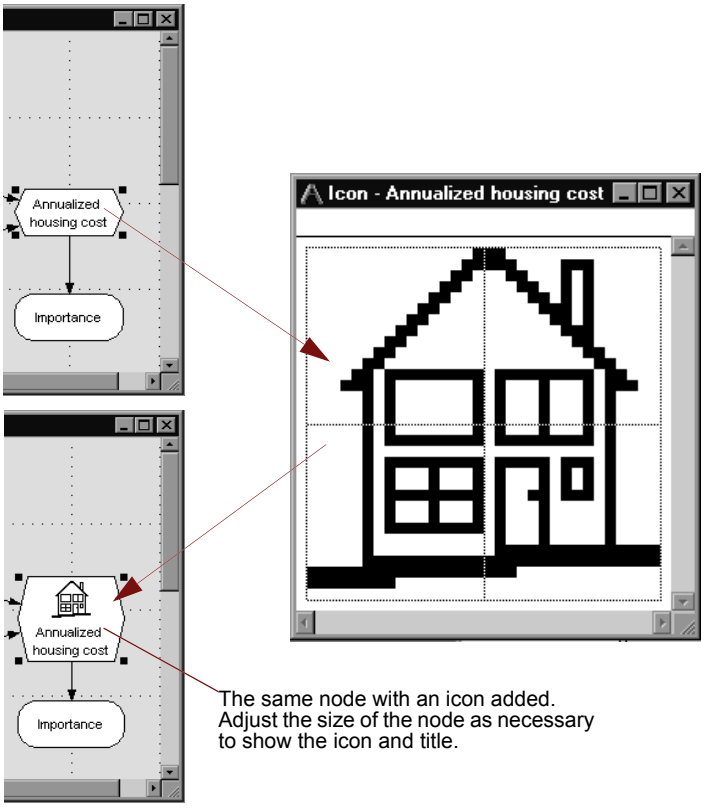
Adding icons to nodes

You can add an icon to any node in a diagram. The Icon window contains an enlarged space that you can use for creating or editing an icon.

Opening the Icon window

To add an icon:

1. Make sure that the Edit tool is selected.
2. Select the node that you wish to illustrate.
3. Choose **Edit Icon** from the **Diagram** menu to open the Icon window.



The same node with an icon added. Adjust the size of the node as necessary to show the icon and title.

Drawing or editing an icon

You can draw or edit the icon one pixel at a time using mouse clicks, or you can draw lines by holding down the mouse button as you drag the cursor.

- To make a dark pixel light or a light pixel dark, click on the pixel.
- To set the node's icon, click on the  button.
- To restore the original icon in the window (or to clear the window if there was no previous icon), click on the  button.

You can copy and paste an icon from one place in a model to another using the standard **Copy** (Ctrl-C) and **Paste** (Ctrl-V) commands.

Graphics, frames, and text in a diagram

Adding graphics

You can add a graphic image created in another application to any node or to the diagram background. Both color bitmaps and PICT graphics can be pasted in.

To paste in a graphic:

1. Copy (Ctrl-C) the graphic to the clipboard from within a graphics application.
2. Make sure that the Edit tool is selected in Analytica.
3. Select the node or the diagram window where you want the graphic to appear.
4. Paste (Ctrl-V) the graphic from the clipboard.

When you paste a graphic into the diagram window, a special node of class ***picture*** is created. Variable, module, and function nodes can be placed on top of picture nodes.

To remove a graphic, select it and press *Delete*, or choose **Clear** from the **Edit** menu.

Adding a frame

You can create a rectangular frame for nodes in a diagram in either of the following ways:

- Paste a graphic into the diagram window to create a picture node, then delete the graphic. This leaves a blank picture node. Use the Node Style dialog box (see “Node Style dialog box” on page 123) to display the border of the node. Other nodes can be placed on top of this node.
- Create a decision node and leave the title blank. Give it a definition of 0 (or any number) to remove the cross-hatch pattern. Use the Node Style dialog box (see “Node Style dialog box” on page 123) to hide the label and fill color. Create this frame first, then create the nodes to be framed and place them in the frame. If you create a framing decision node after you create the nodes to be framed, the nodes will be "under" the framing decision node; they will be visible, but you will not be able to select them.

Adding text

To add text to a diagram, drag a text node from the text button () on the toolbar to the diagram and enter the desired text. This creates a new node with a special class ***text.*** Use the handles to resize the node, and use the Node Style Dialog box (see “Node Style dialog box” on page 123) to change the font or to change the background from transparent to filled.

Models in XML file format

By default, Analytica 3.1 saves new models Analytica saves models in its own slot-filler format. The XML format lets you use a variety of applications that work with XML to read and edit the model files.

The format for saving models

Analytica 3.1 remembers which file format a model used and will save models in the same format. Hence, models created in earlier releases of Analytica for Windows or Macintosh will continue to use the old format. You can override that format by (un)checking **Save in XML Format** in the **Save as...** dialog selected from the **File** menu.

Compatibility with older releases

If you want to share models created in Analytica 3.1 with users who are using earlier releases, such as the Macintosh edition,

you should uncheck the **Save in XML Format** check box in the **Save as...** dialog. You will also need to avoid using any of the new syntax or functions introduced in Analytica 3.0 or 3.1 and described in this Upgrade Guide.

Sample old file format

Here is part of a sample model file in the old "slot filler" format:

```
{ From user Max Henrion, Model
Sample_old_file_format ~~
at Sep 1, 2003 3:56 PM}
Softwareversion 3.1.0

Model Sample_old_file_format
Title: Sample of old file format
Author: Max Henrion
Date: Sep 1, 2003 11:55 PM
Savedate: Sep 1, 2003 3:56 PM

Objective Net_income
Title: Net income
Units: $ millions
Definition: Revenues - Expenses
Nodelocation: 304,64,1

Variable Revenues
Title: Revenues
Units: $ millions
Definition: 700 * (1+ 0.10)^(Year - 2003)
Nodelocation: 176,32,1

Variable Expenses
Title: Expenses
Units: $ millions
Definition: Table(Year)(750,750,780,800,850)
Nodelocation: 176,96,1

Close Sample_old_file_format
```

Sample XML file format

Here is part of the same model, saved in the XML format:

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<ana user="Max" project="Sample_XML_file_format"
generated=" Sep 1, 2003 3:57 PM"
```

```
softwareversion="3.1.0" software="Analytica">

<model name="Sample_XML_file_format">
   <title>Sample XML file format</title>
   <author>Max Henrion</author>
   <date> Sep 01, 2003 11:55 AM</date>
   <saveauthor>Max Henrion</saveauthor>
   <savedate>Wed, Sep 1, 2003 3:57 PM</savedate>
   <fileinfo>0,Model Sample_XML_file_format,
    2,2,0,1, C:\Documents\Upgrade guide\Netincome
example XML.ANA </fileinfo>
<objective name="Net_income">
   <title>Net income</title>
   <units>$ millions</units>
   <definition>Revenues - Expenses</definition>
   <nodelocation>304,64,1</nodelocation>
   <nodesize>48,24</nodesize>
   <valuestate>2,313,273,197,250,0,MIDM
     </valuestate>
   <numberformat>1,D,4,2,0,1</numberformat>
   </objective>

   <Variable name="Revenues">
   <title>Revenues</title>
   <units>$ millions</units>
   <definition>700 * (1+ 0.10)^(Year - 2003)
</definition>
   <nodelocation>176,32,1</nodelocation>
   <nodesize>48,24</nodesize>
  </Variable>
<Variable name="Expenses">
   <title>Expenses</title>
   <units>$ millions</units>
  <definition>Table(Year)(750,750,780,800,850)
   </definition>
   <nodelocation>176,96,1</nodelocation>
   <nodesize>48,24</nodesize>
  </Variable>
</model>
</ana>
```

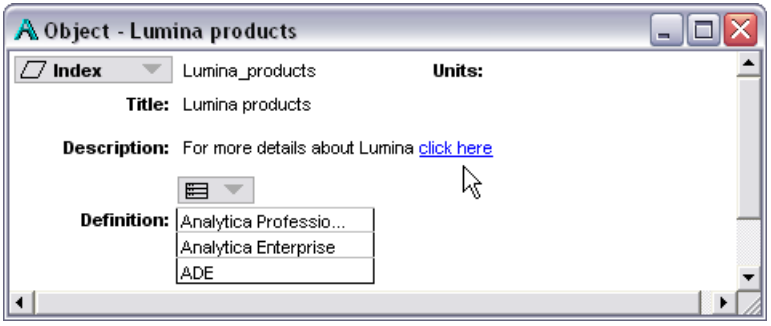
Hyperlinks in model documentation

Any Description, or other textual Attribute of a Variable or other Object, can now contain a hyperlink to any Web page. This is useful for linking to detailed explanations, data, or references for a model, or even to related downloadable Analytica models. In Browse mode, hyperlinks appear conventionally underlined in blue. When you click on a hyperlink, your computer will show the indicated web page in your default web browser.

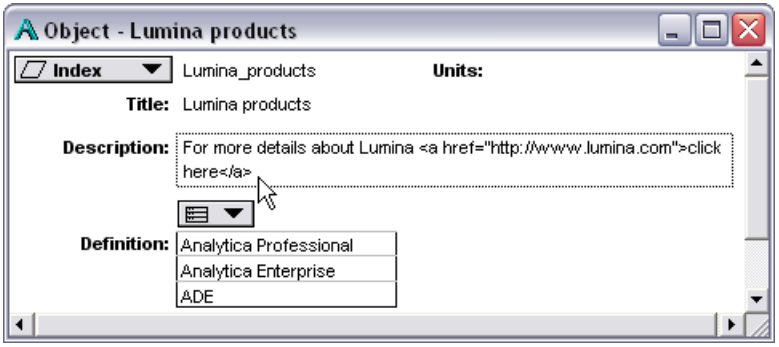
To define or edit a hyperlink, enter Edit mode, and use a standard HTML link syntax of the form

```
<a href="http://www.lumina.com">Click here</a>
```

In Browse Mode

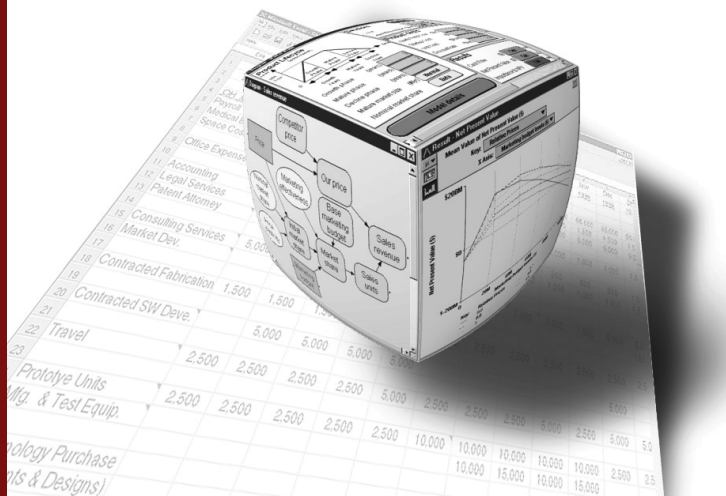


In Edit Mode



Chapter 10

Using Expressions



In this Chapter

This chapter tells you how to

- Write values, including numbers, Booleans, and text values
- Write expressions using arithmetic, logical, and comparison operations, and functions
- Select common functions that operate on numbers and text values

10: Expressions

This chapter describes the building blocks for creating and editing expressions to define Variables: numbers, operators and mathematical functions.

Numbers

The following formats are all valid for entering numbers:

Number Format	Examples
Integers	2, 10, 1234
Decimals	32.5, .0002, 0.000012345
Suffix	250K, 10.5M, 10.5m, 22%
Exponential form	53E11, 1E20, 4.5632E-25

- The signed integer after the E is an exponent that denotes a power of ten. For example:

  $5E4 = 5 \times 10^4 = 50{,}000$

  $4.3E\text{-}3 = 4.3 \times 10^{-3} = 0.0043$

- A character suffix denoting a power of ten is a convenient way to express very large or small numbers. For example:

  $50K \rightarrow 50{,}000$

  $1.5m \rightarrow 0.0015$

  The character suffixes are the same as used in the default output number format (see the table on page 136).

Power of 10	Suffix	Prefix	Power of 10	Suffix	Prefix
3	K	Kilo	-2	%	percent
6	M	Mega or Million	-3	m	milli
9	B	Billion	-6	μ	micro (mu)
9	G	Giga	-9	n	nano

Power of 10	Suffix	Prefix	Power of 10	Suffix	Prefix
12	T	Tera or Trillion	-12	p	pico
15	Q	Quad	-15	f	femto

Analytica Note: *The character suffixes m (10^{-3}) and M (10^{6}) are distinct. This is the only situation in which the case of a letter makes any difference for input to Analytica. For example, you can use k and K interchangeably.*

Range

Analytica can represent numbers between 10^{-308} and $9 \cdot 10^{+307}$.

Numbers out of range

When a calculation results in a number whose absolute value is less than the smallest number that can be represented, Analytica rounds the number to 0 (zero) without warning. For example:

```
1/10^1000 → 0
```

INF (infinity)

When a calculation results in a number whose absolute value is greater than the largest that can be represented, Analytica displays it as INF or -INF, for positive or negative infinity. For example:

```
10^1000 → INF
-10^1000 → -INF
1/0 → INF
```

You can enter INF as a value in an expression. Analytica can perform some computations with INF, such as:

```
INF + 10 → INF
INF/0 → INF
10 - INF → -INF
```

Other computations with INF, such as difference and ratio, give results that are ill-defined and return NAN (Not A Number):

```
INF - INF → NAN
INF/INF → NAN
```

A NAN may be detected in an expression using the **IsNaN()** function. See page 200.

Precision

The maximum internal precision of numbers is 15 significant digits.

Some calculations, especially those that involve small differences between numbers, may result in less precision than the maximum.

Text values

You can specify a text value by enclosing text in single quotes, or in double quotes, for example:

```
'A', "A25", 'A longish text - with punct.'
```

A text value can contain any character, including comma, space, and new line. To include a single quote(') or apostrophe, you can type two single quotes in sequence, such as:

```
'Isn''t this easy?'
```

The resulting text will contain only one apostrophe character. Or you can enclose the text value in double quotes:

```
"Don't do that!"
```

Similarly, if you want to include double quotes, enclose the text in single quotes:

```
'Did you say "Yes"?'
```

You can enter a text value directly as the value of a Variable, or in an expression, including as an element of a list (see “Creating an index” on page 215 and “List vs. list of labels” on page 219) or Edit Table (see “Creating an array with an Edit Table” on page 225). Analytica displays text values in results without the enclosing quotes.

Also see “Text functions” on page 194.

Boolean or logical values

There are two ***Boolean*** or ***logical*** values—*True* and *False*. You can specify a Boolean value in an expression as *False* or *True*, or, equivalently, as the numbers, 0 or 1. For example:

```
False or True → True
1 And 0 → False
```

Analytica treats every nonzero number as *True*. For example:

```
2 And True → True
```

Analytica displays Boolean results as 0 or 1, by default. To display them as *False* or *True*, change the format of the definition or result to Boolean (see “Number Format dialog box” on page 135).

Operators

An ***operator*** is a symbol, such as a plus sign (+), that represents a computational operation or action such as addition or comparison. Analytica includes the following sets of standard operators.

Arithmetic operators

The arithmetic operators apply to numbers and produce numbers.

Operator	Meaning	Examples
+	plus	3+2 → 5
-	minus	3-2 → 1
*	multiplied by	3*2 → 6
/, ÷	divided by	3/2 (= $\frac{3}{2}$) → 1.5
^	to the power of	3^2 (= 3^2) → 9
^*fraction*	root (fractional exponent)	4^.5 (= $4^{\frac{1}{2}}$) → 2

Comparison operators

The comparison operators apply to numbers and text values and produce Boolean values.

Operator	Meaning	Examples (1 = true, 0 = false)	
<	less than	`2<2`	→ 0
		`'A'<'B'`	→ 1
<=, ≤	less than or equal to	`2<=2`	→ 1
		`'ab'<='ab'`	→ 1
=	equal to	`100=101`	→ 0
		`'AB'='ab'`	→ 0
>=, ≥	greater than or equal to	`100>=1`	→ 1
		`'ab'>='cd'`	→ 0
>	greater than	`1>2`	→ 0
		`'A'>'a'`	→ 1
<>, ≠	not equal to	`1<>2`	→ 1
		`'A'<>'B'`	→ 1

Alphabetic ordering of text values

The comparison operators, >, >=, >=, and <, compare the alphabetic ordering based on ASCII coding of two text values. For example,

`'Analytica' < 'Excel' → 1 (true)`

Using the numerical (ASCII) representation of the characters, means:

a. Digits precede (are smaller than) letters, so

`'9' < 'A' ‡ 1 (True)`

b. Uppercase letters precede lowercase letters. If you want to alphabetize without regard to case, first use **`TextUppercase`** (or **`TextLowerCase`**) to convert all letters to the same case.

`'Analytica' > 'excel' → 0 (False)`

`TextUpperCase('Analytica') <`

```
TextUpperCase('excel') → 1 (True)
```

c. Letters with accents, umlauts, cedillas, ligatures, and other decoration come after undecorated letters, hence alphabetic ordering may be different from what you expect.

Sortindex(*a, i*) sorts text values in ***a*** using the same ordering scheme. But, **Rank(d)** works only on numerical values, and does not rank text values.

Logical operators

The logical operators apply to Boolean values and produce Boolean values.

Operator	Meaning	Examples (1 = true, 0 = false)
b1 **AND** *b2*	true if both *b1* and *b2* are true, otherwise false	`1 AND 20<2` → 0
b1 **OR** *b2*	true if *b1* or *b2* or both are true, otherwise false	`0 OR 1<2` → 1
NOT *b*	true if *b* is false, otherwise false	`NOT (2<3)` → 0

Scoping operator (::)

Later versions of Analytica introduced many functions that did not exist in previous releases of Analytica. Some models created in previous releases may contain Variables or user-defined functions with the same name as new built-in functions. In this situation, an identifier name appearing in an expression may be ambiguous.

For example, suppose a model written in Analytica 1.2 contains a user-defined function named `Irr`. If a Variable in this model uses the `Irr` function, the `::` operator is prepended to, or omitted from, the identifier name in order to disambiguate whether the definition refers to the user-defined `Irr` function, or the built-in `Irr` function.

Prepending `::` to the name of a built-in function causes the reference to always refer to the built-in function when there is an ambiguity. Otherwise, the identifier will refer to the user's Variable or function. With this convention, existing models are not changed by the introduction of new built-in functions.

Example

Suppose a model from an older release of Analytica contains the user-defined function `Irr( Values,I )`. Then

`Irr( Payments,Time )`	User's `Irr` function
`::Irr( Payments,Time )`	The built-in function

Operator binding precedence

A precedence hierarchy resolves potential ambiguity when evaluating operators and expressions. The hierarchy of precedence for operators, from most tightly bound to least tightly bound is:

```
functions, not
^
- (unary)
*, /
+, -
<, >, <=, >=, =, <>
and, or
If ... Then ... Else
```

Within each level of this hierarchy, the operators bind from left to right (left associative).

Examples

The following arithmetic expression:

```
1 / 2 * 3 - 3 ^ 2 + 4
```

is interpreted as:

```
((1 / 2) * 3) - (3 ^ 2) + 4
```

The following logical (Boolean) expression:

```
If a and b > c or d + e < f ^ g Then x Else y + z
```

is interpreted as:

```
If ((a and (b > c)) or ((d + e) < (f ^ g))) Then x
Else (y + z)
```

Conditional operators

The conditional operators are:

```
If B then U else V
Ifonly B then U else V
Ifall B then U else V.
```

All three conditional operators return elements of *U* and *V* depending on the value of *B*. The three operators differ on whether *U* and *V* are evaluated when *B* is constant, and on what the final result is indexed by. The following table summarizes what gets evaluated in each case:

What gets evaluated:

	B contains only True	B contains only False	B contains both True and False
`If`	B, U	B, V	B, U, V
`Ifonly`	B, U	B, V	B, U, V
`Ifall`	B, U, V	B, U, V	B, U, V

What the final result is indexed by (*B* in the table indicates that the result is indexed by the dimensions of *B*, etc.):

	B contains only True	B contains only False	B contains both True and False
`If`	B, U	B, V	B, U, V
`Ifonly`	U	V	B, U, V
`Ifall`	B, U, V	B, U, V	B, U, V

If *B* Then *U* Else *V*

Returns *U* or *V*, or an array whose cells contain values of *U* or *V*, depending on the value of *B*.

- If *B* has the value *True* (or any nonzero number), it returns the value of *U* and does not evaluate *V*.
- If *B* has the value *False* (or 0), it returns the value of *V* and does not evaluate *U*.
- If *B* is an array which contains at least one *True* (nonzero) value and at least one *False* (0) value, it evaluates both *U* and *V*. It returns an array indexed by the union of the indexes of *B*, *U*, and *V*, containing elements from *U* or *V* according to the corresponding elements of *B*.
- If *B* is an array containing only *True* (only non-zero numbers), *U* is evaluated, *V* is not evaluated, and the result is indexed by the indexes of *B* and *U*.

- If *B* is an array containing only *False* (only zeros), *U* is not evaluated, *V* is evaluated, and the result is indexed by the indexes of *B* and *V*.

Examples

N:

1	2	3

```
If N > 0 Then 'Yes' Else 'No' →
```

N ▶

	1	2	3
	'Yes'	'Yes'	'Yes'

```
If N = 2 Then 'Yes' Else 'No' →
```

N ▶

	1	2	3
	'No'	'Yes'	'No'

```
If N < 0 Then 'Yes' Else 'No' → 'No'
```

Avoiding evaluation

You may want to avoid evaluation of *U* for elements of *B* that give undefined results. For example:

Myarray:
In2 ▶

	21	22	23
	-10	0	10

If `Myarray > 0 Then Ln(Myarray) Else 0` gives a warning message on evaluating `Ln(-10)`. Ignoring the message gives

In2 ▶

	21	22	23
	0	0	2.303

To avoid evaluation of *U* for the elements that are false, evaluate `If...Then...Else` on each element of *Myarray* using a `For...Do` loop (see page 443) or `Using..In..Do` (see page 443).

```
Using Temp := Myarray in In2 Do
    If Temp > 0 Then Ln(Temp) Else 0
```

Ifonly *B* Then *U* Else *V*

Ifonly is similar to **If**, except that it does not include the dimensions of *B* in the result if *B* is constant. Like **If**, the **Else** part is optional.

- If *B* has the value *True* (or any nonzero number), it returns the value of *U* and does not evaluate *V*.
- If *B* has the value *False* (or 0), it returns the value of *V* and does not evaluate *U*.
- If *B* is an array which contains at least one *True* (nonzero) value and at least one *False* (0) value, it evaluates both *U* and *V*. It returns an array indexed by the union of the indexes of *B*, *U*, and *V*, containing elements from *U* or *V* according to the corresponding elements of *B*.
- If *B* is an array containing only *True* (only non-zero numbers), *U* is evaluated, *V* is not evaluated, and *U* is returned. Unlike **If**, The dimensions of *B* are not included in the result.
- If *B* is an array containing only *False* (only zeros), *U* is not evaluated, *V* is evaluated, and *V* is returned. Unlike **If**, The dimensions of *B* are not included in the result.

Analytica Note: *Omitting* **else** *should only be used when the statement is followed by a semi-colon with another expression following, as in:*

```
Var A := Min([X,Y]);
If A<0 Then A:=0;
Sqrt(A)
```

When to use

The main difference between **Ifonly** and **If** is that **Ifonly** collapses the array dimensions when *B* is constant. In general, **Ifonly** can cause confusion when dimensions disappear simply because numbers (in *B*) come out equal in a coincidental situation. However, if your intention is to reduce the dimensionality of the result when *B* is constant, then use **Ifonly**.

In the world of array abstraction, one can consider an array that is not indexed by *I* to be equivalent to an array that is constant across *I*, with each slice along *I* being equal to the original array. In this sense, **If** and **Ifonly** return equivalent results. If dimensions are re-introduced later, the downstream results will be the same in the two cases. However, since the dimensionality is smaller for results of **Ifonly**, there can be a slight computational

advantage over `If`. Due to the use of sparse array representations inside Analytica's engine, the difference in computational advantage is usually very small.

Examples

N:

1	2	3

```
Ifonly N > 0 Then 'Yes' Else 'No' → 'Yes'
Ifonly N = 'A' Then 'Yes' Else 'No' → 'No'
Ifonly N = 2 Then 'Yes' Else 'No' →
```

N ▶

	1	2	3
	'No'	'Yes'	'No'

Ifall *B* Then *U* Else *V*

`Ifall` is similar to `If`, except that it always evaluates both *U* and *V*, and returns a value whose dimensions are always the same—the union of the indexes of *B*, *U*, and *V*. `If`, `Ifonly`, and `Ifall` give the same result when *B* is an array whose values are partially true. While the `else` clause is optional for `If...then...else`, it is not optional for the `Ifall...then...else`.

When to use

When *B* is an array, the number and identity of the dimensions of the result of an `If` expression can vary according to the values in *B*. Use `Ifall` instead of `If` to ensure that the dimensions of the result are always the same.

Examples

N:

1	2	3

```
Ifall N > 0 Then 'Yes' Else 'No' →
```

N ▶

	1	2	3
	'Yes'	'Yes'	'Yes'

```
Ifall N = 2 Then 'Yes' Else 'No' →
```

N ▶

	1	2	3
	'No'	'Yes'	'No'

```
Ifall N = 'A' Then 'Yes' Else 'No'→
```

N ▶

	1	2	3
	'No'	'No'	'No'

Functions

Analytica provides a large number of built-in functions for performing mathematical, array, statistical, textual, and financial computations. There are also probability distribution functions for uncertainty and sensitivity analysis. The Enterprise edition of Analytica also includes functions for accessing external ODBC data sources. Finally, you can write and use your own user-defined functions.

Calls to Analytica functions have the form:

```
FunctionName( param1, param2, ... )
```

In other words, the function name followed by a comma-delimited list of parameters. Parameters can themselves be expressions built out of constants, Variable names, operators, and functions. Here are some simple examples of expressions involving functions.

```
Exp(1) → 2.718281828459
Sqrt(3^2 + 4^2) → 5
Round(2*Pi) → 6
Mod( X, 3 ) → 1      where X → 7
Pmt( 8%, 30, -1000) → $88.83
N * Sum( w*w, J )
Normal(500,100)
```

Functions are described in the chapters that follow. The rest of this chapter describes the basic (non-array) functions.

Example data

The examples in this chapter refer to the following Variables:

Car_type:

	VW	Honda	BMW

Years:

1985	1986	1987	1988

Mpg:

26	30	35

Time:

0	1	2	3	4

Cost: Mpg ▼, Car_type ▶

	VW	Honda	BMW
26	2185	2810	3435
30	1705	2330	2955
35	1585	2210	2835

Car_prices: Car_type ▼, Years ▶

	1985	1986	1987	1988
VW	8000	9000	9500	10K
Honda	12K	13K	14K	14.5K
BMW	18K	20K	21K	22K

Cost_in_time: Mpg ▼, Time ▶, Car_type = VW

	0	1	2	3	4
26	2185	2294	2409	2529	2656
30	2810	2951	3098	3253	3416
35	3435	3607	3787	3976	4175

Cost_in_time: Mpg ▼, Time ▶, Car_type = Honda

	0	1	2	3	4
26	2385	2314	2529	2649	2856
30	2910	3041	3238	3343	3526
35	3535	3847	3897	4166	4365

Cost_in_time: Mpg ▼, Time ▶, Car_type = BMW

	0	1	2	3	4
26	3185	3294	3409	3529	3656
30	3810	3951	4098	4253	4416
35	4435	4607	4787	4976	5175

Math functions

These functions can be accessed under the **Definition** menu **Math** command, or in the Object Finder dialog box, **Math** library. Additional math functions that are highly specialized or less common are also available in the **Advanced Math** library described in “Function List” on page 537.

Abs (*x*)

Returns the absolute value of *X*.

```
Abs(180) → 180
Abs(-210) → 210
```

Arctan (*x*)

Returns the arctangent of *X* in degrees.

```
Arctan(0) → 0
Arctan(1) → 45
Arctan(Tan(45)) → 45
```

See also `Arctan2` in the Advanced Math functions on page 193.

Ceil(*x*)

Returns the smallest integer that is greater than or equal to X.

```
Ceil(3.1) → 4          Ceil(5) → 5
Ceil(-2.9999) → -2     Ceil(-7) → -7
```

Cos (*x*)

Returns the cosine of *X, X* assumed in degrees.

```
Cos(180) → -1
Cos(-210) → -0.866
```

Degrees(*R*)

Converts from radians to degrees.

```
Degrees( Pi/2 ) → 90
Degrees( -Pi ) → -180
```

Exp (*x*)

Returns the exponential of *X*—that is, **ex**. *X* must not be greater than 709.

```
Exp(5) → 148.4
Exp(-4) → 0.01832
```

Factorial (*x*)

Returns the factorial of *X*, which must be between 0 and 170.

```
Factorial(5) → 120
Factorial(0) → 1
```

If *X* is not an integer, *X* is rounded to the nearest integer before taking the factorial.

Floor(*x*)

Returns the largest integer that is smaller than or equal to X.

```
Floor( 2.999 ) → 2      Floor(3) → 3
Floor(-2.01 ) → -3      Floor(-5) → -5
```

Ln (*x*)

Returns the natural logarithm of *X,* which must be positive.

```
Ln(150) → 5.011
Ln(Exp(5)) → 5
```

Logten (*x*)

Returns the logarithm to the base 10 of *X,* which must be positive.

```
Logten(180) → 2.255
Logten(10 ^ 30) → 30
```

Mod(*X, Y*)

Returns the remainder (modulus) of X/Y.

```
Mod(7,3) → 1            Mod(12,4) → 0
Mod(-14,5) → -4
```

Radians(*D*)

Converts from degrees to radians.

```
Radians(-90) → -1.57079633
Radians(180) → 3.141592654
```

Round (*x*)

Returns the value of *X* rounded to the nearest integer.

```
Round(1.8) → 2        Round(-2.8) → -3
Round(1.499) → 1      Round(-2.499) → -2
```

Sin (*x*)

Returns the sine of *X*, *X* assumed in degrees.

```
Sin(30) → 0.5
Sin(-45) → -0.7071
```

Sqr (*x*)

Returns the square of *X*.

```
Sqr(5) → 25
Sqr(-4) → 16
```

Sqrt (*x*)

Returns the square root of *X*, which must be positive or zero.

```
Sqrt(25) → 5
```

Tan(*x*)

Returns the tangent of *X*, *X* assumed in degrees.

```
Tan(45) → 1
```

Advanced math functions

These functions can be accessed under the **Definition** menu **Advanced Math** command, or in the Object Finder dialog box, **Advanced Math** library. Functions in this section are generally for

more advanced mathematical users than those found in “Math functions” on page 190.

Arccos(*x*)

Returns the inverse cosine of *X*, where *X* is between 0 and 1. The result is in degrees, between 0 and 180.

```
Arccos(1) → 0
Arccos(Cos(45)) → 45
```

Arcsin(*x*)

Returns the inverse sine of *X*, where *X* is between 0 and 1. the result is in degrees, between -90 and 90.

```
Arcsin(1) → 90
Arcsin(Sin(45)) → 45
```

Arctan2(*Y, X*)

Returns the arctangent of *Y/X* without loosing information about which quadrant the point is in. The result is the angle (in degrees) between the *X* axis and the point (*X,Y*) in the two dimensional plane, in the range (-180,180]. If `Y=X=0`, returns zero.

```
Arctan2( -1,1 ) → -45
ArcTan2( 0,-1 ) → 180
```

Cosh(*x*)

The hyperbolic cosine of *X*, *X* assumed to be in degrees.

Lgamma(*x*)

Returns the Log Gamma function of *X*. Without numerical overflow, this function is exactly equivalent to `ln(GammaFn(X))`. Because the gamma function grows so rapidly, it is often much more convenient to use LGamma() to avoid numeric overflow.

Sinh(*x*)

The hyperbolic sine of *X*, *X* assumed in degrees.

Tanh(*x*)

The hyperbolic tangent of *X*, *X* assumed in degrees.

Text functions

Analytica provides several functions for manipulating ***text values*** (sometimes known as *strings*). These are available in the **Text** library.

Some text functions had different names in Analytica 2.0. Please use the new names listed here, even though the old names still work for backward compatibility. See “Forward and backward compatibility” on page 531 for details.

Asc(*t*)

Returns the ASCII code (a number between 0 and 255) of the first character in text value ***t***. This is occasionally useful, for example to understand the alphabetic ordering of text values.

Chr(*n*)

Returns the character corresponding to the numerical ASCII code ***n*** (a number between 0 and 255). Chr and Asc are inverses of each other, for example:

```
Chr(65) → 'A',     Asc(Chr(65)) → 65
Asc('A') → 65,     Chr(Asc('A')) → 'A'
```

Chr is useful for creating characters that cannot easily be typed, such as ***tab***, which is Chr(9) and ***carriage return*** (CR), which is Chr(13). For example, if you read in a text file, x, you can use `SplitText(x, Chr(13))` to generate an array of lines from the text.

FindinText(*t1, t2, start*)

Returns the position of the first occurrence of the text ***t1*** within the text ***t2***, as the number of characters to the first character of ***t1***. If ***t1*** does not occur in ***t2***, it returns 0. For example,

```
Variable People := 'Amy, Betty, Carla'
FindinText('Amy', People) → 1
FindinText('Betty', People) → 6
```

```
FindinText('Fred', People) → 0
```

The optional third parameter, ***start***, specifies the position to start searching at, for example, if you want to find a second occurrence of ***t1*** after you have found the first one.

Joining Text: *a* & *b*

The "&" operator joins (concatenates) two text values to form a single text value, for example:

```
'What is the' + ' number' + '?'
   → 'What is the number?'
```

If one or both operands are numbers, it will first convert them to text using the number format setting for the Variable whose definition contains this function call (or the default suffix format if none is set), for example:

```
'The number is ' & 10^8 →  'The number is 100M'
```

This is also useful for converting (or "coercing") numbers to text. (See page 199.)

Analytica Note: *In Analytica 2.0, the '+' operator also joins two text values, and when applied to a number and a text value first converts the number to text. Of course, it does not convert two numbers to text—it simply sums them! '+' still joins two text values or a text and a number, but now it also issues a warning about this usage, to encourage you to use '&' instead.*

JoinText(*a, i, sep, finalsep*)

Returns the elements of array ***a*** joined together into a single text value over index ***i***. If elements of ***a*** are numeric, they are first converted to text using the number format settings for the Variable whose definition contains this function call. For example,

```
I : ['A', 'B', 'C']
JoinText(I, I) → 'ABC'
A: Array(I, ['VW', 'Honda', 'BMW'])
JoinText(A, I) → 'VWHondaBMW'
```

If the optional parameter ***sep*** is specified, it is inserted as a separator between successive elements, for example:

```
JoinText(A, I, ', ') → 'VW, Honda, BMW'
```

The optional parameter ***finalsep***, if present, specifies a different separator between the second-to-last and last elements of ***a***.

```
JoinText(A, I, ', ', ', and ') → 'VW, Honda, and
BMW'
```

Analytica Note: *As an undocumented feature in Analytica 2.0,* ***Sum****(****a, i****) also converts all numbers to text and joins all the text elements over index* ***i*** *into a text value—but only if at least one element of* ***a*** *is a text value.* ***Sum*** *still behaves this way in Release 3.1, but it first issues a warning. We encourage you to use* ***JoinText****, not* ***Sum*** *to join text values.*

SelectText(*t, m, n*)

Returns text containing the ***m***th through the ***n***th character of text ***t*** (where the first character is ***m***=1)***.*** If n is omitted it returns characters from the ***m***th through the end of ***t***.

```
SelectText('One or two', 1, 3) → 'One'
SelectText('One or two', 8) → 'two'
```

SplitText(*t, sep*)

Returns a list of text values formed by splitting the elements text value ***t*** at each occurrence of separator ***sep***. For example,

```
SplitText('VW, Honda, BMW', ', ') → ['VW',
'Honda', 'BMW']
```

SplitText is the inverse of **JoinText**, if you use the same separators, for example:

```
Var x:=SplitText('Humpty Dumpty sat on a wall.', '
')
→ ['Humpty', 'Dumpty', 'sat', 'on', 'a', 'wall.']
JoinText(x, ' ') → 'Humpty Dumpty sat on a wall.'
```

Analytica Note: With `SplitText()`***,*** ***t*** *must be a single text value, not an array. Otherwise, it might generate an array of arrays of different length. See page 452 on what to do if you want apply it to an array.*

TextLength(*t*)

Returns the number of characters in text ***t***.

```
TextLength('Supercalifragilisticexpialidocious')
→ 34
```

TextLowerCase(*t*)

Returns the text ***t*** with all letters as lowercase. For example,

```
TextLowerCase('What does XML mean?')
→ 'what does xml mean?'
```

TextReplace(*t, t1, t2, all*)

If *all* is omitted or false, it returns text ***t*** with the first occurrence of text ***t1*** replaced by ***t2.*** If *all* is true, it returns text ***t*** with all occurrences of text ***t1*** replaced by ***t2.***

```
TextReplace('StringReplace, StringLength',
'String', 'Text')
→ 'TextReplace, StringLength'
TextReplace('StringReplace, StringLength',
'String', 'Text', True)
→ 'TextReplace, TextLength'
```

TextSentenceCase(*t*)

Returns the text ***t*** with the first character (if a letter) as uppercase, and any other letters as lowercase. For example,

```
TextSentenceCase(SplitText('mary ann FRED
Maylene', ' '))
→ ['Mary', 'Ann', 'Fred', 'Maylene']
```

TextUpperCase(*t*)

Returns the text ***t*** with all letters as uppercase. For example,

```
TextUpperCase('What does XML mean?')
→ 'WHAT DOES XML MEAN?'
```

ReadTextFile(*filename*)

Reads a file ***filename*** and returns its contents as a text value. If ***filename*** contains no directory path, it will try to read from the current folder, usually the folder containing the current model file. If it doesn't find the file, it will open a Windows Browser dialog box to prompt the user. For example,

```
Function LinesFromFile(filename : Atomic Textual)
Definition:
    VAR r :=  SplitText(ReadTextFile(filename),
           Chr(10));
    Index lines :=1..Size(r);
    Array(lines, r)
```

This function reads in the file and splits the text up at the end of each line, with the carriage-return = Chr(10) character. It then defines a local index lines, to be used as the index of the array of lines that it returns.

WriteTextFile(*filename, text: TextType; append, warn: Boolean optional; sep: TextType optional*)

Writes *text* to the file *filename*. The filename is relative to the

current data directory (see "New current directory mangement" on page 15). It returns the full pathname of the file if it is successful in writing or appending to it. By default, the *append* flag is False and *warn* flag is True. If the file doesn't already exist, it creates the file in the current data directory — and if the file does exist, it asks if you want to replace it. If append is True (1), and the file already exists, it appends the text to the end of the file. If warn is False (0), it will not issue a warning before overwriting an existing file when append is False, or when creating a new file when append is True.

If text is an array, it writes each element to the file, inserting separator *sep* between elements, if provided. If text has more than one dimension, you can control the sequence in which they are written by using function `JoinText( )` to join the text over the index you want innermost.

You can write or append to multiple files when filename is an array of file names. If text has the same index(es), it will write the corresponding slice of text to each file — following proper array abstraction.

Converting a number to text

If you apply the '&' operator or **JoinText** to numbers, they convert the numbers to text values, using the number format specified for the Variable or Function in whose definition they appear. You can use this effect to convert ("coerce") numbers into text values, for example:

```
123456789 & '' → '123.5M'
123456789 & '' → '$123,456,789.00'
'The date is: ' & 38345
→ 'The date is: Thu, Dec 25, 2008'
```

Analytica Note: *The actual result depends on Number Format setting for the Variable or function in whose definition the expression appears. The first example assumes the default Suffix format. The second assumes Fixed Point format, with currency and thousands separators checked, and 2 decimal digits. The third assumes the Abbreviated Date format. Use the* ***Number Format*** *dialog on the* ***Result*** *menu to set the formats.*

Converting text to a number

You can use the **Evaluate**(t) function to convert a text representation of a number into an actual number, for example:

```
Evaluate('12350') → 12.35K
```

`Evaluate()` can convert any number format that Analytica can usually handle in an expression—and no others. Thus, it can handle decimals, exponent format, dates, true or false, a '$' at the start of a number (which it ignores), and letter suffixes, like 'K' and 'M'. (See page 465 for more on `Evaluate()`.)

An alternative method, for converting text to a number is to use the `Coerce Numeric` qualifier on a user-defined function. For example, you could define a user-defined function such as:

```
ParseNum( X : Coerce Numeric ) := X
```

Alphabetic ordering of text values

The comparison operators, >, >=, >=, and <, compare the alphabetic ordering (based on ASCII coding) of two text values. For example,

```
'Analytica' < 'Excel' → 1 (true)
```

Using the numerical (ASCII) representation of the characters, means:

a. Digits precede (are smaller than) letters, so

```
'9' < 'A' → 1 (True)
```

b. Uppercase letters precede lowercase letters. If you want to alphabetize without regard to case, first use `TextUpperCase()` (or `TextLowerCase()`) to convert all letters to the same case.

```
'Analytica' > 'excel' → 0 (False)
TextUpperCase('Analytica')
    < TextUpperCase('excel') → 1 (True)
```

c. Letters with accents, umlauts, cedillas, ligatures and other decoration come after undecorated letters, hence alphabetic ordering may be different from what you expect.

Sortindex(*a, i*) sorts text values in ***a*** using the same ordering scheme. But, **Rank**(d) works only on numerical values, and does not rank text values.

Datatype functions

Non-array values in Analytica may be numbers, text, or the special value `undefined`. The functions in this section, found on the **Special** menu, can be used to determine the value type.

Isnan(*x*)

Returns True if *X* is numeric but not a number nor infinity (*i.e.*, if *X* is `NaN`).

```
0/0 → NAN
IsNaN(0/0) → True        IsNaN(5) → False
IsNaN(Inf) → False       IsNaN('Hello') → False
```

Isnumber(*x*)

Returns True if *X* is numeric, including INF or NAN

```
IsNumber(0) → True       IsNumber(0/0) → True
IsNumber(Inf) → True     IsNumber('hi') → False
IsNumber(5) → True       IsNumber('5') → False
IsNumber(NAN) → True
```

Istext(*x*)

Returns True if *X* is a text value.

```
IsText(7) → False        IsText('hello') → True
IsText('7') → True
```

Isundef(*x*)

Returns `True` if **X** is either of the special values `undefined` or `Null`. Equivalently, returns `False` if **X** is a number or a text value.

The value **Undefined** displays as a blank in a result table and generally indicates that a value is unavailable or hasn't been computed. For backward compatibility with releases of Analytica prior to release 3.1, this can also be used to detect **Null**.

The special value `undefined` cannot be directly entered in an expression. However, it may result from the evaluation of certain Analytica expressions. For example, the `Subindex()` function returns **Null** if the given value is not found.

```
Isundef('hello') → False    Isundef(5) → False
Isundef(0/0) → False        Isundef(1/0) → False
Isundef(Subindex(Time*2,1000,Time)) → True
```

Note: *In the last example, Time*2 does not contain the value 1000, so Subindex returns* ***Null****.*

Null, Undefined, NAN, and INF

Analytica may return the following system constants:

Undefined means that a value has never been defined or is uncomputed.

Null means that there is no such item.

NAN means the result is numeric, but not a real number or infinity; *e.g.,* `Sqrt(-1)` or `0/0`

Inf means infinity or a real number larger than can be represented, *e.g.,* `1/0`

-Inf rmeans negative infinity or a number smaller than can be represented, *e.g.,* **-1/0**

Functions such as **Slice**, **Subscript**, **Subindex**, or **MDTable** may return **Null—**for example, when trying to **Slice** out the ***n***th ele-

ment of an array whose index has less than ***n*** elements, for example:

```
Index I := 1..5
Slice(I^2, I, 6) → Null
```

Undefined may result from trying to access the value of an attribute that hasn't been defined (*e.g.,* using Attribute OF Variable), or from attempting to use the value of an optional parameter in a user-defined function that hasn't been provided by the caller.

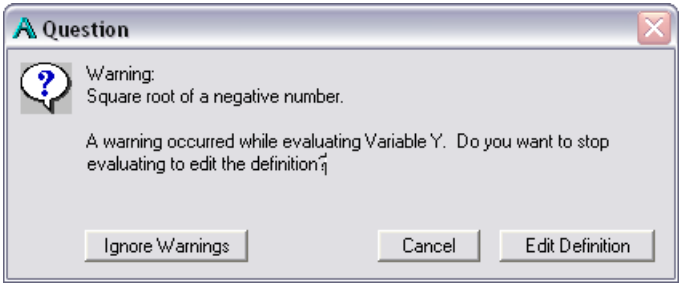
You can test for `Null` using the standard = or <> operators.

Analytica releases prior to 3.1 used **Undefined** for both **Undefined** and **Null**, as defined above.

Warnings

Warnings may occur during evaluation, for example when trying to take the square root of a negative number or divide by zero, for example:

```
VARIABLE X := Sequence(-2, 2)
VARIABLE Y := Sqrt(X) →
```



This Warning dialog gives you the option to ignore this and future warnings. If you select **Ignore Warnings**, `Y` yields:

```
Y → [NAN, NAN, 0, 1, 1.414]
```

The NAN (Not A Number) values may be propagated further into a model.

Analytica 3.1 displays warning conditions detected while evaluating an expression *only if* the resulting value assigned to a Variable contains an explicit error. In the following example, the errant NAN does not appear in the result, so it does not display a warning:

```
Variable Z := IF X<0 THEN 0 ELSE Sqrt(X)
Z → [0, 0, 0, 1, 1.414]
```

Because `(X>0)` evaluates to an array containing both True (1) and False (0) values, the expression will evaluate `Sqrt(X)`, and generate `NAN` as for `Y` above. But, the conditional means that resulting value for `Z` contains no `NANs`, and so Analytica generates no warning when `Z` is evaluated.

You can also make use of the return value, even if it might be errant, as in the following example:

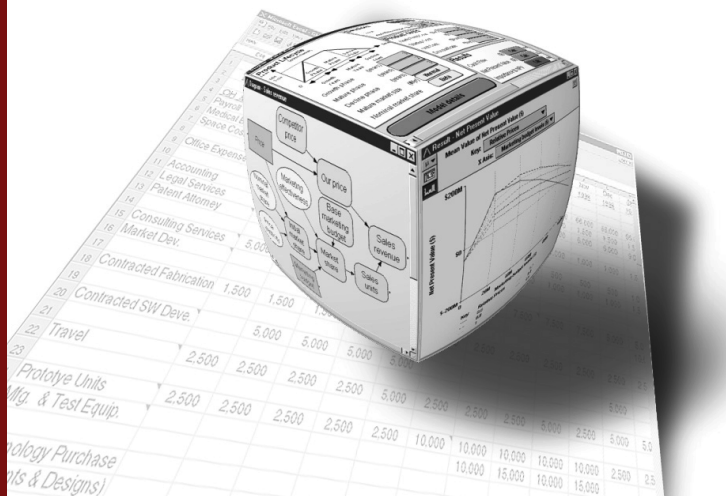
```
VAR x:=sqrt(y);
IF IsNaN(x) THEN 0 ELSE x
```

The commonly-encountered conditions of "subscript or slice value out of range" are now warnings (they used to be errors) with the return value of **Null**, for example:

```
Index I := 1..5
Slice(I^2, I, 6) → Null
```

Chapter 11

Arrays and Indexes



In this Chapter

This chapter shows you how to handle arrays and tables.

11: Arrays and indexes

The value of a Variable may be ***atomic***—a single number, text, reference, or Boolean—or it may be an ***array***—a collection of values, viewable as a table with one or more dimensions. The ease and flexibility with which you can create, operate with, and display multi-dimensional arrays is the source of much of the power of Analytica for creating and managing substantial models. An array's dimensions are identified using ***index Variables***. You can extend a dimension by adding elements to its index, or add a dimension to an array Variable, and the change in dimensions will automatically carry through the rest of the model.

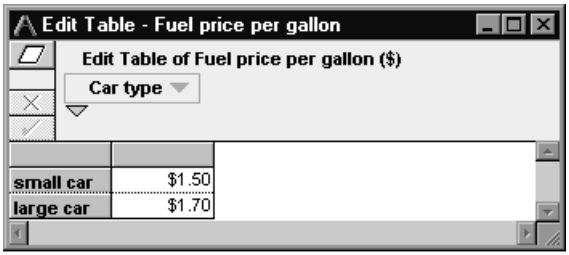
There are some subtleties to the effective use of arrays. Your prior experience with spreadsheets or programming languages may mislead you about how best to use arrays in Analytica. So, if you plan to use arrays in your models, we suggest that you first read the following section, "Introduction to arrays" on page 207 and "Operations on arrays" on page 211. The remainder of this chapter provides the details on how to create index Variables, how to use Edit tables to create array values, and how the arithmetic, comparison, logical, and conditional operators work with arrays. "Advanced Array Functions" on page 235, describes the special functions that create and operate on arrays.

Analytica Note: *Many of the examples in this chapter use the same example data used in Chapter 10, "Using Expressions", see page 188.*

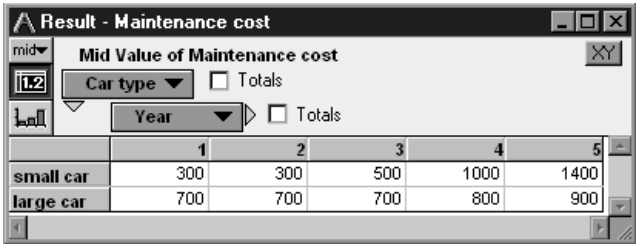
Introduction to arrays

What is an array?

An array is a collection of values that you can view as a table or graph. An array has one or more dimensions, which may appear as the row headers or column headers of a table. For example, the value of Variable *Fuel price per gallon* is a one-dimensional array with two values, $1.50 for the small car (which uses regular gasoline) and $1.70 for the large car (which uses premium gasoline):



Maintenance cost per year is defined as a two-dimensional array, which varies by *Car type* and by *Year*:



The small car is cheap to maintain initially, but it gets more expensive than the large car after 3 years as its components start to wear out and need replacing.

Analytica Note: *You can swap the rows (Car type) and columns (Year) by using the row or column popup menus (see "Index selection area" on page 47).*

Intelligent Arrays™ refers to the full set of features in Analytica for handling array abstraction. See "Intelligent Arrays™" on page 237 for more information.

What is an index?

Each dimension of an array is identified by an index Variable. The index Variable holds the possible values, either a list of numbers or a list of labels. In the examples above, *Car type* is a list of labels, "small car" and "large car". *Year* is a list of numbers.

Car type:
small car
large car

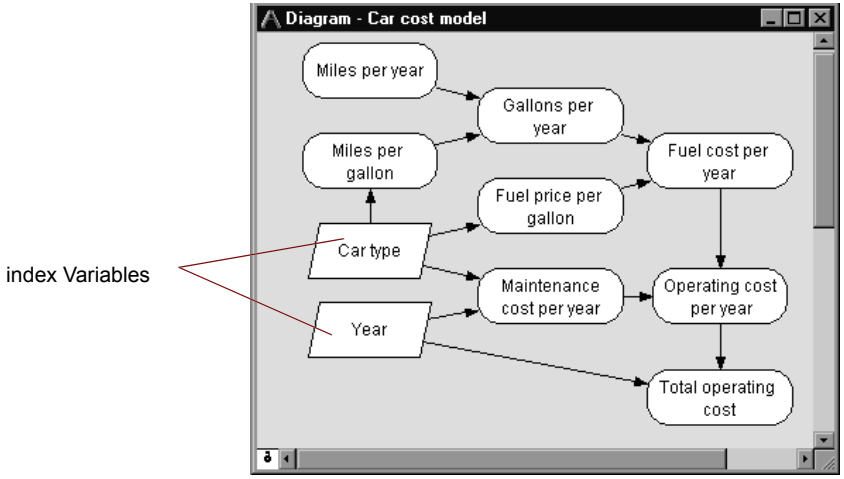
Year:
1
2
3
4
5

To create an index, see “Creating an index” on page 215.

Before creating an array, it is usually best to create the indexes for the array’s dimensions. An index may be used in multiple arrays. When building a model that will use several multidimensional arrays, a key task is to define the indexes.

Index Variables in a diagram

Below is a diagram of a *Car cost model*, which includes the Variables described above. The two index Variables are shown as parallelogram nodes on the diagram.



Fuel price per gallon is the destination of an arrow from *Car type* because it is defined as an array indexed by *Car type*. Similarly, *Maintenance cost per year* has arrows from *Car type* and from *Year*, because it is indexed by both.

Analytica Note: *By default, Analytica does not show arrows to and from index Variables. You can display these arrows, as in this example, by selecting the option in the Diagram Style dialog from the* **Diagram** *menu (see “Diagram Style dialog box” on page 121).*

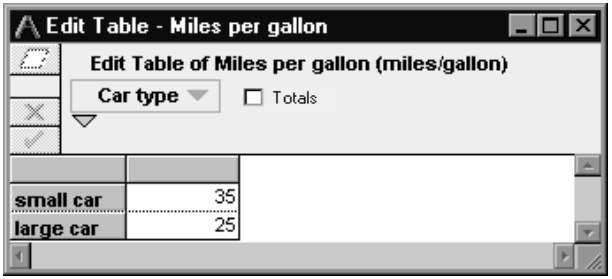
Viewing an array as an Edit table

An ***Edit table*** is a window that appears similar to a Result table. Unlike a Result table, you can select or change the indexes of the array and enter or edit the value of each element. If you select a

Variable defined as an Edit table and click on the edit definition button ( ), you will see its Edit table.

Example (continued from above)

Miles per gallon:



To create or edit an array with an Edit table, see “Creating an array with an Edit Table” on page 225.

Two sources of array value

When you evaluate a Variable and its Result window shows an array value, there are two possible sources. A Variable will have an array value if:

- it is defined as an array using an Edit table, or
- it is defined as an expression calculated from one or more other array-valued Variables.

Array abstraction

Analytica performs operations on arrays without your needing to explicitly identify or iterate over the dimensions of each array. When you use Variables in expressions, you only need to refer explicitly to dimensions that are relevant to the operations being performed. If the actual values involve dimensions other than those that appear in your expressions, Analytica will automatically abstract over those dimensions with no extra effort on your part.

Because array abstraction automatically takes care of most iteration over arrays, Analytica expressions seldom contain explicit looping constructs. Individual expressions involving multi-dimensional arrays can be very simple, while in other languages the same operations would require multiple nested loops over the non-relevant dimensions.

Designing a model often requires you to make hard trade-offs between computational complexity, which dimensions to include, and the degree of detail. Spreadsheets and other programming languages force you to make these decisions early before you have implemented your algorithms and obtained the information that is relevant for making these trade-offs. The automatic management of dimensionality provided by array abstraction makes it easy for you make these trade-offs late in the model building process.

Operations on arrays

Arithmetic operations and simple functions generalize straightforwardly when they are applied to arrays, according to the dimensions of the arrays. This section gives some simple examples.

Operation on a scalar and an array

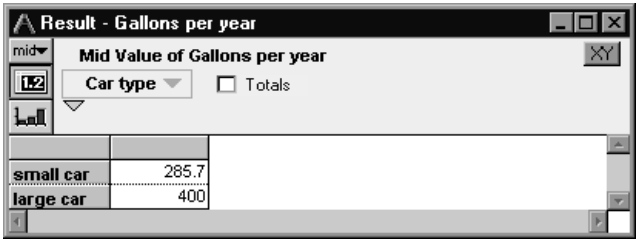
An operation applied to a scalar and an array results in an array of the same shape, applying the scalar operation to each element in the array.

Example (continued)

Miles_per_year: `10K  (a scalar)`

Gallons per year: `Miles_per_year / Miles_per_gallon`

The result of an operation (division in this case) combining a scalar and an array is a result array with the same index(es) as the original array:



Operation on two arrays with the same indexes

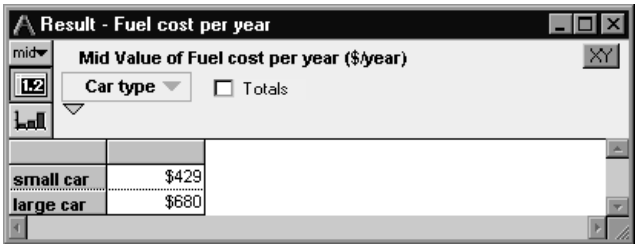
An arithmetic operator applied to two arrays with the same indexes creates another array with the same indexes. Analytica applies the operator to pairs of corresponding elements.

Example (continued)

Fuel cost per year:

```
Fuel_price_per_gall * Gallons_per_year
```

Both *Fuel price per gallon* and *Gallons per year* are arrays with the same index, *Car type*. The result is an array also indexed by *Car type*, containing the value obtained by multiplying the corresponding elements of each array:



Operation on a one- and two-dimensional array

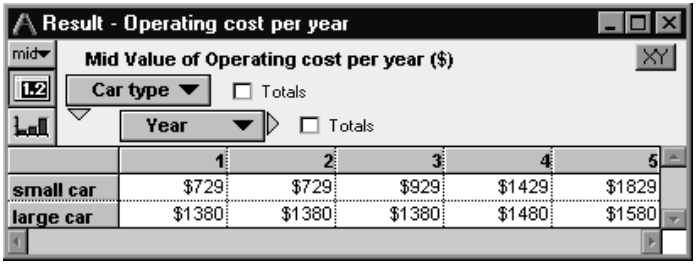
An arithmetic operator applied to a one-dimensional array and a two-dimensional array, that have one index in common, creates another two-dimensional array with the same two indexes.

Example (continued)

Op_cost_per_year:

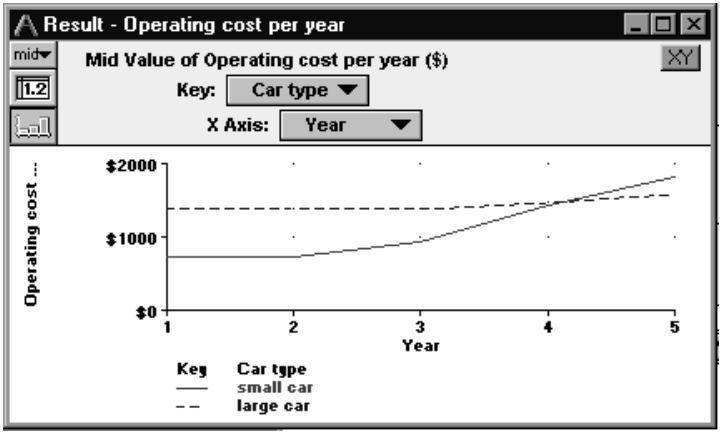
```
Fuel_cost_per_year + Maintenance_per_year
```

Operating cost per year is the sum of a one dimensional Variable indexed by *Car type* and a two-dimensional Variable indexed by *Car type* and *Year*. The result is a two-dimensional array indexed by both indexes:



Each *Car type* (row) in the result uses the fuel cost and maintenance cost for the corresponding *Car type*. Each *Year* (column) uses the same annual fuel cost, which does not change by year, and the corresponding maintenance cost, which does change by year.

Changing the above table to a graph, using the graph button ( ), shows:



The graph shows how the operating costs of the small car are less than the costs of the large car in the first 3 years and grow to be larger in the 5th year, crossing over just after the 4th year.

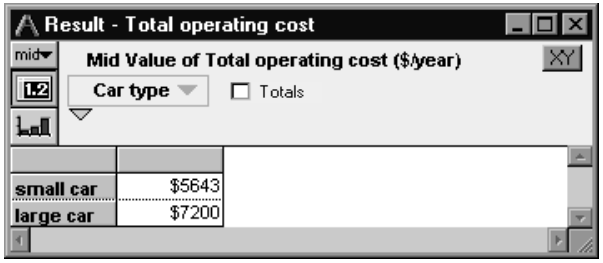
Summing over an index Variable

The `Sum()` function sums an array over one index, giving a result without that index.

Example (continued)

Total operating cost: `Sum(Op_cost_per_year,Year)`

This operation sums *Operating cost per year* over the *Year* dimension, producing a result indexed only by the *Car type* dimension:



Analytica Note: *The expression does not need to mention any other possible indexes, such as Car type.*

Because the `Sum()` function eliminates one index of an array, it is called an array-reducing function. Analytica includes several

array-reducing functions (see “Array-reducing functions” on page 247).

Operation on arrays with different dimensions

An arithmetic operator applied to two one-dimensional arrays with different indexes creates a two-dimensional array with both indexes.

Example (continued)

Miles per year is redefined as a list (see “Creating a list” on page 216):

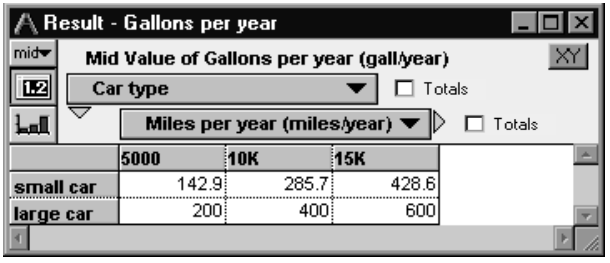
Miles_per_year: `[5000, 10K, 15K]`

A list is a one-dimensional array that is indexed by itself. Lists are eligible to be used as indexes of other arrays.

The definitions of *Miles_per_gallon*, an array indexed by *Car type*, and *Gallons per year*, a ratio, remain unchanged.

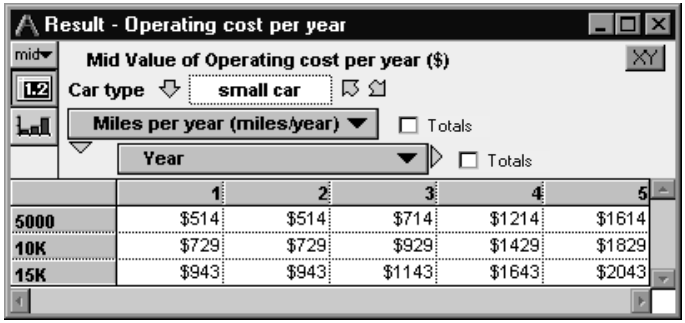
Gallons per year: `Miles_per_year / Miles_per_gallon`

The result of *Gallons per year* is now an array indexed by both *Miles per year* and *Car type* (compare to the definitions in the section “Operation on a scalar and an array” on page 211):

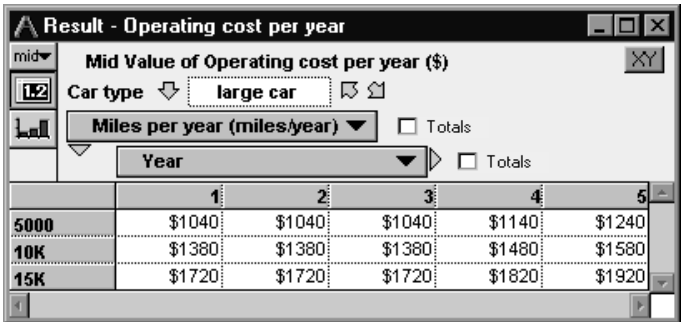


Each value in the table is computed from the *Miles per year* for the column divided by the *Miles per gallon* for each *Car type* (row). For example, 5000 miles per year divided by the large car’s 25 miles per gallon gives 200 gallons per year.

The list value for *Miles per year* propagates through the model as a new dimension to all its dependent Variables. Recomputing the result for *Operating cost per year* now gives a three-dimensional table with an added index of *Miles per year*:



The results for the other *Car type* can be displayed by clicking on the diagonal arrow ( ):



General rule for operations on arrays

We can summarize and generalize the behavior of an operation on two arrays with the following rule: An operation on two arrays yields an array whose indexes are the union of the indexes of the two arrays. In this way, Analytica combines arrays without requiring explicit iteration over each index. We call this feature of generalized operations for multidimensional values Intelligent Arrays™.

Creating an index

Analytica includes a specific class of Variable node—the ***index Variable***—to identify dimensions of arrays. Other Variables, such as decision nodes, are often also used to identify dimensions of arrays. Actually, any Variable defined as a list (one-dimensional array) can serve as an index to an array. For clarity in your model

diagram, use the index Variable whenever possible. (The terms ***index*** and ***index Variable*** are used interchangeably here.)

Create an index Variable in the following way:

1. Select the Edit tool ( ) and have the Diagram window active.
2. Drag the parallelogram shape ( ) from the node palette to the diagram.
3. Give the new index a descriptive title.
4. Define the index as a list, list of labels, or sequence (see below).

Creating a list

To define a Variable as a list, first select the Variable and open one of the following:

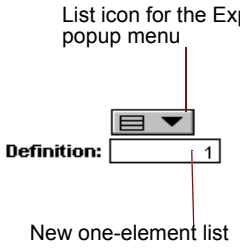
- The Variable's Object window.
- The Attribute panel of the diagram (see "Displaying the Attribute" on page 35).
- In the Attribute panel, select **Definition** from the Attribute popup menu (see "The Attribute popup menu" on page 35) as the Attribute to display.

To create a list:

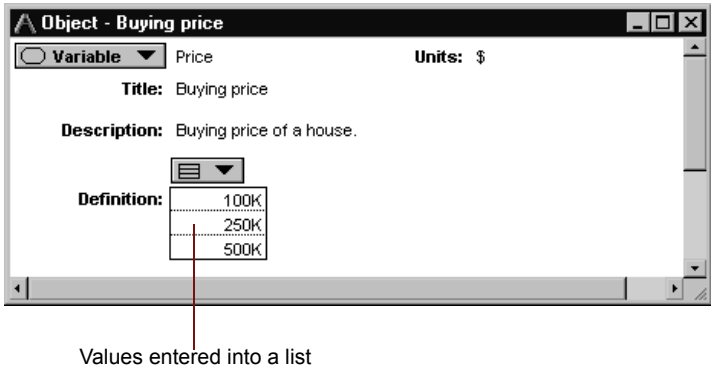
1. Press the Expression popup menu above the definition field and select **List** (for numbers) or **List of Labels** (for text).

(If the Variable already has a definition, Analytica confirms that you wish to replace it. Click on **OK** to replace the definition with a one-element list.)

A one-element list is displayed in the definition field.



2. Select the element by clicking on it.
3. Type in a number or expression (for **List**) or text (for **List of Labels**).
4. Press *Enter* and type in the next value.
5. Repeat step 4 until you have entered all the values you want.



Autofilling a list

Analytica gives the first cell of a list the default value 1 or the value of the Variable's previous definition. When you press *Enter*, Analytica gives the second cell the value of the first cell plus 1.

After you have entered at least two values, Analytica gives each new cell a value that is incremented by the difference between the last two values.

Autofilling a list of labels

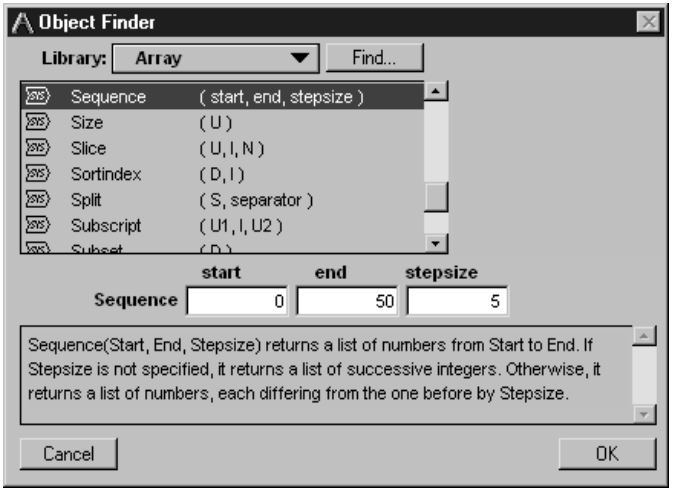
Analytica gives the first cell of a list of labels the default text value `item 1`. Analytica gives each subsequent cell receives a value the same as the previous cell.

Creating a list with the Sequence option

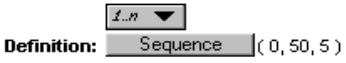
For the classes of nodes that are often defined as lists, such as index and decision Variables, the Expression popup menu includes the **Sequence** option.

The **Sequence** option provides a quick way to define a list of equally spaced numbers.

When you select **Sequence**, the Object Finder opens, showing the `Sequence()` function (see page 223).



After entering the *Start, End,* and *Stepsize* values, click **OK**; the definition field shows the **Sequence** button with its parameters.

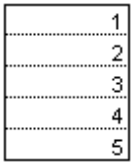


Analytica Note: *To edit the Sequence, click on the* **Sequence** *button.*

List vs. list of labels

You can display a list or list of labels in two ways: "List View" or "Expression View". The "List View" displays by default; the Expression popup menu shows the list or list of labels icon.

List view



The "Expression View" displays when you select  in the Expression popup menu.

Expression view

```
[1,2,3,4,5]
```

List (of numbers)

In a list of numbers (usually called simply a list), each value is a number or an expression that evaluates to a number. For example, the sequence of five integers above is a list.

List of labels

In a list of labels, every value is text. For example, the set of states below is a list of labels; in the expression view, each label is contained in single quotation marks.

List view

Expression view

```
['Alabama', 'Alaska', 'Arizona','Arkansas']
```

To include a single quote (apostrophe) as part of the text in a label in expression view, insert two adjacent single quotes, *e.g.*

```
['can''t','won''t','didn''t']
```

Mixing numbers and text

A list can include a mix of cells containing text and numbers. In both views the text is contained in single quotation marks. For example:

List view

Expression view

```
[1, 'Alabama', 2, 'Alaska']
```

If you attempt to mix numbers and text in a list of labels, all the values will be treated as text. For example:

List view

Expression view

```
['1', 'Alabama', '2', 'Alaska']
```

Analytica Note: *A list cell can contain any valid expression, including one that refers to other Variables or one that evaluates to an array.*

Editing a list

You can edit a list by changing, adding or deleting ***cells*** (list items).

Inserting cells

To add a cell at the end of the list, select the last cell and press *Enter* or the down arrow key.

To insert a cell anywhere other than at the end of the list, select a cell and choose **Insert Rows** (Ctrl-I) from the **Edit** menu. The value in the selected cell is duplicated in the new cell.

To insert several contiguous cells in the middle of the list, select the number of cells you want to insert and choose **Insert Rows** (Ctrl-I) from the **Edit** menu. The value of the last selected cell is duplicated in the new cells.

Deleting cells

To delete one or more cells, select them and do one of the following:

- Choose **Delete Rows** (Ctrl-K) from the **Edit** menu.
- Press *Delete*.

Analytica Note: *If you add or delete a cell in a list that is an index of an Edit Table, the corresponding elements of the table are also added or removed (see “Editing a table” on page 228).*

Navigating a list

Use the up and down arrow keys to move the cursor up and down the list.

Functions that create indexes

Use the **List** option in the Expression popup menu to define a Variable as a list of numbers or text values (labels) (see “Creating a list” on page 216). You can also create a list within a Variable definition using the constructs and functions described below.

[*u1, u2, u3, … um*]

The list of expressions, separated by commas and surrounded by brackets, creates a list, whose values are *u1, u2, u3, … um*.

Using square brackets to specify a list directly as an expression is equivalent to using the **List** or **List of Labels** options in the Expression popup menu, as described in “Creating a list” on page 216, according to the type of values.

Examples

```
[8000, 12K, 15K]
['VW', 'Honda', 'BMW']
```

CopyIndex(*i*)

CopyIndex makes a copy of the values of index ***I***, to be assigned to a new Index Variable (global or local). For example, suppose you want to create a matrix of distances between a set of origins and destinations, where the destinations are the same set of cities as the origins:

```
Index Origins
Definition:['London', 'New York', 'Tokyo',
    'Paris', 'Delhi', 'Lagos']
Index Destinations
Definition: CopyIndex(Origins)

Variable Flight_times := Table(Origins,
Destinations)
```

If `Destinations` was the same `Origins`, rather than a copy, the resulting table would have only one dimension. By defining `Destinations` with `CopyIndex`, it becomes an independent dimension.

m .. *n*

Returns a sequence from ***m*** to ***n*** by increments of 1 when ***n*** >= ***m*** or from ***m*** to ***n*** by -1 if ***n*** < ***m***. It is simply a convenient shorthand for Sequence(m,n). For example,

```
2003..2006 → [2003, 2004, 2005, 2006]
```

Analytica Note: *The parameters n and m must be scalars, that is single numbers. Otherwise, it would result in a non-rectangular array. See “Functions needing scalars and array abstraction” on*

page 452 on how to use this construct in a way that supports array abstraction.

Sequence (*Start, End, Stepsize*)

Creates a list of numbers increasing or decreasing from *Start* to *End* by increments (or decrements) of *Stepsize*. *Stepsize* is optional and must be a positive number; if it is omitted, Analytica uses increments of 1. *Start*, *End*, and *Stepsize* must be deterministic scalar numbers, not arrays.

Using this function is equivalent to using the **Sequence** option in the Expression popup menu, as described in "Creating a list with the Sequence option" on page 218.

The expression `m .. n` using the operator `".."` is a version of `Sequence(m, n, 1)`, that is it generates a list of sequential numbers from **m** to `n`.

Library Array

Examples If *End* is greater than *Start*, the sequence is increasing:

```
Sequence(1,5) →
```

List view:

1
2
3
4
5

Expression view: `[1,2,3,4,5]`

If *Start* is greater than *End*, the sequence is decreasing:

```
Sequence(5,1) → [5,4,3,2,1]
```

If *Start* and *End* are not integers, and if *Stepsize* is not specified, Analytica rounds them first:

```
Sequence(1.2,4.8) → [1,2,3,4,5]
```

If *Stepsize* is specified, Analytica can create non-integer values from *Start* to *End* with the *Stepsize*:

```
Sequence(0.5,2.5,0.5) → [0.5,1,1.5,2,2.5]
```

Sortindex (*D, I*)

D is an array indexed by *I*. *SortIndex(D,I)* returns the elements of *I*, rearranged to indicate the ordering of the values in *D* (from

smallest to largest value). The result is indexed by *I*. If *D* is indexed by dimensions other than *I*, each "column" is individually sorted, with the resulting sort order being indexed by the extra dimensions. To obtain the sorted array D, use (see page 258):

```
D[I=Sortindex(D,I)]
```

When *D* is a one-dimensional array, the second parameter is optional. When the second parameter is omitted, the result is an unindexed list. The one-parameter form should be used only when it is necessary to obtain an unindexed result, such as when the result is being assigned to an index Variable. The one-parameter form cannot array abstract if a new dimension is added to *D*.

Library Array

Examples

Maint_costs:
Car_type ▶

	VW	Honda	BMW
	1950	1800	2210

`SortIndex (Maint_costs,Car_type)` →
SortIndex: Car_type ▶

	VW	Honda	BMW
	Honda	VW	BMW

`SortIndex (Maint_costs)` →
SortIndex ▶

	Honda	VW	BMW
	1800	1950	2210

Define *Index_new* as an index node:

Index_new: `Sortindex(Maint_costs)`

`Subscript(Maint_costs, Car_type, Index_new)` →

	Honda	VW	BMW
	1800	1950	2210

Note: *Example Variables are defined on page 188.*

Subset (*D*)

Returns a list containing all the elements of *D*'s index for which *D*'s values are true (that is, non-zero). *D* must be a one-dimensional array.

When to use: Use `Subset()` to create a new index that is a subset of an existing index.

Library Array

Example `Subset(Years < 1987)` → `[1985, 1986]`

Note: *Example Variables are defined on page 188.*

Unique(A,I)

Returns a maximal subset of I such that each indicated slice of A along I is unique.

When to use: Use `Unique()` to remove duplicate slices from an array, or to identify a single member of each equivalence class.

Library Array

Example

DataSet: PersonNum ▼, Field ▶

	LastName	FirstName	Company
1	Smith	Bob	Acme
2	Jones	John	Acme
3	Johnson	Bob	Floorworks
4	Smith	Bob	Acme

`Unique(DataSet,PersonNum)` → [1,2,3]

`Unique(DataSet[Field='Company'],PersonNum)` → [1,3]

Creating an array with an Edit Table

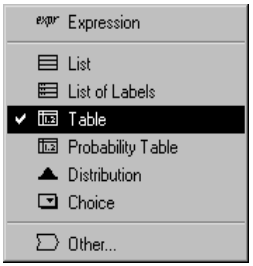
To define a Variable as an array (table), first select the Variable and open one of the following:

- The Variable's Object window.
- The Attribute panel of the Diagram window (see "Displaying the Attribute" on page 35).

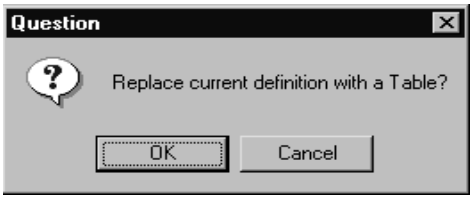
  In the Attribute panel, select **Definition** from the Attribute popup menu (see "The Attribute popup menu" on page 35) as the Attribute to display.

To create a table:

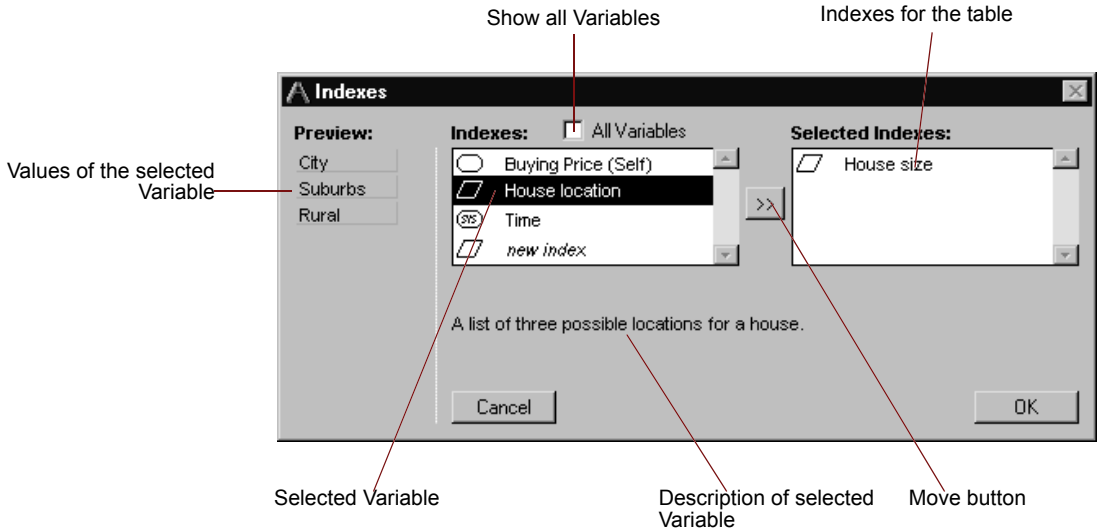
1. Press the Expression popup menu above the definition field and select **Table**.



If the Variable already has a definition, you are asked to confirm that you wish to replace it.



2. The Indexes dialog box displays for selecting the table's indexes (dimensions).



3. Select a Variable from the Indexes list and click on the move button (>>), or double-click on the Variable, to select it as an index of the table. Repeat for each index you want.
4. Click on **OK** to create the table and open the Edit Table window for editing the table's values (see "Editing a table" on page 228).

Indexes dialog box

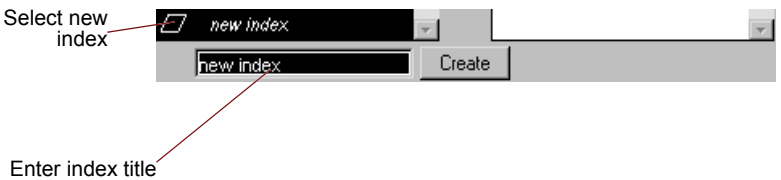
The Indexes dialog box contains (see figure above):

Preview	A list of the values of the selected index Variable. If the selected Variable is not a list, it says "Can't use as index."
All Variables checkbox	If checked, the Indexes list includes all Variables in the model. If not checked, it lists only Variables of the class Index and Decision, plus the Variable being defined (*Self*) and *Time*. If you select this Variable (*Self*) as an index, the Variable itself holds the alternative index values.
Selected indexes	A list of all indexes already selected for this Variable.
New index	Select to create a new index.

Creating a new index

You can create an index Variable in the course of creating a table, in the following way:

1. Select **new index** from the Variables list in the Indexes dialog box.
2. Enter a title for the index.



3. Click on the **Create** button.

4. To make the new index an index of the table, click on the >> button.

Enter the values of the Index in the Edit Table window (see the following section).

Removing an index

To remove an index from a table:

1. Select the index from the Selected Indexes list.
2. Click on the << button.

Removing an index will leave the first table (slice) along that index as the value of the array.

System index Variables *Run* and *Time*

Analytica includes two system index Variables: *Run* and *Time*. You can generally treat these index Variables like any other index Variable.

Run is the index for the array of sample values for probabilistic simulation. You can examine the array with the Sample uncertainty mode (see “Sample” on page 56) or the `Sample()` function (see page 340).

Time is the index for dynamic simulation. It is the only index permitted for cyclically dependent modeling (see “Modeling Changes over Time” on page 359).

Editing a table

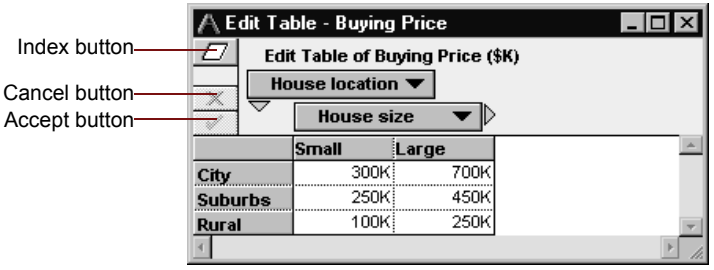
To open the Edit Table window, click on the **Edit Table** button in either:

- The Object window (see “The Object window” on page 32)
- The Attribute panel of the diagram (see “Displaying the Attribute” on page 35)

  In the Attribute panel, select **Definition** from the Attribute popup menu (see “The Attribute popup menu” on page 35).

The Edit Table window

The Edit Table window appears similar to the Result window Table view (see “Viewing a result as a table” on page 49). The difference is that you can add indexes and edit (change) the values in cells in an Edit Table window.



Selecting cells

Select a single cell

Click on the cell once.

Select multiple cells

Drag the mouse from one cell to another to select a rectangular region.

Editing a cell

Enter an expression into a cell in the same manner as you would in a definition field. Press *Enter* to accept the value and to select the next cell.

Analytica Note: *Edit tables are not designed for defining complex expressions in each cell. Rather than define a cell as a complex expression, create a new Variable, and define it as the complex expression. Then enter the new Variable’s identifier in the Edit Table cell.*

Adding and deleting cells

To add cells by adding rows or columns to a table:

1. Select the row (or column) before which you wish to insert a new one.
2. Choose **Insert Rows** (or **Insert Columns**) from the **Edit** menu.

   The added cells contain zeros.

To delete cells by deleting rows (or columns) in a table:

1. Select the row (or column) you wish to delete.
2. Choose **Delete Rows** (or **Delete Columns**) from the **Edit** menu.

You can only add or delete an element of an index in this way if the index was defined as a list or list of labels. You cannot modify an index defined as a `Sequence()` or other expression from an Edit Table.

Analytica Note: *When you change an index in this way, you will also affect any other arrays using this index.*

Copying and pasting cells

You can copy a cell or a range (two-dimensional rectangular region) of cells from a table.

To copy a cell or region:

1. Select the cell or region.
2. Choose **Copy** from the **Edit** menu (Ctrl-C).
3. Paste the item(s) into another cell or region by selecting **Paste** from the **Edit** menu (Ctrl-V). The region you paste into must either be a single cell at the top left corner of the destination region, or it must have the same size as the copied region.

Adding or removing indexes

Click on the Index button () to add more indexes (increasing the number of dimensions) or to remove indexes (decreasing the number of dimensions). The Indexes dialog box appears (see “Indexes dialog box” on page 227).

Any new index of size *n* copies the current table with its current values *n* times along the new dimension.

Saving the table

Click on the Accept button () to perform a syntax check and store changes you have made.

Click on the Cancel button () to discard changes made since opening the window, or since the last time you clicked on the Accept button.

If you close an Edit Table window without clicking on the Accept or Cancel buttons, the changes are accepted.

Calculating with arrays

Conventions for array examples

Most of the examples in this and the next chapter show Variables defined as tables (arrays) or evaluating to arrays. Indexes and arrays in these examples are represented as follows:

- An index or list and its values

  IndexName:

value1	*value2*	*valueN*

- An expression that evaluates to a scalar or an array

  `expression` → *result*

- A one-dimensional array

  Index_a ▶

	a	***b***	***c***
	value	*value*	*value*

- A two-dimensional array

  Index_b ▼, *Index_a* ▶

	a	***b***	***c***
x	*value*	*value*	*value*
y	*value*	*value*	*value*
z	*value*	*value*	*value*

- A three-dimensional array

Index_a ▼, *Index_b* ▶, *Index_c displayed value* `

	a	b	c
x	value	value	value
y	value	value	value
z	value	value	value

Scalar functions

Any function that takes a scalar parameter (*e.g.*, see “Math functions” on page 190) can be applied to an array, resulting in an array of the same shape. Each element of the resulting array is calculated by applying the function to the corresponding element of the input array.

This example takes the square root of every value in a one-dimensional array.

```
Sqrt([1, 2, 3, 4, 5]) →
```

	1	2	3	4	5
	1.000000	1.414214	1.732051	2.000000	2.236068

Arithmetic operations

An arithmetic operator (+, -, *, /, ^) applied to two arrays results in an array indexed by every index Variable in the two input arrays. Several examples illustrate this:

- An arithmetic operator applied to a scalar and an array results in an array of the same shape, applying the scalar operation to each element in the array:

  X:

2	3	4

  ```
  10 * X ^ 2 →
  ```

  X ▶

	2	3	4
	40	90	160

- An arithmetic operator applied to two arrays that are both indexed by the same Variable creates another array indexed by the same Variable with the operator applied to pairs of corresponding elements:

`X + Sqr(X)` →

X ▶

	2	3	4
	6	12	20

- An arithmetic operator applied to two arrays with different index Variables (or with no index Variables) creates an array indexed by every index Variable in the two arrays, with the operator applied to all pairs of elements:

  Inflation:

1.05	1.10	1.15

  Price:

5K	10K	15K

  `Inflation * Price` →

  Price ▼, *Inflation* ▶

	1.05	1.10	1.15
5000	5250	5500	5750
10K	10.5K	11K	11.5K
15K	15.75K	16.5K	17.25K

Comparison and logical operations

The comparison operators (>, >=, =, <=, < and so on) and the logical operators (**`And`** and **`Or`**) combine array values in the same way as the arithmetic operators (see “Operators” on page 180 for the full list of operators). The only difference from the arithmetic operators is that both comparison and logical operators return arrays of Boolean values, and the logical operators treat their operands as Boolean. Each cell contains either 1 (*True*) or 0 (*False*) (see “Boolean or logical values” on page 180).

For example:

Vw_price:

8250	10K	15K

Honda_price:

12.5K	15K	20K

`Honda_price > Vw_price` →
Vw_price ▼, *Honda_price* ▶

	12.5K	15K	20K
8250	1	1	1
10K	1	1	1
15K	0	0	1

`Honda_price = Vw_price` →
Vw_price ▼, *Honda_price* ▶

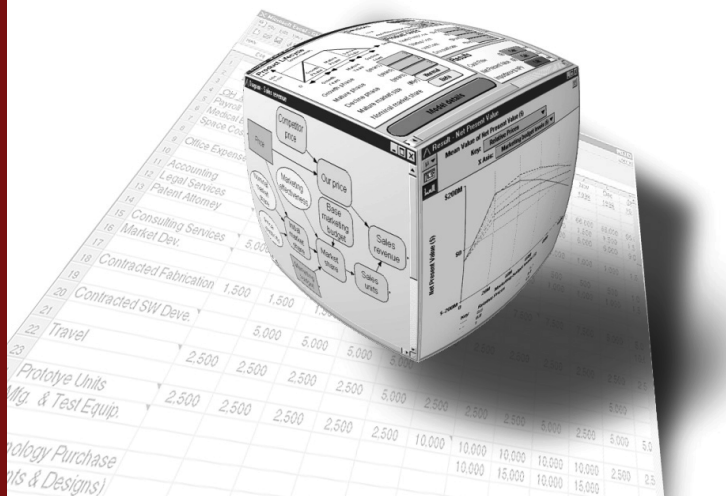
	12.5K	15K	20K
8250	0	0	0
10K	0	0	0
15K	0	1	0

`Honda_price > VW_price OR Honda_price = Vw_price` →
Vw_price ▼, *Honda_price* ▶

	12.5K	15K	20K
8250	1	1	1
10K	1	1	1
15K	0	1	1

Chapter 12

Advanced Array Functions



In this Chapter

This chapter explains the nature and benefits of Intelligent Arrays™, and describes a variety of more advanced array functions that enable you to make the best use of them, including functions for reducing, transforming, selecting, flattening, interpolating arrays; matrix functions and financial functions.

Functions for uncertainty and sensitivity analysis are covered in later chapters.

12: Function reference

Analytica provides a large collection of built-in functions for performing common mathematical, financial, statistical, and array computations.

Overview

This chapter describes Analytica's advanced built-in functions for dealing with Arrays. It is organized by the type of function:

- Functions that create arrays ("Functions that create arrays" on page 242).
- Functions that reduce an array to another array with one fewer dimension ("Array-reducing functions" on page 247).
- Functions that return an array with the same number of dimensions as the input array ("Transforming functions" on page 251).
- Functions that select part or a slice of an array ("Selecting, slicing, and subscripting arrays" on page 255).
- Functions that interpolate values between array elements ("Interpolation functions" on page 262).
- Other array functions ("Other array functions" on page 265).
- Matrix functions for two-dimensional arrays ("Matrix functions" on page 267).
- Functions commonly used for financial computations ("Financial functions" on page 272).

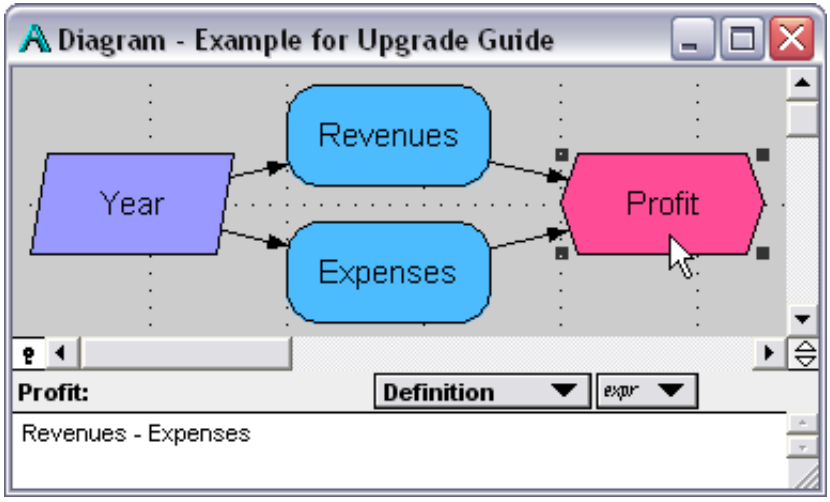
Intelligent Arrays™

Intelligent Arrays™ are one of the most useful and powerful features of Analytica, yet their full implications are easy to miss. Consider this definition in Analytica:

```
Variable Profit := Revenues - Expenses
```

It works equally well if **`Profits`**, **`Revenues`**, and **`Expenses`** are each scalars (single numbers), or arrays of one or more dimensions. If **`Revenues`** and **`Expenses`** are both are indexed by **`Year`**, it

computes the **Profit** for each **Year**, using the corresponding **Revenues** and **Expenses** for that **Year**, as in this example:



The figure below shows an Influence Diagram above and corresponding array values below.

Result - Variables

mid

Mid Value of Variables

XY

Variables

Totals

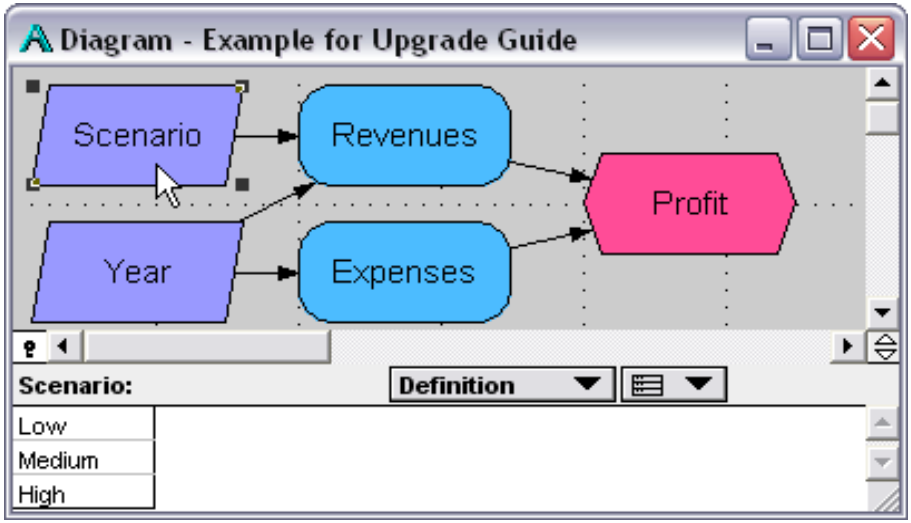
Year

Totals

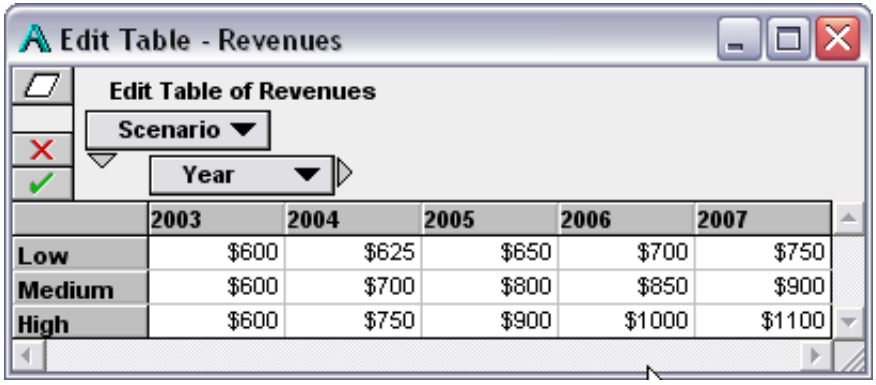
	2003	2004	2005	2006	2007
Expenses	600	700	800	850	900
Revenues	700	700	720	750	800
Profit	-100	0	80	100	100

The definition of **Profit** remains the same, no matter what the dimensions of **Revenues** and **Profits**. If **Revenues** is a scalar (a single number), **Profit** treats it as if it is the same each **Year**.

Or if **Revenues** are specified for three different **Scenarios**—**Low**, **Medium**, and **High**—it computes the corresponding **Profit** for each **Scenario**, whether or not **Expenses** vary by **Scenario**.



Now **Revenues** is indexed by **Scenario** as well as **Year**:



The value of flexibility

This flexibility is very convenient for the modeller. Changing dimensions is much more complicated in a spreadsheet or standard programming language. In a spreadsheet, you would have to explicitly create each of the three Variables as a table with the required number of dimensions. And you would have to craft carefully the formula with the requisite relative cell references, and copy it into each cell of **Profit**. In a programming language, such as Fortran, C++, Java, or Visual Basic, you would have to put the formula inside loops to iterate over the dimensions. A simple one-dimensional case might look something like this:

```
Dim Profit[2000..2010], Revenues[2000..2010],
Expenses[2000..2010]
For Years := 2000 To 2010 DO
    Profit[Years]:=Revenues[Years]-
    Expenses[Years]
```

If you decide to add a dimension, such as **Scenario** to **Revenues** and **Profit**, you would need to redimension both Variables, add another **For** loop over Scenario, and add a second subscript to **Profit** and **Revenues**, nearly doubling the complexity of the program.

To do the same thing in a spreadsheet would require adding two extra rows to the tables of **Revenues** and **Profit**, copying the name **Scenario** and its three values, **Low**, **Medium**, and **High** as row headers into both tables, rewriting the base cell formula in **Profit**, and finally stretching the cell formula across the columns and then down the two new rows. Theeffort to create the two-dimensional model is more than double the effort to create original one-dimensional model, which is itself more than double the scalar model.

Now consider extending the time horizon from 2007 to 2010, a common need in business models. In Analytica, you simply edit the definition of **Year**, changing the 2007 to 2010. The arrays for all three Variables extend automatically over the three extra years. The input Edit tables for **Revenues** and **Expenses** are filled out with zeroes for these new years. You just need to open up those tables and fill in the numbers you want. In a spreadsheet, you would need to extend each table by hand, including copying the Year column headers 2008, 2009, 2010 for each table, and stretching the formulas for **Revenue** over nine new cells.

Array abstraction and Intelligent Arrays

All this work to expand the tables and the formulas for **Revenue** is distracting and quite unnecessary—the relationship between **Profit, Revenues** and **Expenses** should be entirely separate from whatever dimensions they happen to have. The principle of abstracting the representation of the relationships between the Variables from the dimensions of those Variables is sometimes known as ***array abstraction***. Few computer languages offer support for array abstraction. Analytica offers a unique and extensive approach to array abstraction, which is the basis of its **Intelligent Arrays™**. Once you have mastered the basics of Intelligent Arrays, you may find it hard to imagine going back to a modeling

environment (such as a spreadsheet or standard programming language) without array abstraction.

Choosing the right level of detail

Intelligent Arrays provides a flexibility that greatly simplifies the process of developing a model. When starting a model, it is rarely obvious how large and how detailed to make each dimension. Should time be modeled as years, quarters, or months? Should the time horizon be 5 years, 10 years, or 20 years? Is it necessary to treat each geographic region separately, and if so, by continent, nation, or state (province)? How you answer these questions has a large effect not only the accuracy of the model, but also on the quantity of data you will need, the effort to build and verify the model, and the computer resources (time and memory) needed to calculate them.

Ideally, you specify the essential relationships between the Variables first, and decide their dimensions later. You may want to try different levels of detail, starting out with few and simple dimensions, then refining the model by expanding or adding dimensions. You should be able to experiment with the level of detail and computational effort until you get a good balance between effort and precision. With spreadsheets and conventional languages, this kind of experimentation requires so much rebuilding and testing at each stage, that it is usually completely impractical. The result is that models are often too simple or too complicated—or, often, both, with too much detail in areas that do not much matter and not enough detail in areas that do. This kind of stepwise refinement is much easier with Intelligent Arrays, encouraging you to create models with a good balance of accuracy and effort.

Errors, testing, and reliability

Array abstraction also promotes *reliability* by reducing errors and making any errors easier to detect. In a spreadsheet, there are several easy ways to make errors when copying cell references, resulting in frequent bugs that are hard to detect. For example, mistakes in absolute versus relative cell references, or accidentally stretching a sum over only part rather than all of a row or column. In programming languages, it is also easy to make errors in handling dimensions, such as confusing rows and columns in the sequence of subscripts. With Analytica, the relationships are much simpler: There is a single expression defining each Vari-

able, rather than one for each cell in the result. Expressions are uncluttered by looping constructs. You define a Sum over an identified Index, no matter what other dimensions an array may have. This simplicity makes expressions easier to write in the beginning and easier to review for correctness later. Furthermore, provided the formulas support array abstraction, there is no need to modify formulas as you extend or add dimensions—in those Variables, or elsewhere in the model. You can have justifiable confidence that the model remains correct as you extend or add dimensions.

Intelligent Arrays enable several other important capabilities of Analytica. One key feature is support for representing and propagating uncertainty. An uncertain Variable is represented as a random sample of values from its underlying probability distribution, over a dimension indexed by `Run`. The `Run` index has values from 1 to the sample size. Each expression containing one or more uncertain Variables automatically computes its result over all the random samples, generating a result indexed by `Run` (as well as any other dimensions). Array abstraction means that this works, without you (the modeler) having to worry about this extra dimension. Similarly, in parametric analysis, you can set one (or more) input Variables (parameters) each to a number of alternative values to explore the effects of this variation. These values create an array indexed by the input parameters, which is automatically propagated through the model to generate a corresponding table of values for each output, indexed by the alternative values of its parametric inputs.

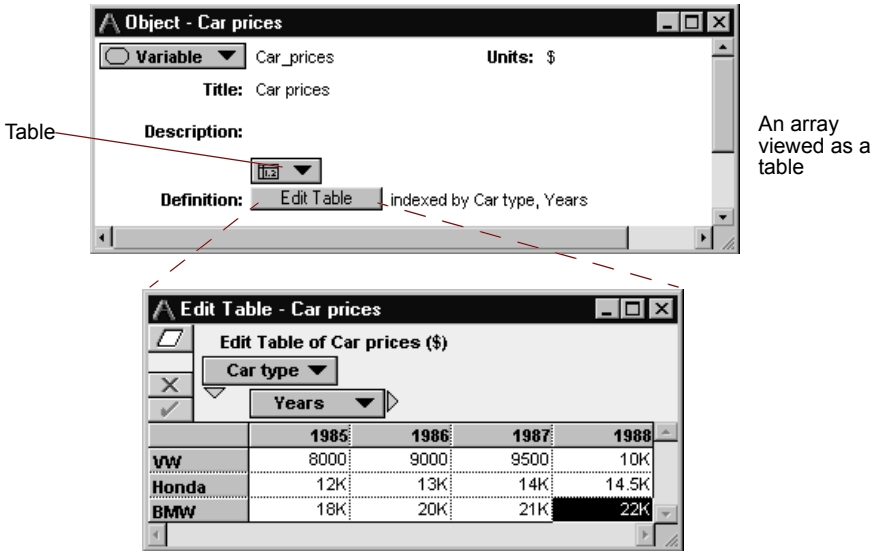
Exceptions to array abstraction

While the vast majority of Analytica functions and constructs fully support Intelligent Arrays™ -- that is they automatically generalize from single (scalar) values to multidimensional arrays -- there are a few that do not without special care. See “Ensuring array abstraction” on page 451 for details.

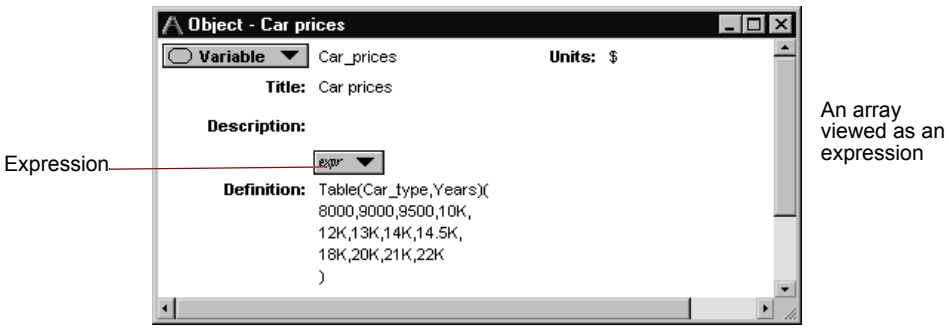
Functions that create arrays

Use the **Table** option in the Expression popup menu to define a Variable as an array (see “Creating an array with an Edit Table” on page 225).

For more flexibility and control, you can define a Variable as an array by entering the **`Array()`** or **`Table()`** function as an expression.



An array viewed as an expression appears in the **`Table()`** function syntax:



Array(*I1, I2, … In, A*)

Assigns a set of indexes, *I1, I2, … In*, as the indexes of the array *A*, with *I1* as the index of the outermost dimension (changing least rapidly), *I2* as the second outermost, and so on. *A* must have at least *n* dimensions. The elements of *A* are listed in square brackets as the last parameter, or *A* is a previously defined array.

Use **Array()** to specify an array directly as an expression. **Array()** is similar to **Table()** (see page 245); in addition, it lets you define an array with repeated values (see Example 3), and change indexes of a previously defined array (see Example 4).

Library Array

Example 1 Definition viewed as an expression:

```
Array(Car_type, [32, 34, 18])
```

Definition viewed as a table:

Car_type ▶

	VW	Honda	BMW
	32	34	18

Note: *Example Variables are defined on page 188.*

Example 2 If an array has multiple dimensions, then the elements are listed in nested brackets, following the structure of the array as an array of arrays (of arrays..., and so on, according to the number of dimensions).

Definition viewed as an expression:

```
Array(Car_type, Years, [[8K, 9K, 9.5K, 10K],
[12K, 13K, 14K, 14.5K], [18K, 20K, 21K, 22K]])
```

Definition viewed as a table:

Car_type ▼, *Years* ▶

	1985	1986	1987	1988
VW	8000	9000	9500	10K
Honda	12K	13K	14K	14.5K
BMW	18K	20K	21K	22K

The size of each array in square brackets must match the size of the corresponding index. In this case, there is an array of three elements (for the three car types), and each element is an array of four elements (for the four years). An error message displays if these sizes don't match. See also **Size()** on page 266.

Note: *Example Variables are defined on page 188.*

Example 3 If an element is a scalar where an array is expected, **Array()** expands it to create an array with the scalar value repeated across a dimension.

Definition viewed as an expression:

```
Array(Car_type, Years, [[8K, 9K, 9.5K, 10K], 13K,
[18K, 20K, 21K, 22K]])
```

Definition viewed as a table:

Car_type ▼, *Years* ▶

	1985	1986	1987	1988
VW	8000	9000	9500	10K
Honda	13K	13K	13K	13K
BMW	18K	20K	21K	22K

Note: *Example Variables are defined on page 188.*

Example 4

Use `Array()` to change an index of a previously defined array.

Car_model:

Jetta	Accord	320

Table_a: `Table(Car_type) (32, 34, 18)`

Table_b: `Array(Car_model, Table_a)` → *Car_model* ▶

	Jetta	Accord	320
	32	34	18

Note: *Example Variables are defined on page 188.*

Table (*I1, I2, … In*) (*u1, u2, u3, … um*)

Creates an *n*-dimensional array of *m* elements, indexed by the indexes *I1, I2, … In*. In the set of indexes, *I1* is the index of the outermost dimension, varying the least rapidly.

The second set of parameters, *u1*, *u2 … um*, specifies the values in the array. The number of values, *m*, must equal the product of the sizes of all of the dimensions.

Each *u* is an expression that evaluates to a number, text value or probability distribution. It can also evaluate to an array, causing the dimensions of the entire table to increase. *u* cannot be a literal list.

Both sets of parameters are enclosed in parentheses; the separating commas are optional except if the table values are negative.

Use `Table()` to specify an array directly as an expression. `Table()` is similar to `Array()` (see page 243); `Table()` requires *m* numeric or text values.

A definition created as a table from the Expressions popup menu uses `Table()` in expression view.

Library Array

Example 1 Definition viewed as an expression:

```
Table(Car_type) (32, 34, 18)
```

Definition viewed as a table:

Car_type ▶

	VW	Honda	BMW
	32	34	18

Note: *Example Variables are defined on page 188.*

Example 2 Definition viewed as an expression:

```
Table(Car_type, Years)
(8K, 9K, 9.5K, 10K, 12K, 13K, 14K, 14.5K, 18K, 20K,
21K, 22K)
```

Definition viewed as a table:

Car_type ▼, *Years* ▶

	1985	1986	1987	1988
VW	8000	9000	9500	10K
Honda	12K	13K	14K	14.5K
BMW	18K	20K	21K	22K

Note: *Example Variables are defined on page 188.*

Example 3 A table created with blank (zero) cells appears in expression view without the second set of parameters.

Definition viewed as a table:

Car_type ▼, *Years* ▶

	1985	1986	1987	1988
VW	0	0	0	0
Honda	0	0	0	0
BMW	0	0	0	0

Definition viewed as an expression:

```
Table(Car_type, Years)
```

Note: *Example Variables are defined on page 188.*

Array-reducing functions

An ***array-reducing function*** operates across a dimension of an array and returns a result that has one dimension less than the number of dimensions of its input array. When applied to an array of *n* dimensions, a reducing function produces an array that contains *n*-1 dimensions.

The function `Sum(X, I)` illustrates some properties of reducing functions.

Examples

`Sum(Car_prices, Car_type)` →

Years ▶

	1985	1986	1987	1988
	38K	42K	44.5K	46.5K

`Sum(Car_prices, Years)` →
Car_type ▶

	VW	Honda	BMW
	36.5K	53.5K	81K

`Sum(Sum(Car_prices, Years), Car_type)` → `171K`

The second parameter, *I*, specifying the dimension over which to sum, is optional. But if the array, *X*, has more than one dimension, Analytica may not sum over the dimension you expect. For this reason, it is safer *always* to specify the dimension index explicitly in `Sum()` or any other array-reducing function.

If the index Variable, *I*, is not a dimension of the array, *X*, `Sum(X, I)` returns *X* unreduced. In this way, Analytica will evaluate the model properly even if the number of dimensions changes.

Note: *Example Variables are defined on page 188.*

Area (*R, I, X1, X2*)

Returns the area (sum of trapezoids) under array *R* across index *I* between *X1* and *X2*. *I* must contain increasing numbers. *X1* and

X2 are optional; if they are not specified, the area is calculated across all of *I*.

If *X1* or *X2* fall outside the range of values in *I*, the first value (for *X1*) or last value (for *X2*) are used. **`Area()`** computes the total integral across I, returning a value with one less dimension than *R*. Compare **`Area()`** to **`Integrate()`** (see page 253).

Library Array

Example

`Area(Cost_in_time, Time, 0, 5000)` →
Car_type ▼, *Mpg* ▶

	26	30	35
VW	9653	12.42K	15.18K
Honda	10.11K	12.84K	15.86K
BMW	13.65K	16.42K	19.18K

Note: *Example Variables are defined on page 188.*

Argmax (*R, I*)

Returns the corresponding value in index *I* for which *R* is the maximum. If more than one value equals the maximum, returns the index of the last occurrence.

Library Special

Example

`Argmax(Car_prices, Car_type)` →
Years ▶

	1985	1986	1987	1988
	BMW	BMW	BMW	BMW

To obtain the corresponding value in index *I* for which *A* is the minimum, use **`Argmax(-A,I)`**.

`Argmax(-Car_prices, Car_type)` →

	1985	1986	1987	1988
	VW	VW	VW	VW

Note: *Example Variables are defined on page 188.*

Average (*X, I*)

Returns the mean value of all of the elements of array *X*, averaged over index *I*.

Library Array

Examples

```
Average(Mpg) → 30.33
```

```
Average(Car_prices, Car_type)→
```
Years ▶

	1985	1986	1987	1988
	12.67K	14K	14.83K	15.5K

Note: *Example Variables are defined on page 188.*

JoinText(*A, I, separator,finalSeparator*)

Returns the elements of *A* (as text) concatenated along *I* and separated by *separator*. If the optional *finalSeparator* parameter is provided, it is used as the final separator. If any elements are numeric, they are converted to text values using the number format settings for the current node.

A: I ▶

	1	2	3
	VW	Honda	BMW

```
JoinText(A,I,', ') → 'VW, Honda, BMW'
```

```
JoinText(A,I,', ',' and ') → 'VW, Honda and BMW'
```

Max (*X, I*)

Returns the highest valued element of *X* along index *I*.

Library Array

Examples

```
Max(Years) → 1988
```

```
Max(Car_prices, Years) →
```
Car_type ▶

	VW	Honda	BMW
	10K	14.5K	22K

To obtain the maximum of two numbers, first turn them into an array:

```
Max([10,5]) → 10
```

Note: *Example Variables are defined on page 188.*

Min (*X, I*)

Returns the lowest valued element of *X* along index *I*.

Library Array

Examples `Min(Years)` → `1985`

`Min(Car_prices, Years)` →
Car_type ▶

	VW	Honda	BMW
	8000	12K	18K

To obtain the minimum of two numbers, first turn them into an array:

`Min([10, 5] )` → `5`

Note: *Example Variables are defined on page 188.*

Product (*X, I*)

Returns the product of all of the elements of *X*, along the dimension indexed by *I*.

Library Array

Examples `Product(Mpg)` → `27.3K`

`Product(Cost, Mpg)` →
Car_type ▶

	VW	Honda	BMW
	5.905G	14.47G	28.78G

Note: *Example Variables are defined on page 188.*

Subindex (*A, U, I*)

Returns the value of *I* corresponding to value *U* in array *A*. If more than one value corresponds, returns the index value of the last occurrence. For the values that do not correspond, returns `undefined` (shows as blank, see also `IsUndef`(..) on page 201).

`Argmax()` uses `Subindex(A, Max(A, I), I)` to return the index value corresponding to the maximum value in *A*. See `Argmax()` on page 248.

Library Special

Examples

`Subindex(Car_prices, 12K, Car_type)` → *Years* ▶

	1985	1986	1987	1988
	Honda			

`Subindex(Car_prices, 12K, Years)` → *Car_type* ▶

	VW	Honda	BMW
		1985	

If *U* is an array of values, an array of index values is returned.

`Subindex(Car_prices, [12K, 21K], Car_type)` → *Subindex* ▼, *Years* ▶

	1985	1986	1987	1988
12K	Honda			
21K			BMW	

Note: *Example Variables are defined on page 188.*

Sum (*X, I*)

Returns the sum of array *X* over the dimension indexed by Variable *I*.

Library Array

Examples

`Sum(Mpg)` → `91`

`Sum(Car_prices, Years)` → *Car_type* ▶

	VW	Honda	BMW
	36.5K	53.5K	81K

Note: *Example Variables are defined on page 188.*

Transforming functions

A ***transforming function*** operates across a dimension of an array and returns a result that has the same dimensions as its input array.

The function `Cumulate(X,I)` illustrates some properties of transforming functions.

Example

`Cumulate(Car_prices,Years)` →
Car_type ▼, *Years* ▶

	1985	1986	1987	1988
VW	8000	17K	26.5K	36.5K
Honda	12K	25K	39K	53.5K
BMW	18K	38K	59K	81K

The second parameter, *I*, specifying the dimension over which to cumulate, is optional. But if the array, *X*, has more than one dimension, Analytica may not cumulate over the dimension you expect. For this reason, it is safer *always* to specify the dimension index explicitly in any transforming function.

Note: *Example Variables are defined on page 188.*

Cumproduct (*X, I*)

Returns an array with each element being the product of all of the elements of *X* along dimension *I* up to, and including, the corresponding element of *X*.

Library Array

Example

`Cumproduct(Cost_in_time, Time)` →
Mpg ▼, *Time* ▶, *Car_type* = *VW*

	0	1	2	3	4
26	2185	5.012M	12.07G	30.54T	81.11Q
30	2810	8.292M	25.69G	83.57T	285.5Q
35	3435	12.39M	46.92G	186.6T	778.9Q

Note: *Example Variables are defined on page 188.*

Cumulate (*X, I*)

Returns an array with each element being the sum of all of the elements of *X* along dimension *I* up to, and including, the corresponding element of *X*.

If *X* is not indexed by *I*, `Cumulate(X,I)` operates as if *X* were indexed by *I*, but constant across *I*. Using this, a convenient trick for numbering the elements of an index is to use `Cumulate(1,I)`.

Library Array

Example

`Cumulate(Cost_in_time, Time)` →
Mpg ▼, *Time* ▶, *Car_type* = *VW* `

	0	1	2	3	4
26	2185	4479	6888	9417	12.07K
30	2810	5761	8859	12.11K	15.53K
35	3435	7042	10.83K	14.8K	18.98K

`Cumulate(1,Car_type)` →
Years ▶

	VW	Honda	BMW
	1	2	3

Note: *Example Variables are defined on page 188.*

Integrate (*R, I*)

Returns the result of applying the trapezoidal rule of integration of array *R* over index *I*. `Integrate()` computes the cumulative integral across *I*, returning a value with the same number of dimensions as *R*. Compare `Integrate()` to `Area()` (see page 247).

An alternative syntax is `Integrate(R1, R2, I)`, which returns the integral of array *R1* over array *R2*. If *R2* has one dimension, its index must also be an index of *R1* and *I* is optional. If *R2* has more than one dimension, then *I* is required and must be an index of both *R1* and *R2*.

Library

Array

Example

`Integrate(Cost_in_time, Time)` →
Mpg ▼, *Time* ▶, *Car_type* = *VW* `

	0	1	2	3	4
26	0	2240	4591	7060	9653
30	0	2881	5905	9081	12.42K
35	0	3521	7218	11.1K	15.18K

Note: *Example Variables are defined on page 188.*

Normalize (*R, I*)

Returns an array that is normalized array *R*, so the area across index *I* is 1.

`Normalize()` does not force the values along index *I* to sum to 1; to make the values sum to 1, divide *R* by `Sum(R, I)`.

An alternative syntax is `Normalize(R1, R2, I)`, which returns the normalized array of array *R1* over array *R2*. If *R2* has one dimension, its index must also be an index of *R1* and *I* need not be stated. If *R2* has more than one dimension, then *I* is required and must be an index of *R1* and *R2*.

Library Array

Example

`Normalize(Cost_in_Time, Time)` →
Mpg ▼, *Time* ▶, *Car_type* = *VW*

	0	1	2	3	4
26	0.2264	0.2377	0.2496	0.2620	0.2752
30	0.2263	0.2377	0.2495	0.2620	0.2752
35	0.2264	0.2377	0.2496	0.2620	0.2751

Note: *Example Variables are defined on page 188.*

Rank (*X, I*)

Returns an array of the rank values of *X* across index *I*. The lowest value in *X* has a rank value of 1, the next-lowest has a rank value of 2, and so on. *I* is optional if *X* is one-dimensional. If *I* is omitted when *X* is more than one-dimensional, the innermost dimension is ranked.

If two values are equal, they receive the same rank and the next higher value receives a rank 2 higher.

Library Array

Examples

`Rank(Mpg)` →
Mpg ▶

	26	30	35
	1	2	3

`Rank(Car_prices, Car_type)` →
Car_type ▼, *Years* ▶

	1985	1986	1987	1988
VW	1	1	1	1
Honda	2	2	2	2
BMW	3	3	3	3

Note: *Example Variables are defined on page 188.*

Uncumulate (*X, I, firstElement*)

Uncumulate(X,I) returns an array whose first element (along *I*) is the first element of *X*, and each other element is the difference between the corresponding element of *X* and the previous element of *X*. Uncumulate(X,I,firstElement) returns an array with the first element along *I* equal to *firstElement*, and each other element equal to the difference between the corresponding element of *X* and the previous element of *X*.

`Uncumulate(X,I)` is the inverse of `Cumulate(X,I)`. `Uncumulate(X,I,0)` is similar to a discrete differential operator.

Library Array

Example

`Uncumulate(Cost_in_time, Time)` →
Mpg ▼, *Time* ▶, *Car_type = VW* `

	0	1	2	3	4
26	2185	109	115	120	127
30	2810	141	147	155	163
35	3435	172	180	189	199

`Uncumulate(Cost_in_time, Time,0)` →
Mpg ▼, *Time* ▶, *Car_type = VW* `

	0	1	2	3	4
26	0	109	115	120	127
30	0	141	147	155	163
35	0	172	180	189	199

Note: *Example Variables are defined on page 188.*

Selecting, slicing, and subscripting arrays

Analytica includes several functions that are useful for selecting an element or slice of an array. A slice may be a single cell —a number, Boolean or text value — or a subarray, with one less dimension than the array from which it was sliced.

Unlike most computer languages, with Analytica you identify the dimension you want to subscript or slice over by naming the index—so you don't need to remember which index refers to rows, to columns, or higher dimensions. Rows and columns are

not intrinsic to the array representation — they are they are just a matter of how you choose to display the array in a table.

With `Choice()`, you can select an element from a list.

With `Slice()`, you can select the nth element or "plane" of an array. With `x[I=U]` and `Subscript()`, you can select the element or "plane" of an array whose index matches a given value.

All these functions return a result that has one dimension less than the number of dimensions of its input array.

Choice (*I, n, inclAll*)

Appears as a popup menu in the definition field, allowing selection of the *n*th item from *I* (see “Creating a popup menu” on page 163). `Choice()` must appear at the topmost level of a definition. It cannot be used inside another expression. The optional *inclAll* parameter controls whether the "All" option (n=0) appears on nucleoli popup (*inclAll* defaults to *True*).

Library Array

Examples `Choice(Years, 2)` → `1986`

If *n*=0, all values of *I* are returned:

`Choice(Years, 0)` →
Years ▶

	1985	1986	1987	1988

Note: *Example Variables are defined on page 188.*

Slice (*U, I, N*)

Returns the element or cross-section of array *U*, for which index *I* has position *N*. *I* must be an index of *U*, and *N* must be an integer or array of integers between 1 and the length of *I*.

If *N* is an integer, the result of `Slice()` is an array indexed by all indexes of *U* except *I*. If *N* is an array, the result of `Slice()` is also indexed by the indexes of *N*.

If *U* is a scalar, `Slice(U,I,N)` returns *U*.

Slice (*U, N*)

If Slice has only two parameters, and U has a single dimension, it returns the Nth element of U. For example:

```
Index Quarters := 'Q' & 1..4

Slice(Quarters, 2) → 'Q2'
```

This method is the only way to extract an element from an unindexed array, for example:

```
Slice(2000..2003, 4) → 2003
```

It also works to get the nth slice of a multidimensional array over an unindexed dimension, for example:

```
Slice(Quarters & ' ' & 2000..2003, 4) → Array(Quarters, ['Q1 2003', 'Q2 2003', 'Q3 2003', 'Q4 2003'])
```

Analytica Note: *If **a** is a scalar, or if **a** is an array with two or more indexed dimensions and no unindexed dimensions,* ***Slice(a, n)*** *simply returns **a**.*

Library Array

Examples Here, Analytica returns the values in *Cost* corresponding to the first element in *Car_type*, that is, the values of *VW:*

```
Slice(Cost, Car_type, 1) →
```
Mpg ▶

	26	30	35
	2185	1705	1585

Here, *N* is an array of positions:

```
Slice(Cost, Car_type, [1, 2]) →
```
Mpg ▶

	26	30	35
1	2185	1705	1585
2	2810	2330	2210

Note: *Example Variables are defined on page 188.*

Subscript (*U1, I, U2*)

Returns the element or slice of array *U1*, for which index *I* has value *U2*. *I* must be an index of *U1*, and *U2* must be value(s) of *I*.

If *U2* is a single value, the result of `Subscript()` is an array indexed by all indexes of *U1* except *I*. If *U2* is an array, the result of `Subscript()` is also indexed by the indexes of *U2*.

If *U1* is a single value, `Subscript(U1,I,U2)` returns *U1*.

`Subscript(U1,I,U2)` is equivalent to `x[I = U]` when **x** is a Variable identifier that evaluates to *U1*. `Subscript()` allows *U1* to be an arbitrary expression.

Library Array

Examples To see the values in *Cost* corresponding to `Mpg = 26`:

`Subscript(Cost, Mpg, 26)` →
Car_type ▶

	VW	Honda	BMW
	2185	2810	3435

Here *U2* is an array of values:

`Subscript(Cost, Car_type, ['VW', 'Honda'])` →
Car_type ▼, *Mpg* ▶

	26	30	35
VW	2185	1705	1585
Honda	2810	2330	2210

Example of an arbitrary expression as the first parameter:

`Subscript(Cost/12, Mpg, 26)` →
Car_type ▶

	VW	Honda	BMW
	182.1	234.2	286.2

Note: *Example Variables are defined on page 188.*

x[*I* = *U*]

Returns a specific element or slice of an array, where ***x*** is the identifier of an array Variable, *I* is an index Variable, and *U* is one or more elements of index *I* that corresponds to the desired array element. `x[I = U]` is equivalent to `Subscript(U1,I,U2)` when ***x*** is a Variable identifier that evaluates to *U1*.

`Subscript(x, i, vj)` and `x[i=u]` just two ways to do the same thing. The only difference is that with `Subscript(x, I, v)`, `x`

can be any expression, while for `x[i=u]`, `x` must be the identifier of a variable.

Library Special

Examples

`Car_prices[Car_type = 'VW']` →
Years ▶

	1985	1986	1987	1988
	8000	9000	9500	10K

`Car_prices[Car_type = ['VW', 'Honda']]` →
Years ▶

	1985	1986	1987	1988
VW	8000	9000	9500	10K
Honda	12K	13K	14K	14.5K

You can specify more than one index when each index is given a single value.

`Car_prices[Car_type = 'Honda', Years = 1986]` → `13K`

Note: *Example Variables are defined on page 188.*

X [*Time-n*]

X*`[Time-n]`** returns the value of Variable ***X for the time period that is `n` time periods prior to the current time period. This function is only valid for Variables defined using the `Dynamic()` function. See “Dynamic (initial1, initial2..., initialn, Expr)” on page 362

Array flattening functions

The `MDArrayToTable()` function "flattens" a multi-dimensional array into a two-dimensional table. The `MDTable()` function does the inverse, creating a multi-dimensional array from a table of values. Viewing tabular results in a multi-dimensional form via `MDTable()` often provides informative new perspective on existing data.

Many external application programs, including spreadsheets and relational databases, are limited to two-dimensional tables. Thus, when transferring multi-dimensional data between these applications and Analytica, it may be necessary to convert multi-dimensional data into two-dimensional tables before transferring.

MDArrayToTable(*A, I, L*)

Transforms a multi-dimensional array, *A*, into a two-dimensional array *(i.e.*, a table) indexed by *I* and *L*. The result contains one row along *I* for each element of *A*. *L* must contain a list of names of the indexes of *A*, followed by one final element. All elements of *L* must be text values. The column corresponding to the final element of *L* contains the cell value. If *L* does not contain all the indexes of *A*, array abstraction will create a set of tables indexed by the dimensions not listed in *L*.

Before using `MDArrayToTable()`, you must define the index *I* with the appropriate number of elements. The number of elements in *I* may be either `size(A)`, or the number of non-zero elements of *A (*in which case the resulting table will contain only the nonzero elements), otherwise an error results.

If the number of elements in *I* is equal to the number of non-zero elements of *A*, `MDArrayToTable()` acts like the inverse of **MDTable( )** on a table that contains a row for only the nonzero elements of the array.

Library Array

Example

```
Rows := sequence(1,size(Cost_in_time))

Cols := ['Mpg','Time','Car_type','Cost']

MDArrayToTable(Cost_in_time,Rows,Cols) →
```

Rows▼, *Cols*▶

	Mpg	Time	Car_type	Cost
1	26	0	VW	2185
2	26	0	Honda	2385
3	26	0	BMW	3185
4	26	1	VW	2294
5	26	1	Honda	2314
6	26	1	BMW	3294
7	26	2	VW	2409
. . .				
45	35	4	BMW	5175

Note: *Example Variables are defined on page 188.*

MDTable(*T,Rows,Cols,Vars,conglomFn,missingval*)

Returns a multi-dimensional array from a two-dimensional table of values. *T* is a two-dimensional array (*i.e.*, a table) indexed by *Rows* and *Cols*. Each row of *T* specifies the coordinates of a cell in a multi-dimensional array, along with the value for that cell.

The dimensions of the final result are given by the optional parameter *Vars*. *Vars* must be a list of index identifiers or index names. The length of *Cols* must be one greater than the length of *Vars*.

If *Vars* is omitted, the dimensions of the final result are specified by the first *n-1* elements of *Cols* (*n=size(Cols)*). In this case, the elements of Cols must be index identifiers or index names.

The first *n-1* columns of *T* specify the coordinates of a cell in the result. The final column of *T* specifies the value for the indicated cell.

Before using MDTable, you must define all of the indexes for the result. Each index *must* include all values that occur in the corresponding column of *T* or an error will result. The `Unique()` function is useful for defining the necessary indexes.

It is possible that two or more rows of *T* specify identical coordinates. In this case, a *conglomeration function* is used combine the values for the given cell. The *conglomFn* parameter is a text value specifying which conglomeration function is to be used. Possible values are: "sum" (default), "min", "max", "average", or "product".

It is also possible that no row in *T* corresponds to a particular cell. In this case, the cell value is set to *missingval*, or if the *missingval* parameter is omitted, the cell value is set to *undefined.* U*ndefined* values can be detected using the `IsUndef()` function.

Library

Array

Example

Suppose *T, Rows,* and *Cols* are defined as indicated by the following table:

Rows▼, *Cols*▶

	Car_type	Mpg	X
1	VW	26	2185
2	VW	30	1705
3	Honda	26	2330
4	Honda	35	2210
5	BMW	30	2955
6	BMW	35	2800
7	BMW	35	2870

```
MDTable(T,Rows,Cols,[Car_type,Mpg],
        average',"n/a") →
```

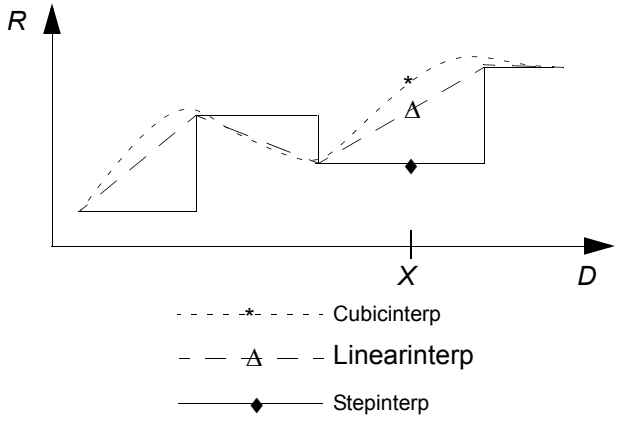
Car_type▼, *Mpg*▶

	26	30	35
VW	2185	1705	n/a
Honda	2330	n/a	2210
BMW	n/a	2955	2835

Notice that in the example, *Rows* 6 and 7 both specified values for *Car_type=BMW, Mpg=35*. The 'average' conglomeration function was used to combine these.

Interpolation functions

Analytica includes three functions that interpolate across arrays. The graph below is a simple comparison of the three.



The first two examples use the following Variables:

Index_a:

a	b	c

Index_b:

1	2	3

Array_a:
Index_a ▼, *Index_b* ▶

	1	2	3
a	7	-3	1
b	-4	-1	6
c	5	0	-2

Cubicinterp (*D, R, X, I*)

Returns the natural cubic spline interpolated values of *R* along *D*, interpolating for values of *X*. *D and R* must both be indexed by *I*, and *D* must be increasing along *I.*

For each value of *X*, `Cubicinterp()` finds the nearest values from *D*, and using a natural cubic spline between the corresponding values of *R*, computes the interpolated value. If *X* is less than the minimum value in *D*, it returns the first value in *R*; if *X* is greater than the maximum value in *D*, it returns the last value for *R*.

Library Special

Example `Cubicinterp(Index_b, Array_a, 1.5, Index_b)` → *Index_a* ▶

	a	b	c
	0.6875	-2.875	2.219

Linearinterp (*D, R, X, I*)

Returns linearly interpolated values of *X*, given *R* representing an arbitrary piecewise linear function. *D* and *R* must both be indexed by *I*, and *D* must be increasing along *I*. *R* is an array of the corresponding output values for the function (not necessarily increasing and may be more than one dimension). *X* may be probabilistic and/or an array.

For each value of *X*, **Linearinterp()** finds the nearest two values from *D* and interpolates linearly between the corresponding values from *R*. If *X* is less than the minimum value in *D*, it returns the first value in *R*. If *X* is greater than the maximum value in *D*, it returns the last value in *R*.

Library Special

Example

```
Linearinterp(Index_b, Array_a, 1.5, Index_b) →
```
Index_a ▶

	a	b	c
	2	-2.5	2.5

Stepinterp (*D, A, X, I*)

Returns the element or slice of array *A* for which *D* has the smallest value that is greater than or equal to *X*. *D* and *A* must both be indexed by *I*, and *D* must be increasing along index *I*. If *X* is greater than all values of *D*, returns the element for which *D* has the largest value.

If *X* is a single value, the result of **Stepinterp()** is an array indexed by all indexes of *A* except *D*'s index. If *X* is an array, the result of **Stepinterp()** is also indexed by the indexes of *X*.

Stepinterp() is similar to **Subscript()** (see page 257); however, **Subscript()** selects based on the index value being equal to *X*, while **Stepinterp()** selects based on the array value being greater than or equal to *X*.

Stepinterp() can be used to perform table lookup.

Library Special

Examples To see the values in *Cost* corresponding to Mpg >= 33:

```
Stepinterp(MPG, Cost, 33, MPG) →
```
Car_type ▶

	VW	Honda	BMW
	1585	2210	2835

Here *X* is an array of values:

```
Stepinterp(MPG, Cost, [28,33], MPG) →
```

	VW	Honda	BMW
28	1705	2330	2955
33	1585	2210	2935

Note: *Example Variables are defined on page 188.*

Other array functions

Concat (*A1, A2, I, J, K*)

Appends array *A2* to array *A1*. *I* and *J* are indexes of *A1* and *A2*, respectively. *K* is the index of the resulting dimension, and usually consists of the list created by concatenating *I* and *J*.

A1 and *A2* must have the same number of dimensions. If they are one-dimensional, the parameters *I*, *J*, and *K* are optional. If they are not specified, the resulting array is unindexed.

If *A1* and *A2* are multidimensional, they must have the same non-concatenated indexes.

Library Array

Examples In addition to the Variables on *page 188*, these examples use the following:

More_years:

1989	1990	1991

All_years:

1985	1986	1987	1988	1989	1990	1991

More_prices: Car_type ▼ *, More_years* ▶

	1989	**1990**	**1991**
VW	11K	12K	12.5K
Honda	15K	15.5K	16.5K
BMW	23.5K	25K	27K

`Concat(Years, More_years)` →
Concat ▶

	1985	1986	1987	1988	1989	1990	1991

Sequence2: `Sequence(1,7)`

`Concat(Years, More_years, Years, More_years, Sequence2)` →
Sequence2 ▶

	1	2	3	4	5	6	7
	1985	1986	1987	1988	1989	1990	1991

`Concat(Car_prices, More_prices, Years, More_years, All_years)` →
All_years ▼, *Car_type* ▶

	VW	Honda	BMW
1985	8000	12K	18K
1986	9000	13K	20K
1987	9500	14K	21K
1988	10K	14.5K	22K
1989	11K	15K	23.5K
1990	12K	15.5K	25K
1991	12.5K	16.5K	27K

IndexNames(A)

Returns a list of the names of the indexes of the array *A*.

Library Array

Example `IndexNames(Car_prices) → ['Car_type','Years']`

Note: *Example Variables are defined on page 188.*

Size (*U*)

Returns the number of array elements of *U*.

Library Array

Examples `Size(Years) → 4`

`Size(Car_prices) → 12`

`Size(10) → 1`

Note: *Example Variables are defined on page 188.*

Matrix functions

Matrix functions perform matrix operations. In Analytica, a ***matrix*** is defined as a two-dimensional array of numbers with indexes of equal length.

Decompose (*C, I, J*)

Returns the Cholesky decomposition (square root) matrix of matrix *C* along dimensions *I* and *J*. Matrix *C* must be symmetric and positive-definite. (Positive-definite means that *v * C * v > 0*, for all vectors *v*.)

Cholesky decomposition computes a lower diagonal matrix *L* such that *L * L′ = C*, where *L′* is the transpose of *L*.

Library Matrix

Example *Matrix*

I ▼, *m* ▶

	1	2	3	4	5
1	6	2	6	3	1
2	2	4	3	1	3
3	6	3	9	3	4
4	3	1	3	8	4
5	1	3	4	4	7

`Decompose(MatrixS,l,m)` →
I ▼, *m* ▶

	1	2	3	4	5
1	2.4495	0	0	0	0
2	0.8165	1.8257	0	0	0
3	2.4495	0.5477	1.6432	0	0
4	1.2247	0	0	2.5495	0
5	0.4082	1.4606	1.3389	1.3728	1.0113

Determinant (*C, I, J*)

Returns the determinant of matrix *C* along dimensions *I* and *J*.

Library Matrix

Example

MatrixA:
j ▼, *i* ▶

	1	2	3
a	4	1	2
b	2	5	3
c	3	2	7

```
Determinant(MatrixA, i, j) → 89
```

EigenDecomp(*A : Numeric [I,J] ; I, J : IndexType*)

Computes the Eigenvalues and Eigenvectors of a square, symmetric matrix `A` indexed by `I` and `J`. `EigenDecomp()` returns a result indexed by `J` and `.item` (where `.item` is a temporary index with two elements: `['value','vector'])`. Each column of the result contains one Eigenvalue/Eigenvector pair. The Eigenvalue is a number, the Eigenvector is a reference to a rows-indexed Eigenvector. If *result* is the result of evaluating `EigenDecomp()`, then the Eigenvalues are given by `result[.item='value']`, and the Eigenvectors are given by `#result[.item='vector']`.
Each Eigenvector is indexed by `I`.

Given a square matrix `A`, a non-zero number (λ) is called an Eigenvalue of `A`, and a non-zero vector `x` the corresponding Eigenvector of `A` when

`A x = λ x`

An *N*x*N* matrix will have *N* (not-necessarily unique) Eigenvalue-Eigenvector pairs. When `A` is a symmetric matrix, the Eigenvalues and Eigenvectors are real-valued. Eigen-analysis is widely used in Engineering and statistics.

Analytica Note: *The matrix* ***A*** *must be square and symmetric. Mathematically, Eigen decompositions do exist for square non-symmetric matrices, but the algorithm used here is limited only to symmetric matrices, since symmetric decompositions are guaranteed to be real-valued, while, in general, Eigen decompositions may be complex.*

Library

Matrix

Example

Convariance Matrix
stock1 ▼, stock2 ▶

	INTC	MOT	AMD
INTC	30.47	13.26	18.9
MOT	13.26	16.58	14.67
AMD	18.9	14.67	17.11

`EigenDecomp( Covariance, Stock1, Stock2 )` →
.item ▼, stock2 ▶

	INTC	MOT	AMD
value	1.025	9.232	53.9
vector	<<ref$_1$>>	<<ref$_2$>>	<<ref$_3$>>

<<ref$_1$>>
stock1 ▼

INTC	0.2845
MOT	0.518
AMD	-0.8067

<<ref$_1$>>
stock1 ▼

INTC	0.6548
MOT	-0.7196
AMD	-0.2312

<<ref$_1$>>
stock1 ▼

INTC	-0.7002
MOT	-0.4625
AMD	-0.5439

Invert (*C, I, J*)

Returns the inversion of matrix *C* along dimensions *I* and *J*.

Library Matrix

Example Set number format to fixed point, 3 decimal digits.

`Invert(MatrixA, i, j)` →
j ▼, *i* ▶

	1	2	3
a	0.326	-0.034	-0.079
b	-0.056	0.247	-0.090
c	-0.124	-0.056	0.202

MatrixMultiply(*A : Numeric all[aRow,aCol] ; aRow, aCol : IndexType ; B : Numeric all[bRow,bCol] ; bRow, bCol : IndexType*)

Performs a matrix multiplication on matrix ***A***, having indexes ***aRow*** and ***aCol***, and matrix ***B***, having indexes ***bRow*** and ***bCol***. The result is indexed by ***aRow*** and ***bCol***. ***A*** and ***B*** must have the specified two indexes, and may also have other

indexes. ***aCol*** and ***bRow*** must have the same length or it flags an error. If ***aRow*** and ***bCol*** are the same index, it returns only the diagonal of the result.

Library Matrix

Example *Matrices*

A x ***B***

index1 ▼ , index2 ▶ index2 ▼ , index3 ▶

	1	2
1	1	2
2	1	0

	a	b	c
1	3	0	1
2	0	1	1

`MatrixMultiply(A,index1,index2,B,index2, index3)` →

index1 ▼ , index3 ▶

	1	2	3
1	3	2	3
2	3	0	1

When the inner index is shared by ***A*** and ***B***, the expression `Sum(A*B,index2)` is equivalent to their dot product (see “Dot product of two matrices” on page 272).

The way to multiply a matrix by its transpose is:

```
MatrixMultiply( A, I, J, Transpose(A,I,J), I, J )
```

It does not work to use `MatrixMultiple(A,I,J,A,J,I)` because the result would have to be doubly indexed by *I*.

SingularValueDecomp(*A, I, J, J2*)

SingularValueDecomp (Singular Value Decomposition) is often used with sets of equations or matrices that are singular or ill-conditioned (that is, very close to singular). It factors a matrix ***A***, indexed by ***I*** and ***J,*** with Size(***I***)>=Size(***J***), into three matrices, **U**, **W**, and **V**, such that

$$\mathbf{A = U \,.\, W \,.\, V'} \qquad \mathbf{(1)}$$

where **U** and **V** are orthogonal matrices and **W** is a diagonal matrix. **U** is dimensioned by ***I*** and ***J***, **W** by ***J*** and ***J2***, and **V** by ***J*** and ***J2***. In Analytica notation:

```
Variable A :=
    Sum(Sum(U*W,J) * Transpose(V,J,J2), J2)
```

The index ***J2*** must be the same size as ***J*** and is used to index the resulting **W** and **V** arrays.

SingularValueDecomp returns an array of three elements indexed by a special system index named **SvdIndex** with each element, **U**, **W**, and **V**, being a reference to the corresponding array. Use the '#' (dereference) operator to obtain the matrix value from each reference, as in:

```
Index J2
Definition: CopyIndex(J)

Variable SvdResult
Definition: SingularValueDecomp(A, I, J, J2)

Variable U
Definition: #SvdResult[SvdIndex='U']

Variable W
Definition: #SvdResult[SvdIndex='W']

Variable V
Definition: #SvdResult[SvdIndex='V']
```

Analytica Note: *Like most other matrix functions,* ***SingularValueDecomp*** *requires its main parameter to be square, and will not work if indexes* ***i*** *and* ***j*** *are not the same size.*

Transpose (*C, I, J*)

Returns the transpose of matrix *C* along dimensions *I* and *J*.

Library Matrix

Example `Transpose(MatrixA, i, j)` → *j* ▼, *i* ▶

	1	2	3
a	4	2	3
b	1	5	2
c	2	3	7

Dot product of two matrices

The dot product (*i.e.*, matrix multiplication) of *MatrixA* and *MatrixB* is equal to

```
Sum(MatrixA * MatrixB, i)
```

Example

`MatrixA` is defined as above.

MatrixB:
k ▼, *i* ▶

	1	2	3
l	3	2	1
m	2	5	3
n	4	1	2

`Sum(MatrixA * MatrixB, i)` →
k ▼, *j* ▶

	a	b	c
l	16	19	20
m	19	38	37
n	21	19	28

Financial functions

These functions can be accessed under the **Definition** menu **Financial** command, or in the Object Finder dialog box, **Financial** library.

Where possible, the function names and parameters match those found in Microsoft Excel.

Parameters

The same parameters occur in many of the financial functions. These parameters are described here. Dollar amounts for both parameters and return values of functions are always expressed as the amount you receive. If you make a payment, the amount is negative. If you receive a payment, the amount is positive.

Rate: The interest rate *per period*. For example, if periods are months, the rate should be adjusted to the monthly rate, not the

annual rate (*e.g.*, `8%/12`, or `1.08^(1/12)-1` with monthly compounding).

Nper: Number of periods in the lifetime of an annuity.

Per: The period (between 1 and Nper) being computed.

Pv: The present value of the annuity. For example, for a loan this is the loan amount (positive if you receive the loan, negative if you are the lender).

Fv: The future value of the annuity. This is the remaining value of the annuity after the final payment. In the case of a loan, for example, this is the balloon payment at the end (positive if you are the lender, negative if you pay the balloon amount). This parameter is usually optional with a default value of zero.

Pmt: The total payment per period (interest + principal). If you receive payments, this is positive. If you make payments, this is negative.

Type: Indicates whether payments are due at the beginning or end of each period.

> **`True`**: Payments are due at the beginning of each period, with the first payment due immediately.
>
> **`False`**: (default) Payments are due at the end of each period.

Cumipmt(*Rate, Nper, Pv, StartPeriod, EndPeriod, Type*)

Returns the cumulative interest paid on an annuity between, and including, **`StartPeriod`** (**`shown as sp in equation below`**) and **`EndPeriod`** (shown as **`ep in equation below`**). The annuity is assumed to have a constant interest rate and periodic payments. This is equal to

$$\sum_{n=sp}^{ep} IPmt(Rate,n,Nper,Pv,0,Type)$$

Example

Interest payments during the first year on a $100,000 loan at 8% is:

```
CumIPmt(8%/12,360,100K,1,12) → -7,969.81
```

The result is negative since these are payments.

Cumprinc(*Rate, Nper, Pv, Start_period, End_Period, Type*)

Returns the cumulative principal paid on an annuity between, and including, **StartPeriod** (**shown as sp in equation below**) and **EndPeriod** (shown as **ep in equation below**). The annuity is assumed to have a constant interest rate and periodic payments. The result is equal to

$$\sum_{n=sp}^{ep} PPmt(Rate,n,Nper,Pv,0,Type)$$

Example

The total principal paid during the first year on a $100,000 loan at 8% is:

```
CumPrinc(8%/12,360,100K,1,12) → -835.36
```

The result is negative since these are payments.

Fv(*Rate, Nper, Pmt, Pv, Type*)

Returns the future value of an annuity investment with constant periodic payments and fixed interest rate. The result is positive if you receive money at the end of the annuity's lifetime, and negative if you must make a payment at the end of the annuity's lifetime.

Examples

You invest $1000 in an annuity that pays 6% annual interest, compounded monthly (0.5% per month), that pays out $50 at the end of each month for 12 months, and then refunds whatever is left after 12 months. The amount refunded is:

```
Fv(0.5%, 12, 50, -1000) → $444.90
```

You borrow $50,000 at a fixed annual rate of 12% (1% per month). You make monthly payments of $550 for 15 years, and

then pay off the remaining balance in a single balloon payment. That final balloon payment is (the negative is because it is a payment for you):

```
-Fv(1%, 15*12, -550, 50000) → $25,020.99
```

You open a fixed-rate bank account that pays 0.5% per month in interest. At the beginning of each month (including when you open the account) you deposit $100. The amount in the account at the end of the each of the first three years is:

```
Fv(0.5%,[12,24,36],-100,0,True) →
        [$1239.72, $2555.91, $3953.28]
```

Ipmt(*Rate, per, Nper, Pv, Fv, Type*)

Returns the interest portion of a payment on an annuity, assuming constant period payments and fixed interest rate.

Example

The interest you pay in the 24th month on a 30-year fixed $100K loan at 8%/12 monthly interest is (the result of `IPmt` is negative since this is a payment for you):

```
-IPmt(8%/12, 24, 12*30, 100K) → $655.59
```

Irr(*Values, I, Guess*)

Returns the Internal Rate of Return of a series of periodic payments (negative values) and inflows (positive values). The IRR is the discount rate at which the Net Present Value (NPV) of the flows is zero. The array `Values` must be indexed by `I`.

If the cash flow never changes sign, `IRR()` will have no solution and returns `NaN` (not a number). If a cash flow changes sign more than once, `Irr()` may have multiple solutions, and will return the first solution found. The implementation uses an iterative gradient-descent search to locate a solution. The optional argument, `Guess`, can be provided as a starting value for the search (default is 10%). When there are multiple solutions, the one closest to `Guess` will usually be returned. If no solution is found within 30 iterations, `Irr()` returns `NaN`.

To compute the IRR for a non-periodic cash flow, use `XIRR()`.

Example

Earnings: Time▶

1999	2000	2001	2002	2003	2004
-1M	-500K	-100K	100K	1M	2M

```
Irr(Earnings,Time) → 17.15%
```

Nper(*Rate, Pmt, Pv, Fv, Type*)

Returns the number of periods of an annuity with constant periodic payments and fixed interest rate.

Example

You invest $10,000 in an annuity that pays 8% annually. Each year you withdraw $1,000. Your annuity will last for

```
NPer(8%,1000,-10K) → 20.91 (years)
```

Npv(*DiscountRate, Values, I*)

Returns the net-present value of a cash flow with equally spaced periods. The **`Values`** parameter contains a series of periodic payments (negative values) and inflows (positive values), indexed by ***I***. Future values are discounted by **`DiscountRate`** per period. The NPV is given by

$$\sum_{j=1}^{n} \frac{Values[I=j]}{(1+DiscountRate)^{j}}$$

Analytica Note: *The first value is discounted as if it is one step in the future. To compute the NPV for a non-periodic cash flow, use* **`Xnpv()`**.

Example

Earnings: Time▶

1999	2000	2001	2002	2003	2004
-1M	-500K	-100K	100K	1M	2M

At a discount rate of 5%, the net present value of the this cash flow is:

```
Npv( 5%, Earnings, Time ) → $865,947.76
```

Pmt(*Rate, Nper, Pv, Fv, Type*)

Returns the total payment per period (interest + principal) for an annuity with constant periodic payments and fixed interest rate.

Example

You obtain a 30-year fixed mortgage at 8%/12 per month for $100K. Your monthly payment will be (note that the result of Pmt is negative since this is a payment for you):

`-Pmt( 8%/12, 30*12, 100K )` → $733.76

Ppmt(*Rate, Per, NPer, Pv, Fv, Type*)

Returns the principal portion of a payment on an annuity with constant period payments and fixed interest rate.

Example

You have a 30-year fixed $100K loan at a rate of 8%/12 monthly. On your 24th payment, the amount of your payment that goes towards principal is (note that the result of `PPmt()` is negative since this is a payment for you):

`-PPmt(8%/12, 24, 12*30, 100K)` → `$78.18`

Pv(*Rate, Nper, Pmt, Fv, Type*)

Returns the present value of an annuity. The annuity is assumed to have constant periodic payments to you of `Pmt` per period for `Nper` periods, with a return of `Rate` per period.

Example

To receive $100 per month from an annuity that returns 6%/12 per month for the next 10 years, you would need to invest (note that the result from `Pv()` is negative since you are paying to make the investment):

`-Pv(6%/12, 10*12, 100)` → $9,007.35

Rate(*NPer, Pmt, Pv, Fv, Type, Guess*)

Returns the interest rate (per period) for an annuity. The value returned is the interest rate that results in equal payments of `Pmt` per period over the `NPer` periods of the annuity.

In general, `Rate()` may have zero or multiple solutions. The implementation uses an interactive search algorithm. The optional `Guess` may be provided as a starting point for the search, which will usually result in the solution closest to `Guess` being returned. If no solution is found in 30 iterations, `Rate()` returns `NaN`.

Example

You obtain a 30-year mortgage at a supposed 7% annual percentage rate for $100K. To do so, you pay $2,000 up front in "points", and another $1,500 in fees. Assuming you hold the loan for its full term, the effective interest rate of your loan (for you) is

`Rate(30,Pmt`(7%,30,100K),100K-3500) → 7.36%

Xirr(*Values, Dates, I, Guess*)

Returns the annual Internal Rate of Return (IRR) for a series of payments (negative values) and inflows (positive values) that occur at non-periodic intervals. Both **Values** and **Dates** must be indexed by **I**. The **Values** array constrains the cash flow amounts, the **Dates** array contains the date of each payment or inflow, where each date is Analytica's expressed as the number of days since Jan 1, 1904. The rate is based on a 365 day year.

If the cash flow never changes sign, there is no solution and **Xirr()** returns **NaN**. If the cash flow changes sign more than once, **Xirr()** may have multiple solutions, but will return only the first solution found. The optional parameter, **Guess**, may be provided as a starting point for the iterative search, and **Xirr()** will generally find the solution closest to **Guess**. If not provided, **Guess** defaults to 10%. If no solution is found within 30 iterations, **Xirr()** returns **NaN**.

To compute the IRR for a series of period payments, use **Irr()**.

Example

EarningAmt: J▶

1	2	3	4
-400K	-200K	100K	600K

EarningDate: J▶

1	2	3	4
July 5, 1999	Dec 1, 1999	Jan 21, 2000	Aug 10, 2001

XIrr(EarningAmt,EarningDate,J) → 9.31%

Analytica Note: *EarningDate can be entered by selecting* ***Number Format*** *from the* ***Result*** *menu while editing the table for EarningDate. From the Number Format Dialog, select a date format, then enter the dates.*

Xnpv(*Rate, Values, Dates, I*)

Returns the Net Present Value (NPV) of a non-periodic cash flow with a constant discount rate. **Rate** is the annual discount rate for a 365 day year. Both **Values**, the cash-flow amounts, and **Dates**, the date of each payment (negative value) or inflow (positive value), must be indexed by **I**.

See also **Npv()**.

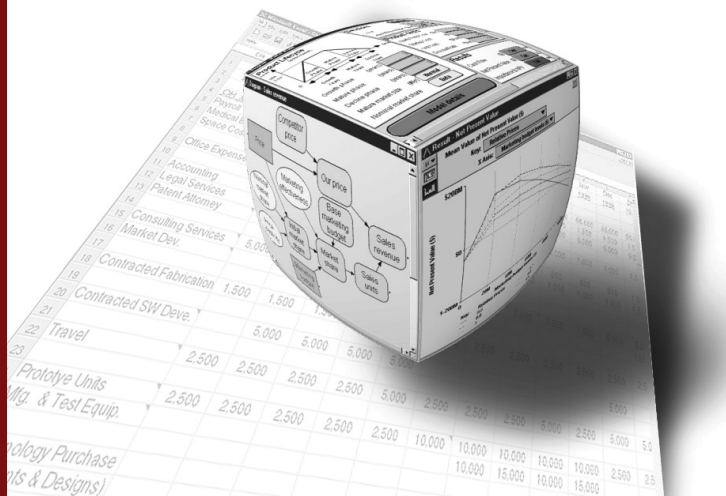
Example

Using the cash flow shown in the example for XIrr() above, the net present value at a 5% discount rate is:

XNpv(5%,EarningAmt,EarningDate,J) → $42,838.71

Chapter 13

Expressing Uncertainty



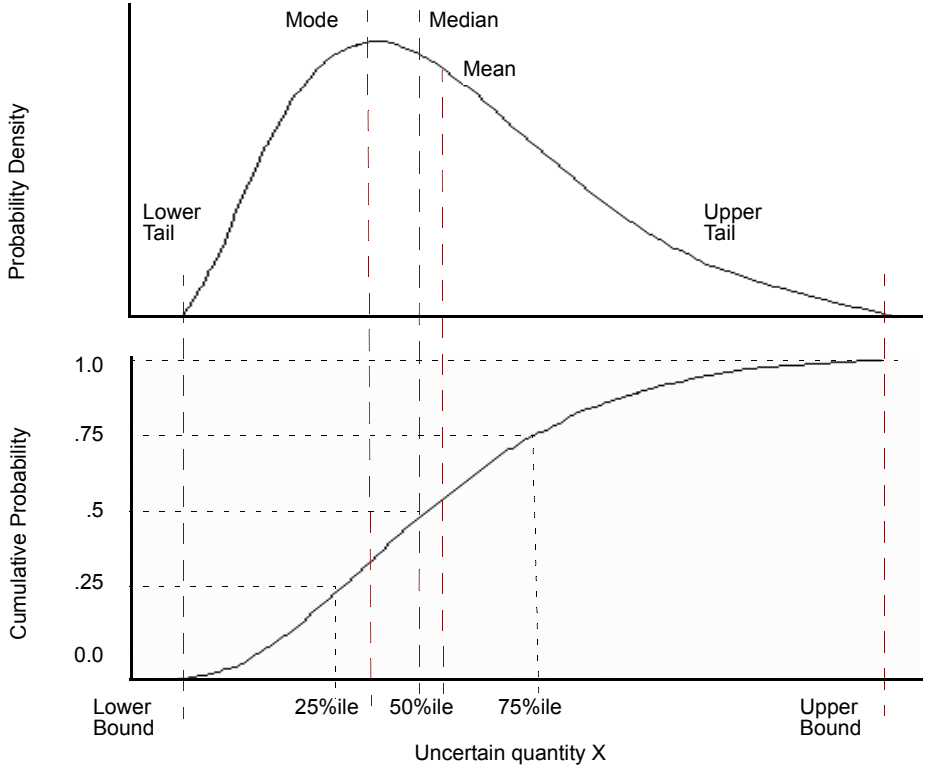
In this Chapter

This chapter shows you how to:

- Choose a distribution
- Define a Variable as a distribution
- Use Analytica's built-in probability distributions

13: Expressing uncertainty

Analytica makes it easy to model and analyze uncertainties even if you have minimal background in probability and statistics. The graphs below review several key concepts from probability and statistics that will help you understand the probabilistic modeling facilities in Analytica. This chapter assumes that you have encountered most of these concepts before, but possibly in the distant past. If you need more information, see the Glossary or refer to an introductory text on probability and statistics.



Choosing an appropriate distribution

With Analytica you can express uncertainty about any Variable by using a probability distribution. You may base the distribution on available relevant data, on the judgment of a knowledgeable individual, or on some combination of data and judgment.

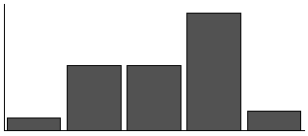
Answer the following questions about the uncertain quantity to select the most appropriate kind of distribution:

- Is it discrete or continuous?
- If continuous, is it bounded?
- Does it have one mode or more than one?
- Is it symmetric or skewed?
- Should you use a standard or a custom distribution?

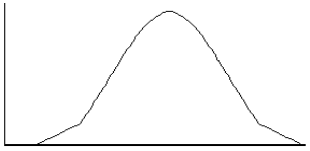
We will discuss how to answer each of these in turn.

Is the quantity discrete or continuous?

When trying to express uncertainty about a quantity, the first technical question is whether the quantity is discrete or continuous.



A ***discrete*** quantity has a finite number of possible values—for example, the gender of a person or the country of a person's birth. ***Logical*** or ***Boolean*** Variables are a type of discrete Variable with only two values, true or false, sometimes coded as yes or no, present or absent, or 1 or 0—for example, whether a person was born before January 1, 1950, or whether a person has ever resided in California.



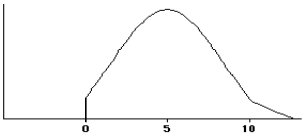
A ***continuous*** quantity can be represented by a real number, and has infinitely many possible values between any two values in its domain. Examples are the quantity of an air pollutant released during a given period of time, the distance in miles of a residence from a source of air pollution, and the volume of air breathed by a specified individual during one year.

For a large discrete quantity, such as the number of humans residing within 50 miles of Disneyland on December 25, 1980, it is often convenient to treat it as continuous. Even though you know that the number of live people must be an integer, you may want to represent uncertainty about the number with a continuous probability distribution.

Conversely, it is often convenient to treat continuous quantities as discrete by partitioning the set of possible values into a small finite set of partitions. For example, instead of modeling human age by a continuous quantity between 0 and 120, it is often convenient to partition people into infants (age < 2 years), children (3 to 12), teenagers (13 to 19), young adults (20 to 40), middle-aged (41 to 65), and seniors (over 65 years). This process is termed

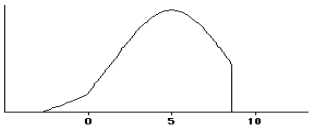
discretizing. It is often convenient to discretize continuous quantities before assessing probability distributions.

Does the quantity have bounds?



If the quantity is continuous, it is useful to know if it is bounded before choosing a distribution—that is, does it have a minimum and/or maximum value?

Some continuous quantities have exact lower bounds. For example, a river flow cannot be less than zero (assuming the river cannot reverse direction). Some quantities also have exact upper bounds. For example, the percentage of a population that is exposed to an air pollutant cannot be greater than 100%.



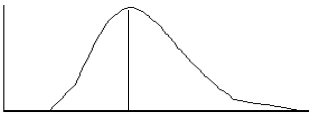
Most real world quantities have *de facto* bounds—that is, you can comfortably assert that there is zero probability that the quantity would be smaller than some lower bound, or larger than some upper bound, even though there is no precise way to determine the bound. For example, you can be sure that no human could weigh more than 5000 pounds; you might be less sure whether 500 pounds is an absolute upper bound.

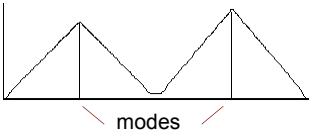
Many standard continuous probability distributions, such as the normal distribution, are unbounded. In other words, there is some probability that a normally distributed quantity is below any finite value, no matter how small, and above any finite value, no matter how large.

Nevertheless, the probability density drops off quite rapidly for extreme values, with near exponential decay, in fact, for the normal distribution. Accordingly, people often use such unbounded distributions to represent real world quantities that actually have finite bounds. For example, the normal distribution generally provides a good fit for the distribution of heights in a human population, even though you may be certain that no person's height is less than zero or greater than 12 feet.

How many modes does it have?

The mode of a distribution is its most probable value. The mode of an uncertain quantity is the value at the highest peak of the density function, or, equivalently, at the steepest slope on the cumulative probability distribution.

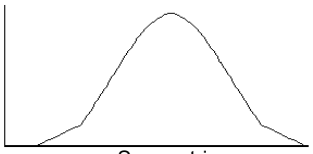


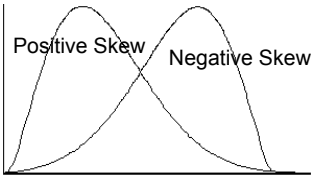


Important questions to ask about a distribution are how many modes it has, and approximately where it, or they, are? Most distributions have a single mode, but some have several and are known as multimodal distributions.

If a quantity has two or more modes, you can usually view it as a combination of two or more populations. For example, the distribution of ages in a daycare center at leaving time might include one mode at age 3 for the children and another mode at age 27 for the parents and caretakers. There is obviously a population of children and a population of parents. It is generally easier to decompose a multimodal quantity into its separate components and assess them separately than to assess a multimodal distribution. You can then assess a unimodal (single mode) probability distribution for each component, and combine them to get the aggregate distribution. This approach is often more convenient, because it lets you assess single-mode distributions, which are easier to understand and evaluate than multimodal distributions.

Is the quantity symmetric or skewed?





A symmetrical distribution is symmetrical about its mean. A skewed distribution is asymmetric. A positively skewed distribution has a thicker upper tail than lower tail; and vice versa, for a negatively skewed distribution.

Probability distributions in environmental risk analysis are often positively skewed. Quantities such as source terms, transfer factors, and dose-response factors, are typically bounded below by zero. There is more uncertainty about how large they might be than about how small they might be.

A standard or custom distribution?

The next question is whether to use a standard parametric distribution—for example, normal, lognormal, or beta—or a custom distribution, where the assessor specifies points on the cumulative probability or density function.

Considering the physical processes that generate the uncertainty in the quantity may suggest that a particular standard distribution is appropriate. More often, however, there is no obvious standard distribution to apply.

It is generally much faster to assess a standard distribution than a full custom distribution, because standard distributions have

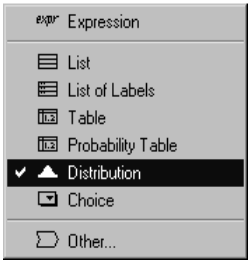
fewer parameters, typically from two to four. You should usually start by assigning a simple standard distribution to each uncertain quantity using a quick judgment based on a brief perusal of the literature or telephone conversation with a knowledgeable person. You should assess a custom distribution only for those few uncertain inputs that turn out to be critical to the results. Therefore, it is important to be able to select an appropriate standard distribution quickly for each quantity.

Defining a Variable as a distribution

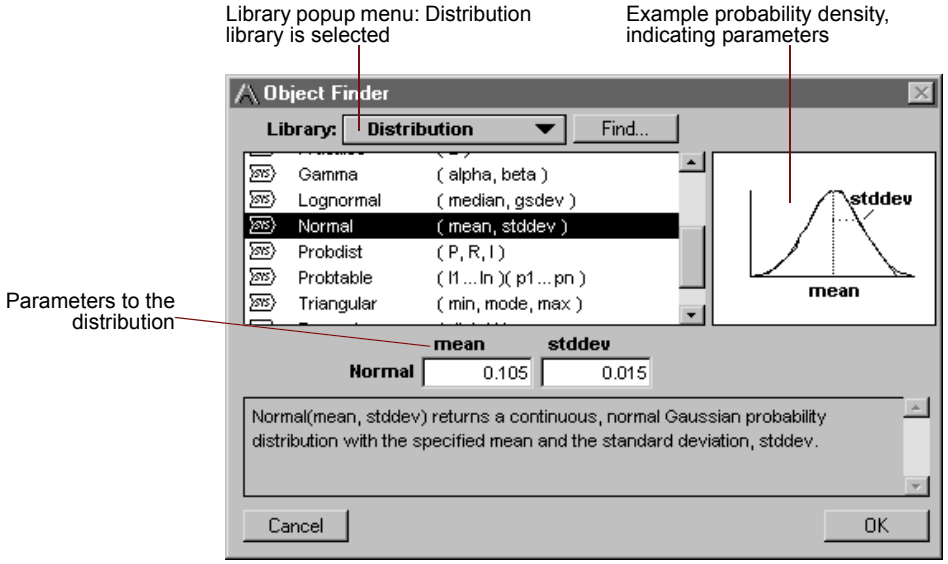
To define a Variable as an Analytica probability distribution, first select the Variable and open either the Variable's Object window or the Attribute panel of the diagram (see "The Attribute panel" on page 34) with **Definition** selected from the Attribute popup menu (see "The Attribute popup menu" on page 35).

To define the distribution:

1. Click on the Expression popup menu above the definition field and select **Distribution**.

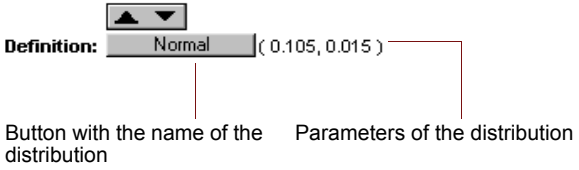


The Object Finder opens, showing the Distribution library.



2. Select the distribution you wish to use.
3. Enter the values for the parameters. You can use an expression or refer to other Variables by name in the parameter fields.
4. Click on **OK** to accept the distribution.

If the parameters of the distribution are single numbers, a button appears with the name of the distribution, indicating that the Variable is defined as a distribution. To edit the parameters, click on this button.



If the parameters of the distribution are complex expressions, the distribution displays as an expression. For example,

```
Normal((Price/Mpy) * Mpg, Mpg/10)
```

Entering a distribution as an expression

Alternatively, you can directly enter a distribution as an expression:

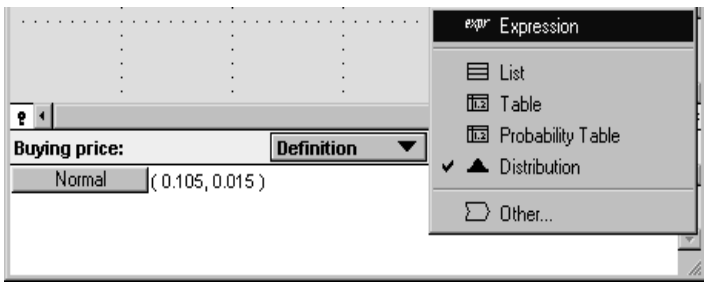
1. Set the cursor in the definition field and type in the distribution name and parameters, *e.g.*

   ```
   Normal(.105,0.015)
   ```

2. Press *Alt-enter* or click on the ☑ button.

You can also paste a distribution from the Distribution library in the **Definition** menu (see “Pasting from a library in the Definition menu” on page 155).

You can edit a distribution as an expression, whether it was entered as a distribution from the Distribution library or as an expression, by selecting **expr** from the Expression popup menu.



Including a distribution in a definition

You can enter a distribution anywhere in a definition, including in a cell of an Edit Table. Thus, you can have arrays of distributions.

To enter a distribution:

1. Set the insertion point where you wish to enter the distribution in the definition field or Edit Table cell.
2. Enter the distribution in any of the following ways:
   - Type in the name of the distribution.
   - Paste it from the from the **Distribution** Library under the **Definition** menu.
   - Select **Paste Identifier** from the **Definition** menu to paste it from the **Object Finder.**

3. Type in missing parameters, or replace parameters enclosed as <<x>>.

Probabilistic calculation

Analytica performs probabilistic evaluation of probability distributions through simulation—by computing a random sample of values from the actual probability distribution for each uncertain quantity. The result of evaluating a distribution is represented internally as an array of the sample values, indexed by *Run*. *Run* is an index Variable that identifies each sample iteration by an integer from 1 to *Samplesize*.

You can display a probabilistic value using a variety of uncertainty view options—the mean, statistics, probability bands, probability density (or mass function), and cumulative distribution function (see “Uncertainty view options” on page 52). All these views are derived or estimated from the underlying sample array, which you can inspect using the last uncertainty view, Sample.

Example

A: `Normal(10,2)` →

Iteration (Run)▶

	1	2	3	4	5	6
	10.74	13.2	9.092	11.44	9.519	13.03

Analytica Note: *The values in a sample are generated at random from the distribution; if you try this example and display the result as a table, you may see values different from those shown here. To reproduce this example, reset the random number seed to 99 and use the default sampling method and random number method (see “Uncertainty Setup dialog box” on page 291).*

For each sample run, a random value is generated from each probability distribution in the model. Output Variables of uncertain Variables are calculated by calculating a value for each value of *Run*.

Example

B: `Normal(5,1)` →

Iteration (Run)▶

	1	2	3	4	5	6
	5.09	4.94	4.65	6.60	5.24	6.96

C: A `+ B` →

Iteration (Run) ▶

	1	2	3	4	5	6
	15.83	18.13	13.75	18.04	14.76	19.99

Notice that each sample value of *C* is equal to the sum of the corresponding values of *A* and *B*.

To control the probabilistic simulation, as well as views of probabilistic results, use the Uncertainty Setup dialog box (see “Uncertainty Setup dialog box” on page 291).

Analytica Note: *If you try to apply an array reducing function (see “Array-reducing functions” on page 247) to a probability distribution across Run, Analytica returns the distribution's Mid value.*

Example:
X: `Beta(2,3)`
`Mid(X)` → `0.3857` and `Max(X,Run)` → `0.3857`
To evaluate the input parameters probabilistically and reduce across *Run*, use `Sample()` (see page 340).

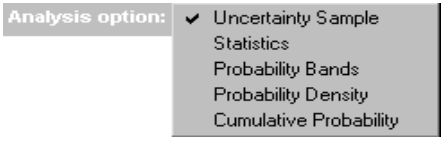
Example:
`Max(Sample(X),Run)` → `0.8892`

Uncertainty Setup dialog box

Use the Uncertainty Setup dialog box to inspect and change the sample size, sampling method, statistics, probability bands, and samples per plot point for probability distributions. All settings are saved with your model.

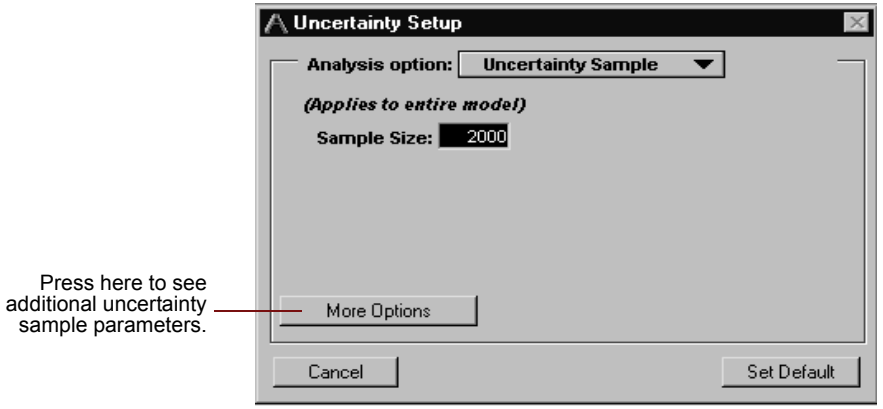
To open the Uncertainty Setup dialog box, select **Uncertainty Options...** from the **Result** menu or Ctrl-U. To set values for a specific Variable, select the Variable before opening the dialog box.

The five options for viewing and changing information in the Uncertainty Setup dialog box can be accessed using the Analysis option popup menu.



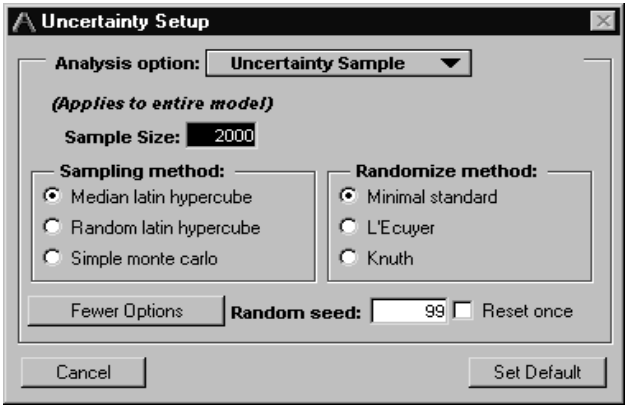
Uncertainty Sample

To change the sample size or sampling method for the model, select the **Uncertainty Sample** option from the **Analysis options** popup menu.



Press here to see additional uncertainty sample parameters.

The default dialog box shows only a field for sample size. To view and change the sampling method, random number method, or random seed, press the **More Options** button.



Sample size

This number specifies how many runs or iterations Analytica performs to estimate probability distributions. Larger sample sizes take more time and memory to compute, and produce smoother distributions and more precise statistics. See “Selecting the sample size” on page 499 for guidelines on selecting a sample size.

The sample size must be between 2 and 32,000. You can access this number in expressions in your models as the system Variable *Samplesize*.

Sampling method

The sampling method is used to determine how to generate a random sample of the specified sample size, *m*, for each uncertain quantity, *X*. Analytica provides three options:

Simple Monte Carlo

The simplest sampling method is known as Monte Carlo, named after the randomness prevalent in games of chance, such as at the famous casino in Monte Carlo. In this method, each of the *m* sample points for each uncertainty quantity, *X*, is generated at random from *X* with probability proportional to the probability density (or probability mass for discrete quantities) for *X*. Analytica uses the inverse cumulative method; it generates *m* uniform random values, u_i for *i*=1,2,...*m*, between 0 and 1, using the specified random number method (see below). It then uses the inverse of the cumulative probability distribution to generate the corresponding values of *X*,

Xi where P($x \le Xi$) = u_i for *i*=1,2,...m.

With the simple Monte Carlo method, each value of every random Variable *X* in the model, including those computed from other random quantities, is a sample of *m* independent random values from the true probability distribution for *X*. You can therefore use standard statistical methods to estimate the accuracy of statistics, such as the estimated mean or fractiles of the distribution, as for example described in “Selecting the sample size” on page 499

Median Latin hypercube (the default method)

With median Latin hypercube sampling, Analytica divides each uncertain quantity *X* into *m* equiprobable intervals, where *m* is the sample size. The sample points are the medians of the *m* intervals, that is, the fractiles

Xi where P(**(**$x \le Xi$) = (*i*-0.5)/*m*, for *i*=1,2,...*m*.

These points are then randomly shuffled so that they are no longer in ascending order, to avoid nonrandom correlations among different quantities.

Random Latin hypercube

The random Latin hypercube method is similar to the median Latin hypercube method, except that instead of using the median of each of the *m* equiprobable intervals, Analytica samples at random from each interval. With random Latin hypercube sampling, each sample is a true random sample from the distribution. However, the samples are not totally independent.

Choosing a sampling method

The advantage of Latin hypercube methods is that they provide more even distributions of samples for each distribution than simple Monte Carlo sampling. Median Latin hypercube is still more evenly distributed than random Latin hypercube. If you display the PDF of a Variable that is defined as a single continuous distribution, or is dependent on a single continuous uncertain Variable, using median Latin hypercube sampling, the distribution will usually look fairly smooth even with a small sample size (such as 20), whereas the result using simple Monte Carlo will look quite noisy.

If the Variable depends on two or more uncertain quantities, the relative noise-reduction of Latin hypercube methods is reduced. If the result depends on many uncertain quantities, the performance of the Latin hypercube methods may not be discernibly better than simple Monte Carlo. Since the median Latin hypercube method is sometimes much better, and almost never worse than the others, Analytica uses it as the default method.

Very rarely, median Latin hypercube can produce incorrect results, specifically when the model has a periodic function with a period similar to the size of the equiprobable intervals. For example, with

X: `Uniform(1, Samplesize)`

Y: `Sin(2*Pi*X)`

median Latin hypercube method will give very poor results. In such cases, you should use random Latin hypercube or simple Monte Carlo. If your model has no periodic function of this kind, you do not need to worry about the reliability of median Latin hypercube sampling.

Random number method

The random number method is used to determine how random numbers are generated for the probability distributions. Analytica provides three different methods for calculating a series of pseudorandom numbers.

Minimal Standard (the default method)

The Minimal Standard random number generator is an implementation of Park and Miller's Minimal Standard (based on a multiplicative congruential method) with a Bays-Durham shuffle. It gives satisfactory results for less than 100,000,000 samples.

L'Ecuyer

The L'Ecuyer random number generator is an implementation of L'Ecuyer's algorithm, based on a multiplicative congruential method, which gives a series of random numbers with a much longer period (sequence of numbers that repeat). Thus, it provides good random numbers even with more than 100,000,000 samples. It is slightly slower than the Minimal Standard generator.

Knuth

Knuth's algorithm is based on a subtractive method rather than a multiplicative congruential method. It is slightly faster than the Minimal Standard generator.

Random seed

This value must be a number between 0 and 100,000,000 (10^8). The series of random numbers starts from this seed value when:

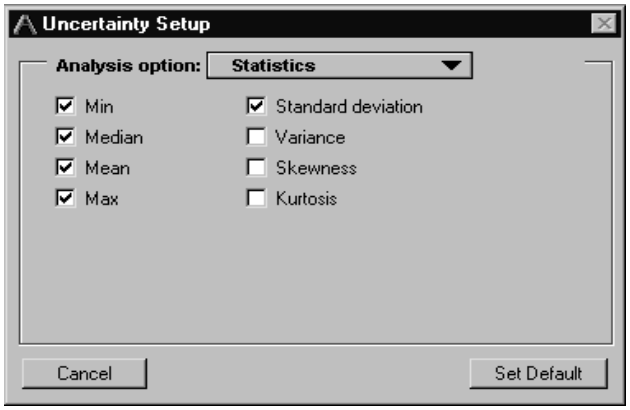
- A model is opened
- The value in this field is changed
- The Reset once box is checked, and the Uncertainty Setup dialog box is closed by clicking on the Accept or Set Default button.

Reset once

Check the Reset once box to produce the exact same series of random numbers.

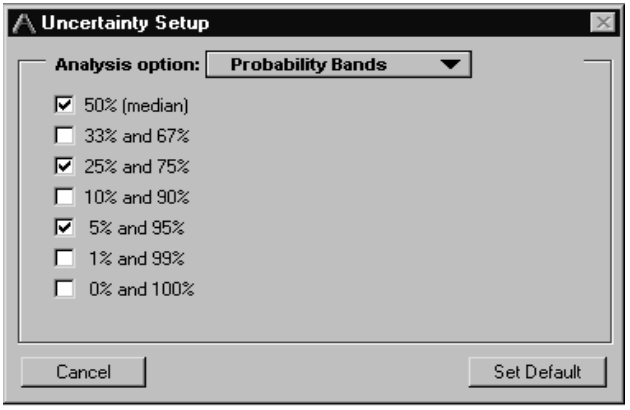
Statistics option

To change the statistics reported when you select **Statistics** as the uncertainty view for a result, select the **Statistics** option from the **Analysis option** popup menu.



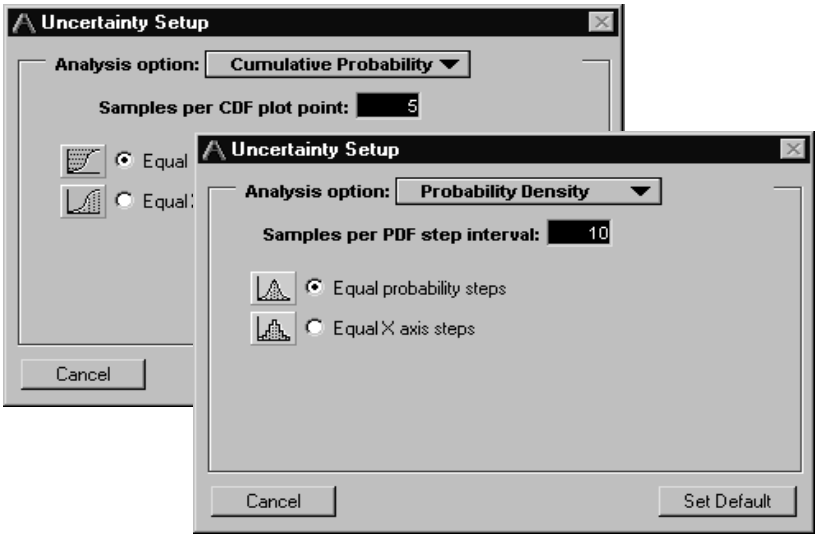
Probability Bands option

To change the probability bands displayed when you select **Probability Bands** as the uncertainty view for a result, select the **Probability Bands** option from the **Analysis option** popup menu.



Probability density and cumulative probability options

To change how probability density or the cumulative probability values are drawn or to change their resolution, select the respective option from the **Analysis option** popup menu.



Analytica estimates the probability density function and cumulative distribution function, like other uncertainty views, from the underlying array of sample values for each uncertain quantity. As with any simulation-based method, each estimated distribution will have some noise and variability from one evaluation to the next.

Samples per plot point

This number controls the average number of sample values used to estimate each point on the probability density function (PDF) or cumulative distribution function (CDF) curves.

For a small number of samples per plot point (less than or equal to 10), more points are each estimated from fewer sample values and so are more susceptible to random noise. If the quantity is defined by a single probability distribution, and if you use median Latin hypercube method (the default), this noise will be slight and the curve will look smooth. In other cases, the noise may have a large effect, and using a larger number of samples per plot point will produce a smoother curve. There is a trade-off; with larger

numbers the smoothing may miss details of the shape of the curve. PDFs may be much more susceptible to random noise than CDFs, so you may wish to use larger numbers for PDFs than CDFs. Ultimately, to reduce the noise, use a larger sample size (for details on selecting the sample size, see Appendix A, “”).

Equal probability steps

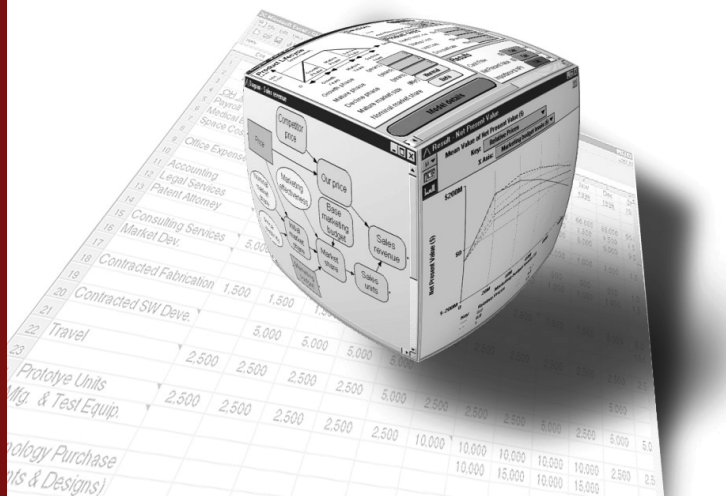
With this option, Analytica uses the sample to estimate a set of *m+!* fractiles (quantiles), X_p, at equal probability intervals, where p=0, q, 2q, ... 1, and q = 1/m. The cumulative probability is plotted at each of the points X_p, increasing in equal steps along the vertical axis. Points are plotted closer together along the horizontal axis in the regions where the density is the greatest. In the probability density graph view, the areas under the density function between successive fractiles are equal because they each represent the same probability, q. The density between two successive fractiles is plotted at the mid point (on the horizontal axis) of the two fractiles.

Equal X axis steps

With this option, Analytica estimates cumulative probability using equally spaced points along the X axis. In the probability density graph view, it shows a histogram where the height of each horizontal is estimated as the fraction of the sample values that fall within that X interval.

Chapter 14

Probability Distributions



In this Chapter

This chapter shows you how to:

- Use Analytica's built-in continuous probability distributions
- Use Analytica's discrete probability table functions
- Use Analytica's discrete probability distributions

14: Probability distributions

This chapter describes how to use probability distributions in Analytica, both discrete and continuous probability distributions. For either type of probability distribution, Analytica allows you to use either a well-defined parametric distribution (such as a Normal or Uniform distribution) or to create a custom distribution.

Built-in probability distributions

Analytica has the following built-in probability functions in the distribution library. The continuous functions are described starting at “Parametric Continuous distributions” on page 313 and the discrete functions are described starting at “Parametic discrete distributions” on page 302.

Parametric Discrete

- `Bernoulli()` page 302
- `Binomial()` page 303
- `Geometric()` page 304
- `Hypergeometric()` page 304
- `Poisson()` page 304

Parametric Continuous

- `Beta()` page 313
- `Certain()` page 315
- `Chisquared()` page 315
- `Exponential()` page 316
- `Gamma()` page 316
- `Logistic()` page 317
- `Lognormal()` page 318
- `Normal()` page 319
- `StudentT()` page 320
- `Triangular()` page 321
- `Uniform()` page 322
- `Weibull()` page 322

Custom Discrete

- `Probtable()` page 307
- `Determtable()` page 312
- `Chancedist()` page 312

Custom Continuous

- `Cumdist()` page 323
- `Fractiles()` page 324
- `Probdist()` page 325
- `Truncate()` page 326

Multivariate

- `Binormal()` page 327
- `Correlate_Dists()` page 328
- `Correlate_With()` page 328
- `Dirichlet()` page 328
- `Gaussian()` page 328
- `Multinomial()` page 328
- `SampleCovariance()` page 329

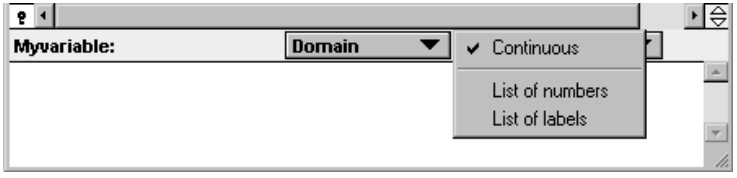
Parametic discrete distributions

To define a Variable as a discrete probability distribution other than probability table and view its Probability Mass Function, you must first set its domain type and then assign the range of possible outcomes to its domain.

Analytica Note: *When sample points are numeric, it can be ambiguous as to whether a Variable is discrete or continuous. For example, Poisson(15) will produce integer-numeric samples. If you define a domain, that tells Analytica that it is to be treated as discrete. Otherwise, Analytica guesses. If it guesses wrong (and graphs a probability density, rather than a probability mass), you can change this by changing the graph type to a solid bar graph (for discrete), or to a histogram (for continuous).*

To set a discrete domain type and assign domain values:

1. Select the Variable and open the Attribute panel of the Diagram window (see “The Attribute panel” on page 34).
2. Select the **Domain** Attribute from the popup menu.
3. The domain type popup menu shows the default of Continuous. Select either List of numbers or List of Labels.



4. Analytica displays a list containing one element. Enter the domain values like any list (see “Creating a list” on page 216).

Analytica Note: *The domain must include all the values that appear in the sample of the discrete probability distribution. If it does not, then the total Probability Mass Function will be less than 1.*

Bernoulli (*P*)

Creates a discrete probability distribution with probability *P* of result 1 and probability (1 - *P*) of result 0. *P* is a probability value

or array of probabilities, each between 0 and 1. The Bernoulli distribution is defined as:

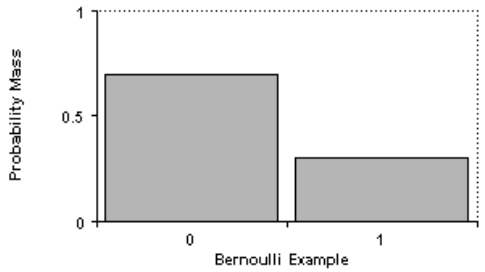
```
If Uniform(0, 1) < P Then 1 Else 0
```

If *P* is greater than 1, the distribution is made up of all 1's. If *P* is less than 0, the distribution is made up of all 0's.

Library Distribution

Example The domain, List of numbers, is [0, 1].

Bernoulli_ex: `Bernoulli (0.3)` →



Binomial(*n, p*)

Consider an event—such as a coin coming down heads—that can be true or false in each trial—or each toss—with probability ***p***—it has a Bernoulli distribution. A binomial distribution describes the number of times an event is true—e.g., the coin is heads —in ***n*** independent trials—or tosses—where the event occurs with probability ***p*** on each trial.

The relationship between the Bernoulli and binomial distributions means that an alternative, if less efficient, way to define a Binomial distribution function would be:

```
Function Binomial2(n, p)
Parameters: (n: Atomic; p)
Definition: Index i := 1..n;
    Sum(FOR J := I DO Bernoulli(p), i)
```

The parameter `n` is qualified as `Atomic` to ensure that the sequence `1..n` is a valid one-dimensional index value. It allows `Binomial2` to array abstract if its parameters `n` or `p` are arrays. See page 452 for details.

Geometric(*p*)

The geometric distribution describes the number of independent Bernoulli trials until the first successful outcome occurs—for example, the number of coin tosses until the first heads. The parameter ***p*** is the probability of success on any given trial.

Hypergeometric(*s, m, n*)

The hypergeometric distribution describes the number of times an event occurs in a fixed number of trials without replacement—e.g., the number of red balls in a sample of ***s*** balls drawn without replacement from an urn containing ***n*** balls of which ***m*** are red. Thus, the parameters are:

s: The sample size—e.g., the number of balls drawn from an urn without replacement. Cannot be larger than ***n***.

m: The total number of successful events in the population—e.g, the number of red balls in the urn.

n: The population size—e.g., the total number of balls in the urn, red and non-red.

Poisson(*m*)

A Poisson process generates random independent events with a uniform distribution over time and a mean of ***m*** events per unit time. **Poisson**(***m***) generates the distribution of the actual number of events that occur in one unit of time. You might use the Poisson distribution to model the number of sales per month of a low-volume product, or the number of airplane crashes per year.

Custom Discrete probabilities

Probability Tables

To describe a Variable as a discrete uncertainty, Analytica provides a special kind of Edit Table called a ***probability table***. (See also “11: Arrays and indexes” on page 207.)

Creating a probability table

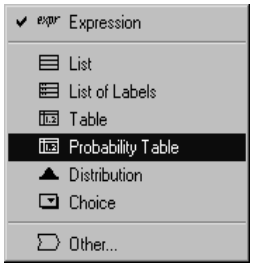
To define a Variable as a discrete probability distribution in a probability table:

1. Determine the Variable's ***domain***—the list of possible outcomes.
2. Select the Variable and open one of the following:

   The Variable's Object window.

   The Attribute panel of the Diagram window (see "The Attribute panel" on page 34).

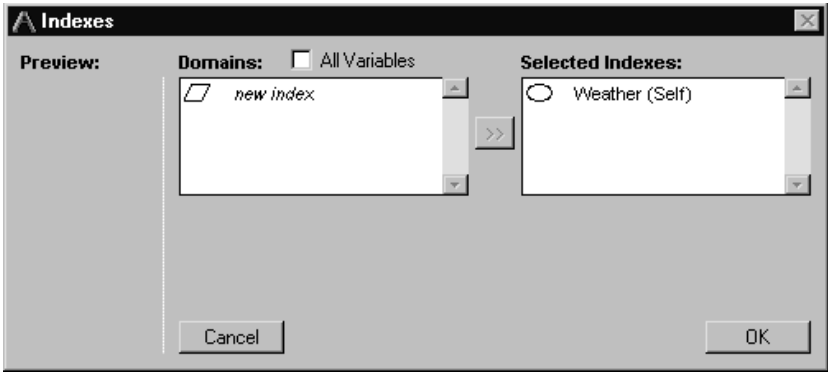
   In the Attribute panel, select **Definition** from the Attribute popup menu (see "The Attribute popup menu" on page 35) as the Attribute to display.
3. Click on the Expression popup menu above the definition field and select **Probability Table**.



   If the Variable already has a definition, a dialog box confirms that you wish to replace it.

Analytica Note: *If the definition of a Variable is already a probability table, a* **ProbTable** *button appears in the definition. Click on it to see the Edit Table window (see "Viewing an array as an Edit table" on page 209).*

4. The Indexes dialog box opens to confirm your choices for the indexes of the table. Only Variables with a domain of List of numbers or List of labels are shown by default. The Variable being defined is already listed as a selected index (with *Self* in parentheses). Add or remove any other discrete inputs (or other Index Variables).

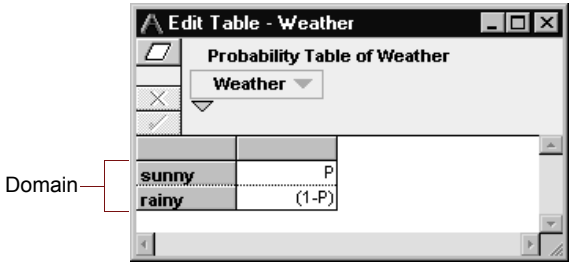


Analytica Note: *Self is required as an index of a probability table. It refers to this Variable's domain values.*

5. Click on the **OK** button. An Edit Table window appears.
6. Enter the possible outcomes (the domain) in the first column. If the outcomes are numeric, they must be in increasing order.
7. Enter the probability of each possible outcome in the second column. (The probabilities should sum to 1.)

Example

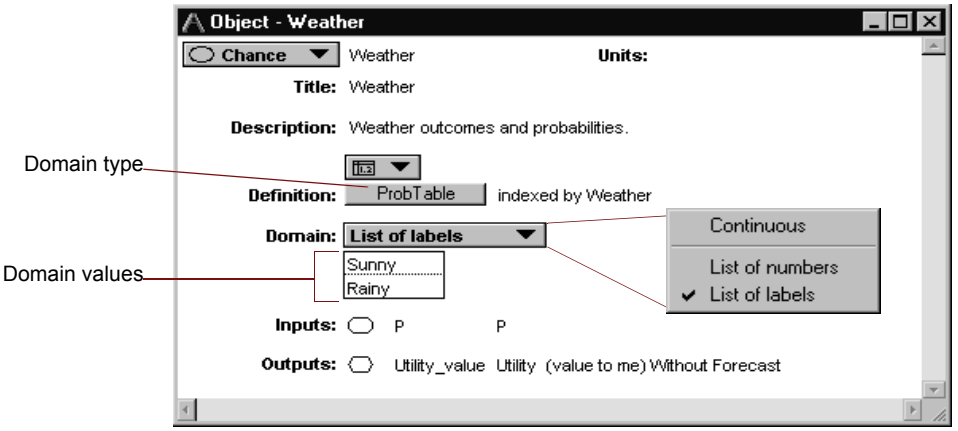
If *P* is a Variable whose value is a probability (between 0 and 1) and the possible weather outcomes are sunny and rainy, then the following is a probability table for weather:



Editing the domain

The domain Attribute has values (the possible outcomes) and a type. In a probability table, you can edit the values directly in the first column of the Edit Table window, as an index of the table. Each entry must be a number or label (text); it cannot be an expression.

You can also edit the domain values in the Object window and Attribute panel.



Changing the domain type

The domain popup menu shows the domain type. For a probability table, the domain type is either a list of numbers or a list of labels and is set by your entry in the first row's cell in the Edit Table.

To change the domain type, press on the popup menu and select the desired type.

Expression view of probability table

When you select the expression view of a definition that was created as a probability table, it has the following appearance. You cannot create a probability table as an expression.

Probtable (*I1, I2, … In*) (*p1, p2, p3, … pm*)

Describes an *n*-dimensional conditional probability table, indexed by the indexes *I1, I2, … In*. One index must be *Self*.

p1, p2, p3, … pm are the probabilities in the array.

Example

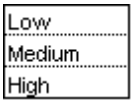
The *Weather* probability table on page 306, when viewed as an expression, looks like this:

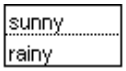
```
Probtable (Self)(P, (1-P))
```

The domain values do not appear in the expression view.

Using labels in a probability table

A discrete probability distribution can describe the probability that a Variable falls into a category. For example:





In a probability table, Analytica assumes that label outcomes are ordered, with the first value being the minimum and the last value being the maximum. In the first example above, this ordering has meaning; in the second example above, it does not. This ordering is used to compute the following statistics. Use these statistics with caution, since they are a function of the order sequence of the qualitative outcomes.

Statistics available for label valued distributions

`Frequency (use Frequency(X,X))`
`Mid Value (Median)`
`Min`
`Max`
`Probability bands`
`Sample`

Statistics *not* available for label valued distributions

`Correlation`
`Kurtosis`
`Getfract`
`Mean`
`Rankcorrel`
`Skewness`
`Standard deviation`
`Variance`

Also use caution applying the logical operators (>,<,=) to a label valued distribution. The logical operators use the ASCII sort sequence, not the ordering of label outcomes.

Adding dimensions to a probability table

You may wish to add dimensions to a probability table. For example, in the *Weather* probability table (see page 306), you may wish to distinguish between daylight and evening, with different probabilities for rainy weather in daylight and evening. So you would add a dimension with two values: daylight and evening.

You can add indexes or decision Variables defined as lists, similar to adding indexes to an Edit Table, as follows:

1. Open the Edit Table window by clicking on the **ProbTable** button.
2. Click on the Indexes ( ) button to open the Indexes dialog box.
3. Click in the **All Variables** check box above the left hand list.
4. Move the desired Variables to add them as indexes.
5. Click on the **OK** button to accept the changes.

Creating a conditional dependency

After you have defined several probability tables, you may want to make some probabilities in a probability table conditional on the outcomes of other Variables. This is called a ***conditional dependency***.

To create a conditional dependency in a probability table:

1. Open the Edit Table window by clicking on the **ProbTable** button.
2. Click on the Indexes ( ) button. Other Variables that are defined as probability tables appear in the list of domains.
3. Move the Variables you wish to be conditionally dependent, to add them as indexes.
4. Click on the **OK** button to accept the changes.

The resulting table is indexed by both the domain of your Variable and the domains of the conditionally dependent Variables.

Analytica Note: *You must have already specified the Variables as probability tables, before adding them with the Indexes dialog box.*

Deterministic conditional tables

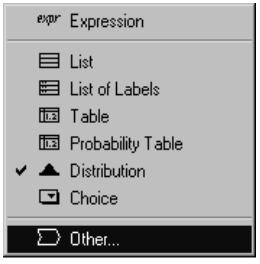
Sometimes a Variable's value is deterministic (not uncertain) and conditionally dependent on the outcomes of discrete uncertain Variables. The `Determtable()` function defines this dependency.

The `Determtable()` function appears similar to an Edit Table or a probability table. Each cell contains non-probabilistic (deterministic) values. At least one index is a probability table (a discrete probabilistic Variable). Other indexes are typically decision Variables defined as lists. The `Determtable()` function returns an array that is reduced across its probabilistic index(es). The evaluation result shows the value considering the uncertain distribution of each probabilistic index.

Creating a determtable

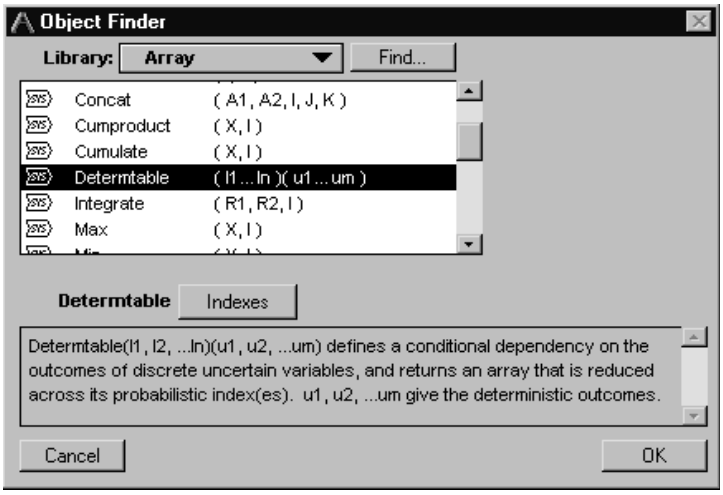
To define a Variable as a determtable:

1. Determine the Variable's domain—the list of possible outcomes.
2. Press the Expression popup menu above the definition field and select **Other**.

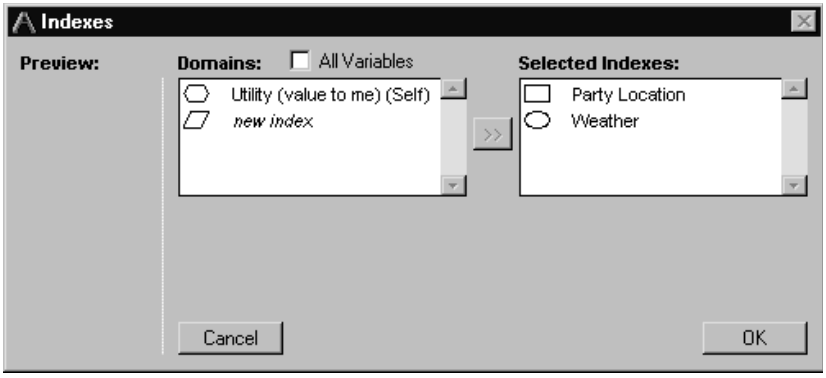


Analytica opens the Object Finder dialog box (see "Object Finder dialog box" on page 152).

3. Select **Array** from the **Library** popup menu and select **Determtable** from the function list.



4. Click on the **Indexes** button to specify discrete probability Variables as inputs. The Indexes dialog box appears.



5. Click on **OK** to accept the indexes and open an Edit Table window.
6. Enter the outcomes corresponding to each outcome of your discrete inputs.

Expression view of a determtable

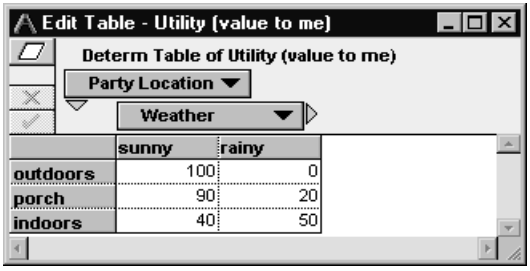
When you select the expression view of a definition that was created as a determtable, it has the following appearance. You cannot initially create a determtable as an expression.

Determtable(*I1, I2, … In*) (*r1, r2, r3, … rm*)

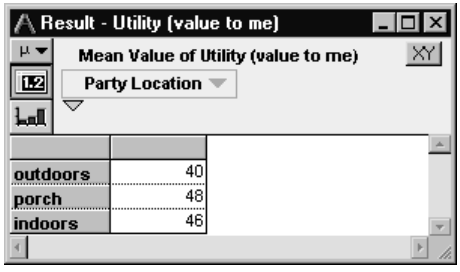
Describes an *n*-dimensional conditional deterministic table, indexed by the indexes *I1, I2, … In*. The last index, *In*, is the innermost index, varying the most rapidly. *r1*, *r2, … rm* are the outcomes in the array. Determtable returns an array that is reduced across its indexes that are probability tables.

Example

In “Creating a probability table” on page 304, *Weather* is defined as a probability table. If *P*, the probability of "sunny", is 0.4, then the probability of "rainy" is 0.6. *Party location* is a decision Variable with values `['outdoors', 'porch','indoors']`. *Value to Me* is a determtable, containing utility values (or "payoffs") for each combination of *Party location* and *Weather*:



Evaluating *Value to Me* gives the value of each party location, considering the uncertain distribution of *Weather*. The mean value of *Value to Me* is the expected utility.



Chancedist (*P, A, I*)

Creates a discrete probability distribution. *A* is an array of outcomes, and *P* is the corresponding array of probabilities. *A* and *P* must both be indexed by *I.*

The values of *A* must be unique; if *A* is numeric the values must be increasing.

When to use

Use `Chancedist()` instead of the probability table when:

- The array of outcomes *A* is multidimensional, or
- The outcomes and probabilities arrays are defined as other Variables; the Variables can be used in other parts of your model.

Library

Distribution

Example

Index_b:

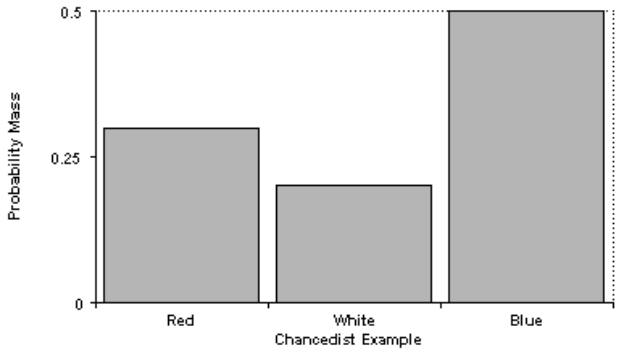
Red	White	Blue

Array_q:
Index_b ▶

	Red	White	Blue
	0.3	0.2	0.5

The domain, List of labels, is ['Red','White','Blue'].

`Chancedist(Array_q,Index_b,Index_b)` →



Parametric Continuous distributions

Analytica Note: *To reproduce the continuous distribution graphs in this chapter, use a sample size of 1000.*

Beta (*X, Y, lower, upper*)

Creates a distribution of numbers between 0 and 1 with $\frac{X}{(X+Y)}$ representing the mean, if the optional parameters *lower* and

upper are omitted. For bounds other than 0 and 1, specify the optional *lower* and *upper* bounds to offset and expand the distribution.

X and *Y* must be positive.

When to use

Use a beta distribution if the uncertain quantity is bounded by 0 and 1 (or 100%), is continuous, and has a single mode. This distribution is particularly useful for modeling an opinion about the fraction of a population that has some characteristic. For example, if you have observed *n* members of the population, of which *r* display the characteristic *c*, you can represent the uncertainty about the true fraction with *c* using a beta distribution with parameters *X* = *r* and *Y* = *n* - *r*.

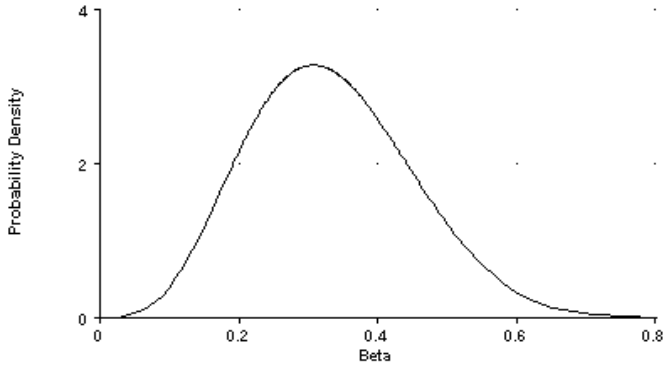
If the uncertain quantity has lower and upper bounds other than 0 and 1, include the lower and upper bounds parameters to obtain a ***transformed beta*** distribution. The transformed beta is a very flexible distribution for representing a wide variety of bounded quantities.

Library

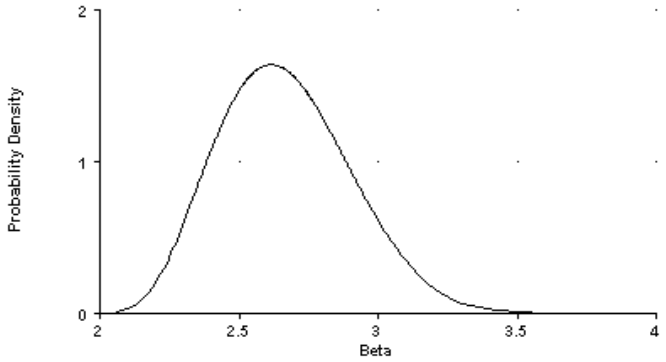
Distribution

Examples

```
Beta (5, 10) →
```



Beta (5, 10, 2, 4) →



Certain (*U*)

Returns the value of *U*.

Library Distribution

When to use Use `Certain()` when an input node is defined as a distribution (see “Using input nodes” on page 161), and, in browse mode, you want to replace the distribution with a non-probabilistic value.

Example *Index_a:*

1	2	3

Array_p:
Index_a ▶

	1	2	3
	0.3	0.4	0.3

Certain (Array_p) →

	1	2	3
	0.3	0.4	0.3

ChiSquared (*d*)

The `ChiSquared` distribution with ***d*** degrees of freedom describes the distribution of a Chi-Squared metric defined as

$$Chi^2 = \sum_{i=1}^{n} y_i^2$$

where each y_i is independently sampled from a standard normal distribution and ***d = n -1*** . The distribution is defined over non-negative values.

The Chi-squared distribution is commonly used for analyses of second moments, such as analyses of variance and contingency table analyses. It can also be used to generate the F distribution. Suppose

```
Variable V := ChiSquared(k)
Variable W := ChiSquared(m)
Variable S := (V/k)*(W/m)
```

s is distributed as an F distribution with **k** and **m** degrees of freedom. The F distribution is useful for the analysis of ratios of variance, such as a one-factor between-subjects analysis of variance.

Exponential(*r*)

Describes the distribution of times between successive independent events in a Poisson process with an average rate of ***r*** events per unit time. The rate ***r*** is the reciprocal of the mean of the Poisson distribution—the average number of events per unit time. Its standard deviation is also 1/***r***.

A model with exponentially distributed times between events is said to be *Markov*, implying that knowledge about when the next event occurs does not depend on the system's history or how much time has elapsed since the previous event. More general distributions such as the gamma or Weibull do not exhibit this property.

Gamma(*A, B*)

Creates a gamma distribution with shape parameter *A* and scale parameter *B*. The scale parameter, *B*, is optional and defaults to *B=1*. The gamma distribution is bounded below by zero (all sample points are positive) and is unbounded from above. It has a theoretical mean of $A \cdot B$ and a theoretical variance of $A \cdot B^2$. When $A > 1$, the distribution is unimodal with the mode at $(A-1) \cdot B$. An exponential distribution results when $A = 1$. As $A \to \infty$, the gamma distribution approaches a normal distribution in shape.

The gamma distribution encodes the time required for *A* events to occur in a Poisson process with mean arrival time of *B*.

Analytica Note: *Some textbooks use Rate=1/B, instead of B, as the scale parameter.*

When to use

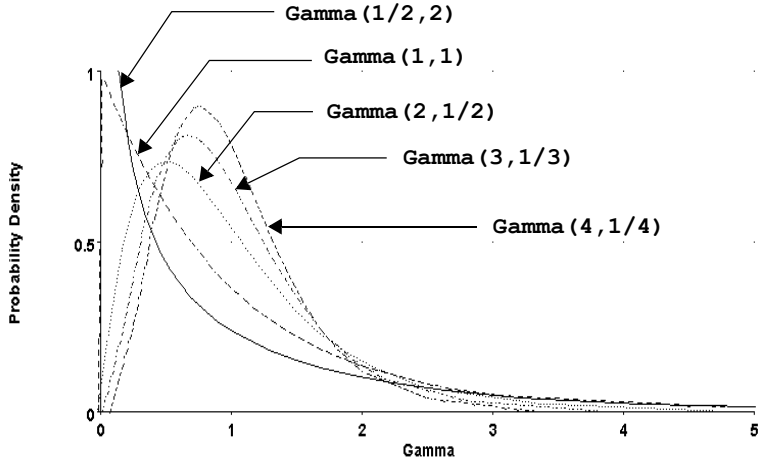
Use the gamma distribution with A>1 if you have a sharp lower bound of zero but no sharp upper bound, a single mode, and a positive skew. The Lognormal distribution is also an option in this case. `Gamma()` is especially appropriate when encoding arrival times for sets of events. A gamma distribution with a large value for *A* is also useful when you wish to use a bell-shaped curve for a positive-only quantity.

Library

Distribution

Examples

Gamma distributions with *mean=1*:



Logistic (*m, s*)

The logistic distribution describes a distribution with a cumulative density given by

$$F(x) = \frac{1}{1 + e^{\frac{-(x-m)}{s}}}$$

The distribution is symmetric and unimodal with tails that are heavier than the normal distribution. It has a mean and mode of ***m***, variance of $s^2\pi^2/3$ and kurtosis of ***6/5*** and no skew. The scale parameter, ***s***, is optional and defaults to 1.

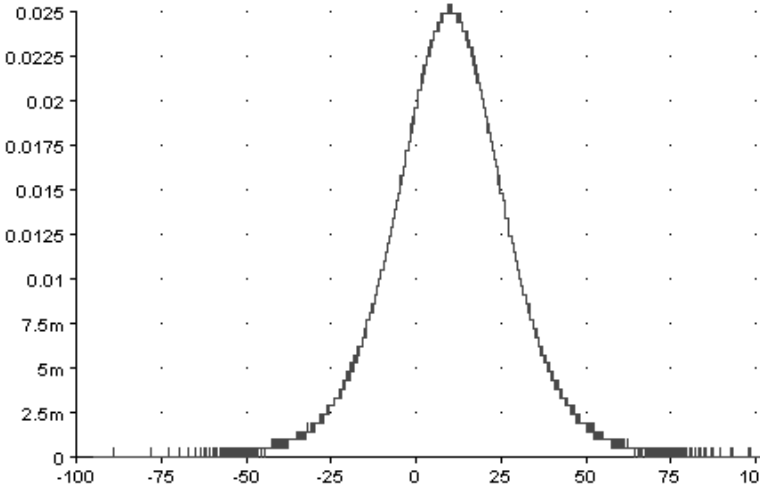
The logistic distribution is particularly convenient for determining dependent probabilities using linear regression techniques, where the probability of a binomial event depends monotonically on a continuous Variable *x*. For example, in a toxicology assay, *x* may be the dosage of a toxin, and *p(x)* the probability of death for an animal exposed to that dosage. Using *p(x) = F(x)*, the logit of *p*, given by

Logit(p(x)) = Ln(p(x) / (1-p(x))) = x/s - m/s

has a simple linear form. This linear form lends itself to linear regression techniques for estimating the distribution—for example, from clinical trial data.

Example

```
Logistic(10, 10)
```



Lognormal (*median, gsdev*)

Creates a lognormal distribution with median of *median* and geometric standard deviation of *gsdev*. The geometric standard deviation must be 1 or greater. The range [*median/gsdev*, *median* × *gsdev*] encloses about 68% of the probability. *Gsdev* is sometimes also known as the ***uncertainty factor*** or ***error factor***.) *Median* and *gsdev* must be positive.

The log of a lognormal quantity has a normal distribution with mean of **Ln**(*median*) and standard deviation of **Ln**(*gsdev*).

When to use

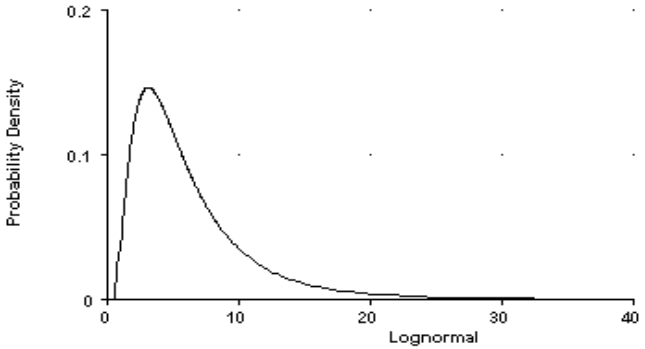
Use the lognormal distribution if you have a sharp lower bound of zero but no sharp upper bound, a single mode, and a positive skew. The gamma distribution is also an option in this case. This distribution is particularly appropriate if you believe that the uncertain quantity is the product (or ratio) of a large number of independent random Variables.

Library

Distribution

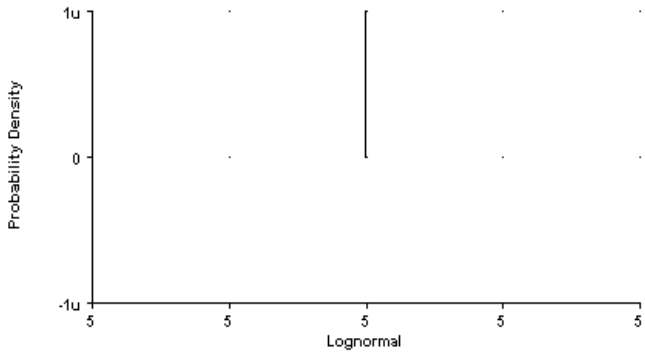
Examples

```
Lognormal(5, 2) →
```



The case of *gsdev*=1 gives a delta function (spike) at *median*.

```
Lognormal(5, 1) →
```



Normal (*mean, stddev*)

Creates a normal or Gaussian probability distribution with *mean* and standard deviation *stddev*. The standard deviation must be 0

or greater. The range [*mean-stddev*, *mean+stddev*] encloses about 68% of the probability.

When to use

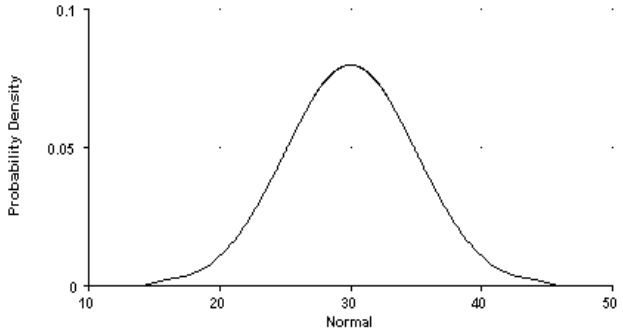
Use a normal distribution if the uncertain quantity is unimodal and symmetric and the upper and lower bounds are unknown, possibly very large or very small (unbounded). This distribution is particularly appropriate if you believe that the uncertain quantity is the sum or average of a large number of independent, random quantities.

Library

Distribution

Example

```
Normal(30, 5) →
```



StudentT(*d*)

The Student T describes the distribution of the deviation of a sample mean from the true mean when the samples are generated by a normally distributed process centered on the true mean. The ***T*** statistic is:

```
T = (m - x)/(s Sqrt(n))
```

where ***x*** is the sample mean, ***m*** is the actual mean, ***s*** is the sample standard deviation, and ***n*** is the sample size. T is distributed according to Students-T with ***d = n-1*** degrees of freedom.

The StudentT distribution is often used to test the statistical hypothesis that a sample mean is significantly different from zero. If $x_1..x_n$ measurements are taken to test the hypothesis ***m***>0,

```
GetFract(StudentT(n-1),0.95)
```

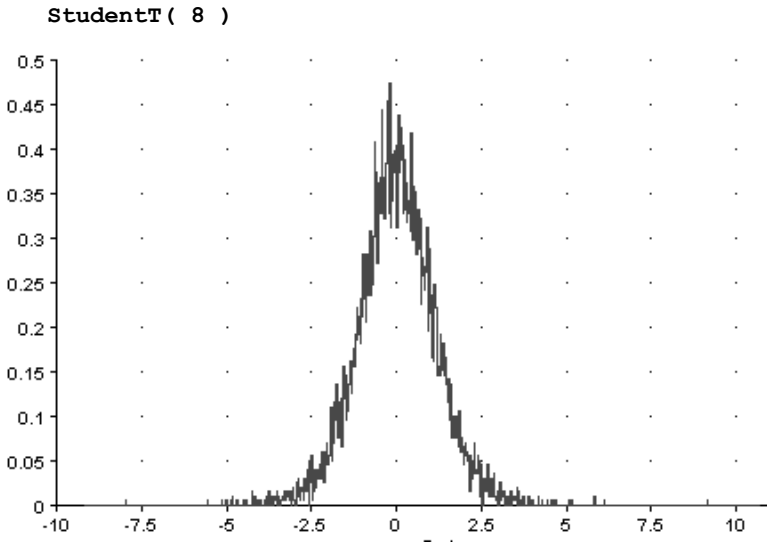
is the acceptance threshold for the ***T*** statistic. If ***T*** is greater than this fractile, we can reject the null hypothesis (that ***m***<=0) at 95% confidence. When using **GetFract** for hypothesis testing, be sure

to use a large sample size, since the precision of this computation improves with sample size.

The Student T can also be useful for modeling the power of hypothetical experiments as a function of the sample size ***n***, without having to model the outcomes of individual trials.

Samples from the Student T distribution are generated using the Monte Carlo sampling method only, regardless of the Uncertainty Settings. Latin Hypercube methods for sample generation are not available.

Example



Triangular (*min, mode, max*)

Creates a triangular distribution, with minimum *min*, mode *mode*, and maximum *max*. *Min* must be not be greater than *mode*, and *mode* must not be greater than *max.*

When to use

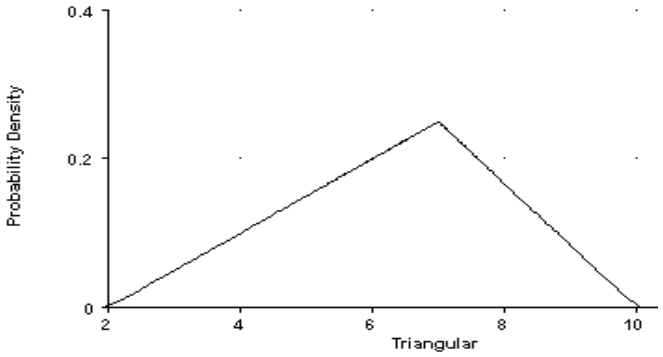
Use the triangular distribution when you have the bounds and the mode, but have little other information about the uncertain quantity.

Library

Distribution

Example

`Triangular(2, 7, 10)` →



Uniform (*min, max*)

Creates a uniform distribution between values *min* and *max*.

When to use

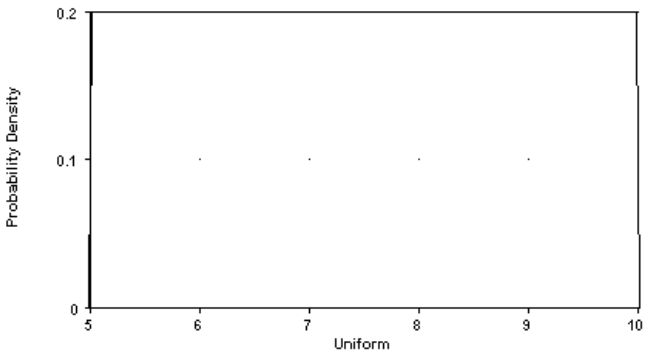
If you know nothing about the uncertain quantity other than its bounds, a uniform distribution between the bounds is appealing. However, situations in which this is truly appropriate are rare. Usually one end, or the middle, of the range is more likely than the rest; that is, the quantity has a mode. In such cases, a beta or triangular distribution is a better choice.

Library

Distribution

Example

`Uniform(5, 10)` →



Weibull(*n, s*)

The Weibull distribution has a cumulative density given by

for **t >= 0**.

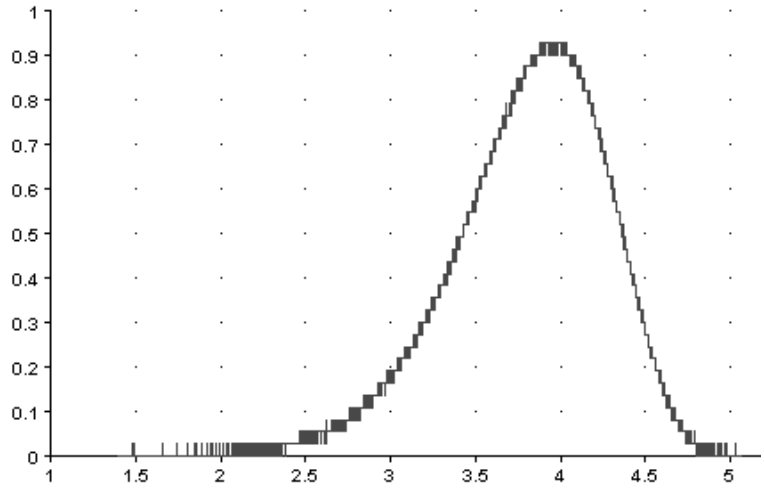
$$f(x) = 1 - e^{-\left(\frac{t}{s}\right)^n}$$

It is similar in shape to the gamma distribution, but tends to be less skewed and tail-heavy.

The Weibull distribution is often used to represent failure time in reliability models. In such models, $f(x)$ may represent the proportion of devices that experience a failure within the first ***x*** time units of operation, the number of insurance policy holders that file a claim within ***x*** days.

Example

```
Weibull(10, 4) →
```



Custom continuous distributions

Cumdist (*P, R, I*)

Specifies a continuous probability distribution by an array of cumulative probabilities, *P*, for an array of corresponding outcome values, *R*, for the quantity. Either *R* must be an index of *P*, or *P* and *R* must have an index in common. If *P* or *R* have more than

one index, you must specify the relevant index for linking *P* and *R* as a third parameter, *I*.

`Cumdist()` uses linear interpolation of the cumulative distribution between the specified points, which implies a piecewise uniform distribution.

The values of *P* must be non-decreasing. *P*'s first value must be 0, and its last value must be 1. The values of *R* must be increasing.

Library Distribution

Example

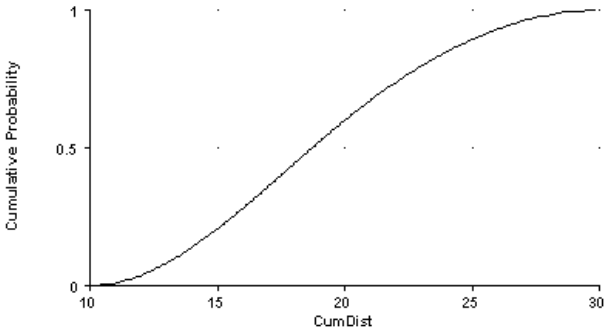
Array_b:
Index_a ▶

	1	2	3
	0	0.6	1.0

Array_x:
Index_a ▶

	1	2	3
	10	20	30

`CumDist(Array_b, Array_x)` →



Fractiles (*L*)

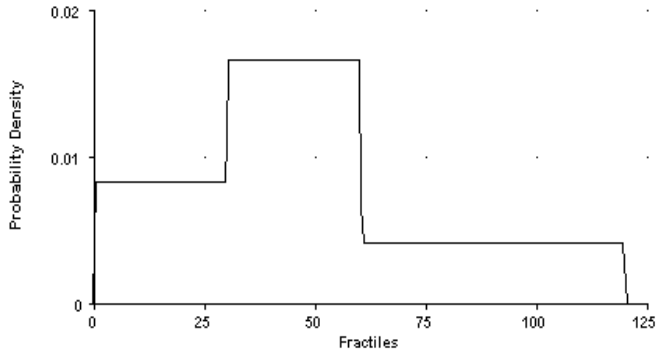
Specifies a continuous probability distribution by an array of evenly spaced fractiles, *L*. *L* must be a one-dimensional array of non-decreasing numbers. If *L* contains *n*+1 numbers, then L_i is the *i/n* fractile—that is, for an uncertain quantity, *x*, $P(x \le L_i) = i/n$. `Fractiles()` uses linear interpolation on the cumulative distribution between the specified fractiles, which implies a piecewise uniform distribution.

If any value in *L* is probabilistic, its mid value is used to obtain the fractile.

Library Distribution

Example The following definition describes a distribution over the range 0 to 120 (0 and 100% fractiles), with its median at 45 (50% fractile), and quartiles at 30 and 60 (25% and 75% fractiles):

```
Fractiles( [0, 30, 45, 60, 120] ) →
```



Probdist (*P, R, I*)

Specifies a continuous probability distribution as an array of probability density values, *P*, for an array of corresponding outcome values, *R*, for the quantity. `Probdist()` performs a linear interpolation between the points on the density function. The values of *P* must be nonnegative. They will be normalized so that the total probability enclosed is 1.0. The values of *R* must be increasing.

The values of *P* should start and end at 0. If the first (or last) value of *P* is not zero, Analytica assumes zero at $2R_1 - R_2$ (or $2Rn - Rn\text{-}1$).

Either *R* must be an index of *P*, or *P* and *R* must have an index in common. If *P* or *R* have more than one index, you must specify the relevant index for linking *P* and *R* as a third parameter, *I*.

Library Distribution

Example *Array_p:*
Index_a ▶

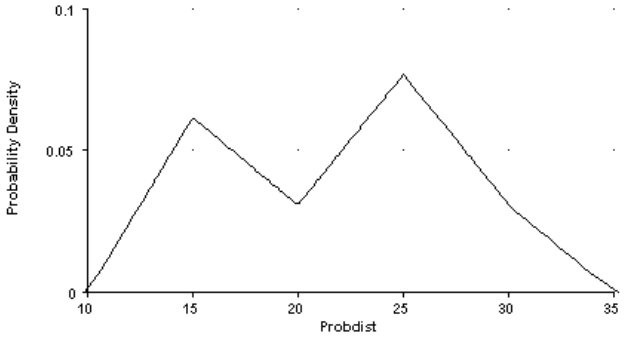
	1	2	3	4	5	6
	0	0.4	0.2	0.5	0.2	0

Array_r:

Index_a ▶

	1	2	3	4	5	6
	10	15	20	25	30	35

Probdist(Array_p, Array_r) →



Truncate (*Dist, X*)

Truncates a probabilistic value *Dist* at and below deterministic value *X*. If *Dist* is not a distribution, *Truncate* returns *Dist*.

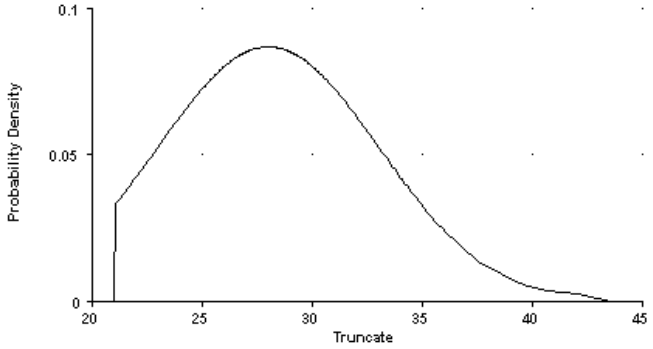
Truncate does not discard sample values; it generates a new complete sample for the quantity with the same probability distribution as *Dist* above *X*, and 0 below *X*.

Since **`Truncate()`** resamples from the truncated distribution, the result will be nearly independent of *Dist*. Hence, importance and other measures that depend on correlations with *Dist* or with probabilistic Variables on which *Dist* depends will be near zero, which may be misleading.

Library Distribution

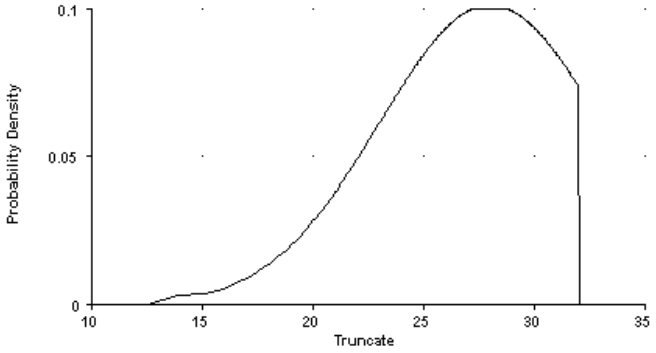
Examples

Mpg: `Normal(28, 5)`
`Truncate(Mpg, 21)` →



To truncate a distribution at or above a specified value, use:

`-Truncate(-Dist, -X)`
`-Truncate(-Mpg, -32)` →



Multivariate distributions

Analytica 3.1 comes with a Multivariate Distributions library containing functions for handling multivariate distribtions. To add this library to your model see “Adding library to a model” on page 405. The Multivariate Distributions library contains the following functions.

Binormal(*Mean_Vector, SDev_Vector, I, CorrelationCoef*)

`Binormal` returns a two-dimensional normal (or bivariate Gaussian) distribution with the indicated means (speciffied by a vector

of means) and separate standard deviations (also specified by a vector, the standard deviations must be positive) and the specified correlation coefficient. The index I, must have exactly two elements, and the standard deviations must be indexed by I.

Correlate_Dists(*Distributions, RankCorrelations, I, J*)

`Correlate_Dists` reorders the samples in Distributions so that they match the desired rank correlations between distributions as closely as possible. Rank correlations must be positive and the valus on the diagonal must all be 1.

Correlate_With(*Sample, ReferenceSample, RankCorrelation*)

`Correlate_Withs` reorders the samples of Samples so that the result is correlated with ReferenceSample with a rank correlation as close to RankCorrelation as possible. *I.e.,* to generate a log-Normal distribution that is highly correlated with Sample1, use `Correlate_With(LogNormal(2,3), Sample1, 0.8)`

Dirichlet(*alpha, N*)

`Dirichlet` returns a Dirichlet distribution with I parameters alpha_i (all greater than 0). Each sample of a Dirichlet distribution produces a random vector whose elements sum to 1. The Dirichlet distribution is the multidimensional generalization of the beta distribution. Dirichlet distributions are commonly used to represent second order probability information.

Gaussian(*MeanVector, CovarianceMatrix, I, J*)

`Gaussian` returns a multivariate Gaussian distribution based on the vector of mean values and the two-dimensional covariance matrix specified. The covariance matrix must be symmetric and positive-definite. The MeanVector must be indexed by I, and the covariance matrix must be indexed by I and J. I and J must have the same length.

Multinomial(*N, theta, I*)

`Multinomial` returns the multinomial distribution. N represents the number of possible outcomes, theta is a vector, indexed by I, representing the probability of each outcome. All the values of

theta should, therefore, sum to 1. If theta does not sum to 1, it is normalized.

This is a generalization of the binomial distribution to N possible outcomes. For example, the distribution of outcomes for rolling a fair die is given by `Multinomial(6, theta,I)`, where theta = (0.16667, 0.16667, 0.16667, 0.16667, 0.16667, 0.16667,) and I = (1, 2, 3, 4, 5, 6).

SampleCovariance(*X, I, J, R*)

`SampleCovariance` returns a covariance matrix based on Sampled data X, indexed by I and R. I is the dimensionality of X, and R corresponds to the samples. The covariance matrix is indexed by I and J. J must be the same length as I.

Analytica Note: *The mean is simply Average(X,R) and so does nto warrant a separate function.*

Advanced Probability Functions

The following functions are not themselves probability distributions, but they are useful for various probabilistic analyses, including building other probability distributions. They are available from the **Advanced math function** option in the **Definition** menu.

BetaFn(*A, B*)

The Beta function, defined as:

$$BetaFn(A,B) = \int_0^1 x^{A-1}(1-x)^{B-1}dx$$

BetaI(*X, A, B*)

The incomplete beta function, defined as:

$$BetaI(X,A,B) = \frac{1}{Beta(A,B)}\int_0^X x^{A-1}(1-x)^{B-1}dx$$

The incomplete beta function is equal to the cumulative probability of the beta distribution at *X*. It is useful in a number of mathematical and statistical applications.

The cumulative binomial distribution, defined as the probability that an event with probability *p* occurs *k* or more times in *n* trials, is given by:

$$Pr = BetaI(p, k, n-k+1)$$

The Student's distribution with *n* degrees of freedom, used to test whether two observed distributions have the same mean, is readily available from the beta distribution as:

$$Student(x|n) = 1 - BetaI(n/(n+x^2), n/2, 1/2)$$

The F-distribution, used to test whether two observed samples with n_1 and n_2 degrees of freedom have the same variance, is readily obtained from **BetaI** as:

$$F(x, n_1, n_2) = BetaI(n_2/(n_1 x + n_2))$$

Combinations(*k, n*)

"*n* choose *k*". The number of unique ways that *k* items can be chosen from a set of *n* elements (without replacement and ignoring the order).

```
Combinations(2,4) → 6
```

They are: {1,2}, {1,3}, {1,4}, {2,3}, {2,4}, {3,4}

Permutations(*k, n*)

The number of possible permutations of *k* items taken from a bucket of *n* items.

```
Permutations(2,4) → 12
```

They are: {1,2}, {1,3}, {1,4}, {2,1}, {2,3}, {2,4}, {3,1}, {3,2}, {3,4}, {4,1}, {4,2}, {4,3}

CumNormal(*X, mean, stddev*)

Returns the cumulative probability

$$p = Pr[x \le X]$$

for a normal distribution with a given mean and standard deviation. ***Mean*** and ***stddev*** are optional and default to ***Mean*** = 0, ***stddev*** = 1.

```
CumNormal(1) - CumNormal( -1 ) → .683
```

i.e., 68.3% of the area under a normal distribution is contained within one standard deviation of the mean.

CumNormalInv(*P, mean, stddev*)

The inverse cumulative probability function for a normal distribution. Returns the value X where

$$\boldsymbol{P} = Pr[x \le X]$$

Mean and ***stddev*** are optional and default to ***Mean*** = 0, ***stddev*** = 1.

Erf(*x*)

The error function, defined as:

$$Erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

ErfInv(*Y*)

The inverse error function. Returns the value X such that Erf(X)=Y.

```
ErfInv(Erf(2)) → 2
```

GammaFn(*x*)

Returns the gamma function of *X*, defined as

$$\Gamma(X) = \int_0^\infty t^{x-1} e^{-t} dt$$

The gamma function grows very quickly. For example, when *n* is an integer, *GammaFn(n+1) = n!*. For this reason, it is often preferable to use the *LGamma* function.

GammaI(*X, A, B*)

Returns the incomplete gamma function, defined as:

$$GammaI(X, A, B) = \frac{1}{\Gamma(A)} \int_0^{x/B} e^{-t} t^{A-1} dt$$

A is the shape parameter, *B* is an optional scale factor (default *B*=1). Some textbooks use $\lambda = 1/A$ as the scale factor. The incomplete gamma function is defined for $X \geq 0$.

The incomplete gamma function returns the cumulative area from zero to *X* under the gamma distribution.

The incomplete gamma function is useful in a number of mathematical and statistical contexts.

The cumulative Poisson distribution function, which encodes the probability that the number of Poisson random events (*x*) occurring will be less than *k* (where *k* is an integer) where the expected mean number is *A*, is given by (recall that parameter *B* is optional):

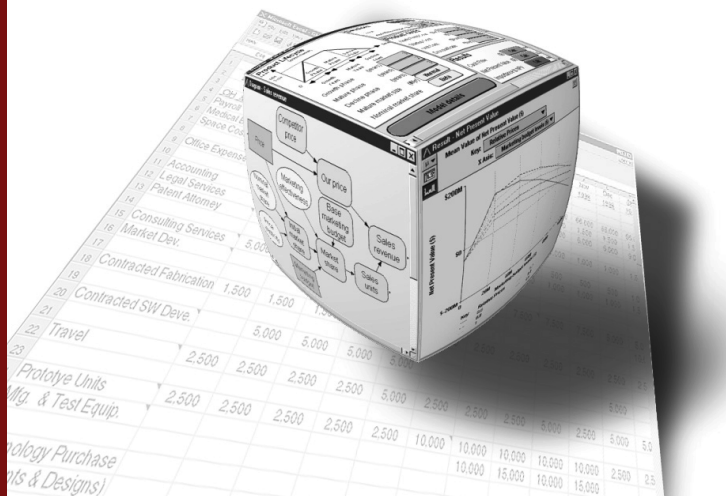
$$P(x < k) = GammaI(k, A)$$

GammaIInv(*Y, A, B*)

The inverse of the incomplete gamma function. Returns the value X such that GammaI(X,A,B)=Y. *B* is optional and defaults to 1.

Chapter 15

Sensitivity and Uncertainty Analysis



In this Chapter

This chapter shows you how to:

- Analyze the uncertainty of Variables
- Analyze relationships between uncertain Variables
- Analyze the sensitivity of outputs to changes in inputs

15: Uncertainty and sensitivity

This chapter describes Analytica's tools for analyzing the uncertainty of Variables, relationships between uncertain Variables, and sensitivity of outputs to changes in inputs. It covers the statistical functions, sensitivity analysis functions, scatter graphs, and importance analysis.

Statistical functions

This section describes Analytica's built-in statistical functions, for use in Variable definitions. Many of these functions are used in the Result window Uncertainty View options (see “Uncertainty view options” on page 52). These functions can assist with analysis of probabilistic Variables.

Analytica Note: *All statistical functions produce estimates from the underlying random sample for each probabilistic quantity. These estimates are not exact, but will vary from one evaluation to the next due to the variability inherent in random sampling. Hence, your results may not exactly match the results shown in the examples here. For greater precision, use a larger sample size (see “Selecting the sample size” on page 499 on how to select a sample size).*

The calculation formulas use the following notation:

x_i	the *i*th sample value of probabilistic Variable *X*
$\bar{x}$	the mean of probabilistic Variable *X* (see **`Mean()`**)
s	standard deviation (see **`Sdeviation()`**)
m	sample size (see Appendix A, “”).

Analytica Note: *These statistical functions will not calculate statistics for an array of data unless it is a sample indexed by Run. To obtain statistics on an array of data with another index, see the Data Statistics library in the Libraries folder.*

The examples in this section use the following Variables:

Alt_ fuel_ price: `Normal(1.25, 0.1)`
Fuel_price: `Normal(1.19, 0.1)`
Skfuel_price: `Beta(4,2,1,1.5)`

Correlation (*X*, *Y*)

Returns an estimate of the correlation between the probabilistic expressions *X* and *Y*, where -1 means perfectly negatively correlated, 0 means no correlation, and 1 means perfectly positively correlated.

`Correlation(X,Y)`, a measure of probabilistic dependency between uncertain Variables, is sometimes known as the Pearson product moment coefficient of correlation, *r*. It measures the strength of the linear relationship between *X* and *Y*, using the formula:

$$\frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \times \sum_i (y_i - \bar{y})^2}}$$

Library Statistical

Example With *Samplesize* set to 100 and number format set to two decimal digits:

```
Correlation(Alt_fuel_price + Fuel_price,
Fuel_price) → 0.71
```

Correlation of two independent, uncorrelated distributions approaches 0 as the sample size approaches infinity.

Example With *Samplesize* = 20:

```
Correlation(Normal(1.19,0.1), Normal(1.19,0.1))
→ -.28
```

With *Samplesize* = 1000:

```
Correlation(Normal(1.19,0.1),Normal(1.19,0.1))
→ 0.03
```

Frequency (*X, I*)

If *X* is a discrete uncertain Variable, returns an array indexed by *I*, giving the frequency, or number of occurrences of discrete values *I*. *I* must contain unique values; if numeric, the values must be increasing.

If *X* is a continuous uncertain Variable and *I* is an index of numbers in increasing order, it returns an array indexed by *I*, with the count of values in the sample *X* that are equal to or less than each value of *I* and greater than the previous value of *I*.

If *X* is non-probabilistic, **`Frequency()`** returns *Samplesize* for each value of *I* equal to *X*.

Since **`Frequency()`** is computed by counting occurrences in the probabilistic sample, it is a function of *Samplesize* (see “Uncertainty Setup dialog box” on page 291). If you want the relative frequency rather than the count of each value, divide the result by *Samplesize*.

Library Statistical

Example (Continuous) *Index_a*: `[1.2,1.25]`

`Frequency(Fuel_price, Index_a)` → *Index_a* ▶

	1.2	1.25
	54	19

Example (Discrete) *Bern_out*: `[0,1]`

(Possible outcomes of the Bernoulli Distribution)

With *Samplesize* = 100:
`Frequency(Bernoulli (0.3), Bern_out)` → *Bern_out* ▶

	0	1
	70	30

With *Samplesize* = 25:
`Frequency(Bernoulli (0.3), Bern_out)` → *Bern_out* ▶

	0	1
	18	7

(Compare the Bernoulli example on page 303.)

Getfract (*X, P*)

Returns an estimate of the *P*th fractile (also known as quantile or percentile) of *X*. This is the value of *X* such that *X* has a probability *P* of being less than that value. If *X* is non-probabilistic, all fractiles are equal to *X*.

The value of *P* must be a number or array of numbers between 0 and 1, inclusive.

Library Statistical

Examples `Getfract(X,0.5)` returns an estimate of the median of *X*.

```
Getfract(Fuel_price, 0.5) → 1.19
```

The following returns a table containing estimates of the 10%ile and 90%ile values, that is, an 80% confidence interval.

Fract: `[0.1,0.9]`

`Getfract(Fuel_price, Fract)` →
Fract ▶

	0.10	0.90
	1.06	1.32

Kurtosis (*X*)

Returns an estimate of the kurtosis of *X*. *X* must be probabilistic.

Kurtosis is a measure of the peakedness of a distribution. A distribution with long thin tails has a positive kurtosis. A distribution with short tails and high shoulders, such as the uniform distribution, has a negative kurtosis. A normal distribution has zero kurtosis.

`Kurtosis(X)` uses the formula:

$$\left(\frac{1}{m} \sum_{i=1}^{m} \left[\frac{x_i - \bar{x}}{\sigma} \right]^4 \right) - 3$$

Library Statistical

Example `Kurtosis(Skfuel_prices)` → `-0.48`

Mean (*x*)

Returns an estimate of the mean of *X* if *X* is probabilistic. Otherwise, returns *X*.

Mean(X) uses the formula:

$$\frac{1}{m}\sum_{i=1}^{m} x_i = \bar{x}$$

Library Statistical

Examples

`Mean(Fuel_price)` → `1.19`
`Mean(Skfuel_price)` → `1.33`

Mid (*x*)

Returns the mid value of *X*. `Mid(X)` forces deterministic evaluation in contexts where *X* would otherwise be evaluated probabilistically.

The mid value is calculated by substituting the *median* for most full probability distributions in the definition of a Variable or expression, and using the mid value of any inputs. The mid value of a Variable or expression is *not* necessarily equal to its true median, but is usually close to it.

Library Statistical

Example `Mid(Fuel_price)` → `1.19`

Probability(*B*)

Returns an estimate of the probability or array of probabilities that the Boolean value *B* is true.

Library Statistical

Example `Probability(Fuel_price < 1.19)` → `0.5`

Probbands (*X*)

Returns an estimate of probability or "confidence" bands for *X* if *X* is probabilistic. Otherwise returns *X* for every band. The probabilities are specified in the Uncertainty Setup dialog box, Probability Bands option (see “Uncertainty Setup dialog box” on page 291).

Library Statistical

Example `Probbands(Fuel_price)` →
Probability ▶

	0.05	0.25	0.5	0.75	0.95
	1.025	1.123	1.19	1.257	1.355

Rankcorrel (*X, Y*)

Returns an estimate of the rank-order correlation coefficient between the distributions *X* and *Y*. *X* and *Y* must be probabilistic.

`Rankcorrel(X,Y)`, a measure of the dependence between *X* and *Y*, is sometimes known as Spearman’s rank correlation coefficient, r_s.

Rank-order correlation is measured by computing the ranks of the probability samples, and then computing their correlation. By using the rank order of the samples, the measure of correlation is not affected by skewed distributions or extreme values, and is, therefore, more robust than simple correlation. Rank-order correlation is used for importance analysis (see “Importance analysis” on page 343).

Library Statistical

Example With *Samplesize* = 100:
`Rankcorrel(Fuel_price, Alt_fuel_price)` → `.02`

Sample (*x*)

Evaluates *X* probabilistically and returns a sample of values from the distribution of *X* in an array indexed by the system Variable *Run*. If *X* is not probabilistic, returns *X*. The system Variable *Samplesize* specifies the size of this sample. You can set *Samplesize* in the Uncertainty dialog box (see “Uncertainty Setup dialog box” on page 291).

Library Statistical

When to use Use when you want to force probabilistic evaluation, or look at raw sample values.

Example Here are the first six values of a sample:

```
Sample(Fuel_price) →
Iteration(Run) ▶
```

	1	2	3	4	5	6
	1.191	1.32	1.19	1.164	1.191	0.962

Sdeviation (*x*)

Returns an estimate of the standard deviation of *X* from its sample if *X* is probabilistic. If *X* is non-probabilistic, returns 0.

`Sdeviation(X)` uses the formula:

$$\sqrt{\frac{1}{m-1}\sum_{i=1}^{m}(x_i-\bar{x})^2} = \sigma$$

Library Statistical

Example

```
Sdeviation(Fuel_price) → 0.10
```

Skewness (*x*)

Returns an estimate of the skewness of *X*. *X* must be probabilistic.

Skewness is a measure of the asymmetry of the distribution. A positively skewed distribution has a thicker upper tail than lower tail, while a negatively skewed distribution has a thicker lower tail than upper tail. A normal distribution has a skewness of zero.

`Skewness(X)` uses the formula:

$$\frac{1}{m}\sum_{i=1}^{m}\left[\frac{x_i-\bar{x}}{\sigma}\right]^3$$

Library Statistical

Example `Skewness(Skfuel_price)` → `-0.45`

Statistics (*x*)

Returns an array of statistics of *X*. Select the statistics in the Uncertainty Setup dialog box, Statistics option (see “Uncertainty Setup dialog box” on page 291).

Library Statistical

Example `Statistics(Fuel_price)` →
Statistics ▶

	Min	Median	Mean	Max	Std. Dev.
	0.93	1.19	1.19	1.45	0.10

Variance (*x*)

Returns an estimate of the variance of *X* if *X* is probabilistic. If *X* is non-probabilistic, returns 0.

`Variance(X)` uses the formula:

$$\frac{1}{m-1}\sum_{i=1}^{m}(x_i-\bar{x})^2 = \sigma^2$$

Library Statistical

Example `Variance(Fuel_price) → 0.01`

Importance analysis

In most complex models, many of the input Variables are uncertain. It is often useful to understand how much each uncertain input contributes to the uncertainty in the output. Typically, a few uncertain inputs are responsible for the lion's share of the uncertainty in the output, while the rest have little impact.

The importance analysis features in Analytica can help you quickly learn which inputs contribute the most uncertainty to the output. You can then concentrate on getting better estimates or building a more detailed model for the one or two most important inputs without spending considerable time investigating issues that turn out not to matter very much.

Importance analysis defined

Importance is the absolute rank-order correlation between the sample of output values and the sample for each uncertain input. It is a robust measure of the uncertain contribution because it is insensitive to extreme values and skewed distributions. Unlike commonly used deterministic measures of sensitivity, it averages over the entire joint probability distribution. Therefore, it works well even for models where the sensitivity to one input depends strongly on the value of another.

Creating an importance Variable

To create an importance analysis Variable:

1. Be sure you are in Edit mode. Select an output Variable's node (usually your model's objective node, but it can be any Variable that has uncertain inputs).
2. Select **Make Importance** from the **Object** menu.

Analytica creates two new Variables, an index Variable and a general Variable. If *Output Variable* is the title of the node you selected, the index Variable is titled *Output Variable Inputs,* and the general Variable is titled *Output Variable Importance*.

Note: *The importance analysis variables do not automatically update when new chance variables are added.*

Example

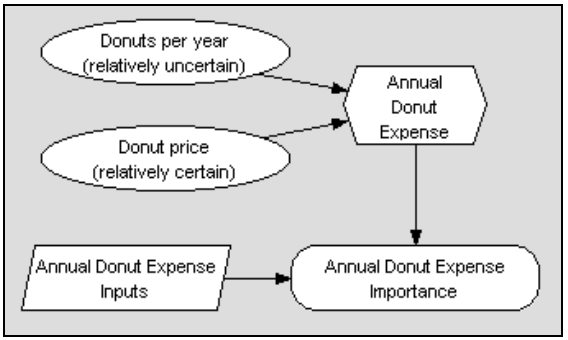
Donuts_per_year: `Normal(150,50)`

Donut_price: `Normal(0.4,0.04)`

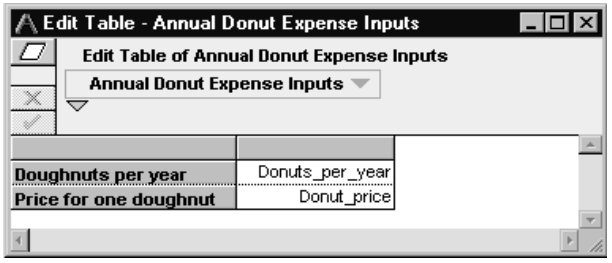
Annual_donut_expense: `Donuts_per_year * Donut_price`

Since the two inputs are multiplied, we would expect the input with the greater relative uncertainty, *Donuts_per_year*, to contribute more uncertainty to *Annual_donut_expense*.

After you select *Annual_donut_expense* and then **Make Importance** from the **Object** menu, the diagram contains two new Variables.



Annual_donut_expense Inputs is a one-dimensional Edit Table of the chance Variables. Its index contains the titles of the chance nodes, and its values are the identifiers of those nodes.

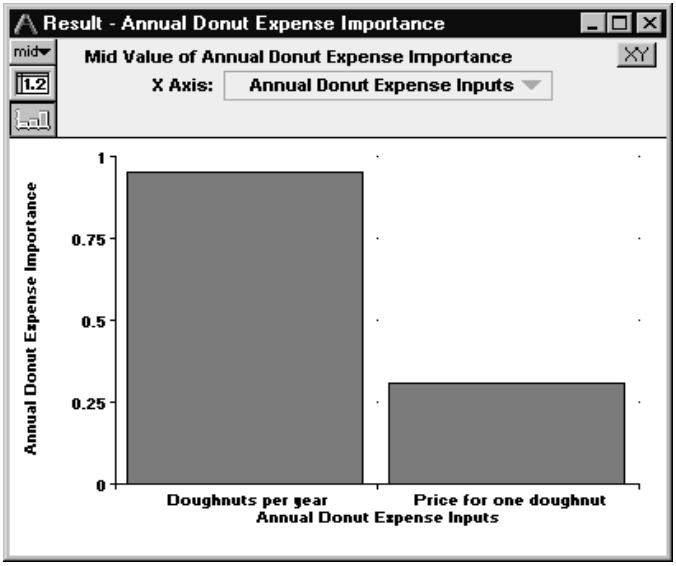


Annual donut expense Inputs evaluates to a set of probability distributions, one for each chance Variable.

Annual donut expense Importance is defined as

```
Abs(Rankcorrel(Annual_donut_expense_inputs,
Annual_donut_expense))
```

The `Rankcorrel()` function computes the rank-order correlation of each input to the output, and then the `Abs()` function computes the absolute value, yielding a positive relative importance.



As expected, *Donuts_per_year* contributes considerably more uncertainty to *Annual_donut_expense* than *Donut_price*.

Analytica Note: *Importance, like every other statistical measure, is estimated from the random sample. The estimates may vary slightly from one sample to another due to random noise. For a sample size of 100, an importance of 0.1 may not be significantly different from zero. But an importance of 0.5 is significantly different from zero. The main goal is to discover those uncertain inputs, typically only two to five, that are the primary contributors to the uncertainty in the output. For greater precision, use a larger sample size.*

Editing importance Variables

If you create an importance analysis Variable for a model, and subsequently refine the model by redefining or adding uncertain inputs, you may want to change the set of input Variables used for the importance analysis. Or, you may want to remove Variables that you already know don't contribute significantly to the uncertainty in the result. Do one of the following:

- Select *Output Variable* and then **Make Importance** from the **Object** menu. The *Output Variable Inputs* node will be updated.
- Open *Output Variable Inputs*' Edit Table and edit the list of input Variables.

Sensitivity analysis functions

Sensitivity analysis enables you to examine the effect of a change in the value of an input Variable on the values of its output Variables.

Examples

The examples in this section refer to the following Variables:

`Gasprice: Normal(1.3, .3)`

(cost of gasoline per gallon within market fluctuations)

Mpy: `12K`

(the average number of miles driven per year)

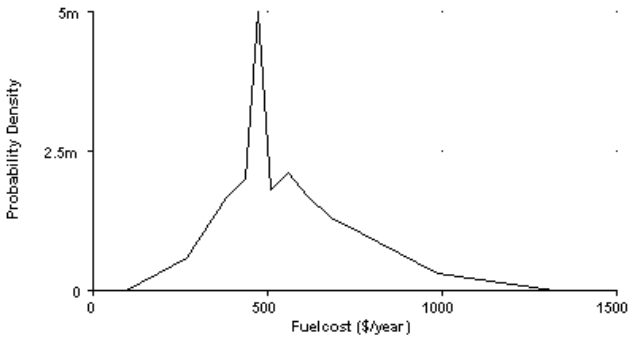
Mpg: `Normal(28, 5)`

(fuel consumption averaged over driving conditions)

Fuelcost: `Gasprice * Mpy / Mpg`

(annual cost of fuel)

Probability density of *Fuelcost:*



Dydx (*Y, X*)

Returns the derivative of expression *Y* with respect to Variable *X*, evaluated at mid values. This function returns the ratio of the change in *Y* to a small change in *X* that affects *Y*. The "small change" is $X/10000$, or 1.0E-6 if *X* = 0.

Library Special

Examples Because *Fuelcost* depends on *Mpg*, a small change in *Mpg* seems to have a modest negative effect on *Fuelcost:*

```
Dydx(Fuelcost, Mpg) → -19.7
```

The reverse is not true, because *Mpg* is not dependent on *Fuelcost*. That is, *Fuelcost* does not cause any change in *Mpg:*

```
Dydx(Mpg, Fuelcost) → 0
```

In this model of *Fuelcost*, a small change in *Gasprice* has by far the largest effect of all its inputs:

```
Dydx(Fuelcost, Gasprice) → 428.6
```

```
Dydx(Fuelcost, Mpy) → 0.04643
```

Analytica Note: *When you evaluate* **DyDx** *in determ (Mid) mode, the Mid value for X is varied and the Mid value of Y is evaluated. In Prob-mode, the sample of X is varied and the sample for Y is computed in prob-mode. Therefore, when Y is a statistical function of X, care must be taken to ensure that the evaluation modes for X and Y correspond. So, for example,*

```
Y := DyDx(Kurtosis( Normal(0,X ) ), X)
```

would not produce the expected result. In this case, when evaluating Y in determ mode, Kurtosis evaluates its parameter, and thus X, in prob mode, resulting in a mis-match in computation modes. To get the desired result, you should explicitly use the Mid value of X:

```
Y := DyDx( Kurtosis(Normal(0,Mid(X))), X )
```

Elasticity (*Y, X*)

Returns the percent change in Variable *Y* caused by a 1 percent change in a dependent Variable *X*.

`Elasticity()` is related to `Dydx()` in the following manner:

```
Elasticity(Y,X) = Dydx(Y,X)*(X/Y)
```

Library Special

Examples

```
Elasticity(Fuelcost, Mpg) → -0.9901
```

```
Elasticity(Fuelcost, Gasprice) → 1
```

A 1% change in Variables *Mpg* and *Gasprice* cause about the same degree of change in *Fuelcost*, although in opposite directions.

Mpg is inversely proportional to the value of *Fuelcost*, while *Gasprice* is proportional to it.

Analytica Note: *When you evaluate* **Elasticity** *in determ (Mid) mode, the Mid value for X is varied and the Mid value of Y is evaluated. In Prob-mode, the sample of X is varied and the sample for Y is computed in prob-mode. Therefore, when Y is a statistical function of X, care must be taken to ensure that the evaluation modes for X and Y correspond.*

Regression(*Y, B, I, K*)

Generalized linear regression. Finds the best-fit (least squared error) curve to a set of data points. Regression finds the parameters a_k in an equation of the form:

$$y = \sum_k a_k B_k(\hat{x})$$

The data points are contained in **Y** (the dependent Variable) and **B** (the independent Variables), both of which must be indexed by **I**. **B** is the basis set and is indexed by **I** and **K**. The function returns the set of parameters a_k indexed by **K**.

With the generalized form of linear regression, it is possible to have several independent Variables, and your basis set may even contain non-linear transformations of your independent Variables. **Regression()** may be used to find the best-fit planes or hyperplanes, best-fit polynomials, and more complicated functions.

Regression uses a state-of-the-art algorithm based on singular-value decomposition that is numerically stable, even if the basis set contains redundant terms.

Example 1

Suppose a set of (x,y) points are contained in **X** and **Y**, both indexed by **I**, and we wish to find the parameters m and b of the best-fit line $y = mx + b$. We first define an index **K** as a list of labels:

```
K: ['m','b']
```

Next, define **B** as a table indexed by **K**:

B: K ▶

	m	*b*
	X	*1*

Regression(Y,B,I,K) returns the coefficients m and b as an array indexed by **K**.

Example 2

We wish to fit the following polynomial to (x,y) data:

$$y = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Define **K** to be the list:

```
B: [X^5,X^4,X^3,X^2,X,1]
```

Regression(Y,B,I,B) returns the best-fit coefficients of the polynomial indexed by **B**.

Whatif (*Ident, Tempval, X*)

Returns the value of expression *Ident* when Variable *Tempval* is set to the value of expression *X*. *Tempval* must be a Variable. The original definition of *Tempval* is restored after evaluation of the **Whatif()** expression, allowing you to explore the effect of a change in *Tempval* without permanently changing it.

Library Special

Example

```
Fuelcost → 557.1

Whatif(Fuelcost, Mpy, 14K) → 650
```

WhatIfAll(*Ident, varlist, X*)

Returns the mid value of *Ident* when the each of Variables in *varList* is assigned the value in *X* one at a time, with the remain-

ing Variables remaining at their nominal values. The result is indexed by *varList*. By having *X* indexed by *varList*, a different value can be assigned to each Variable.

WhatIfAll is useful for performing *ceteris paribus* style sensitivity analysis, in which only on Variable is varied at a time. Tornado diagrams are one such example. Tornado-style analyses are useful since they do not require inputs to be uncertain.

Library

Special

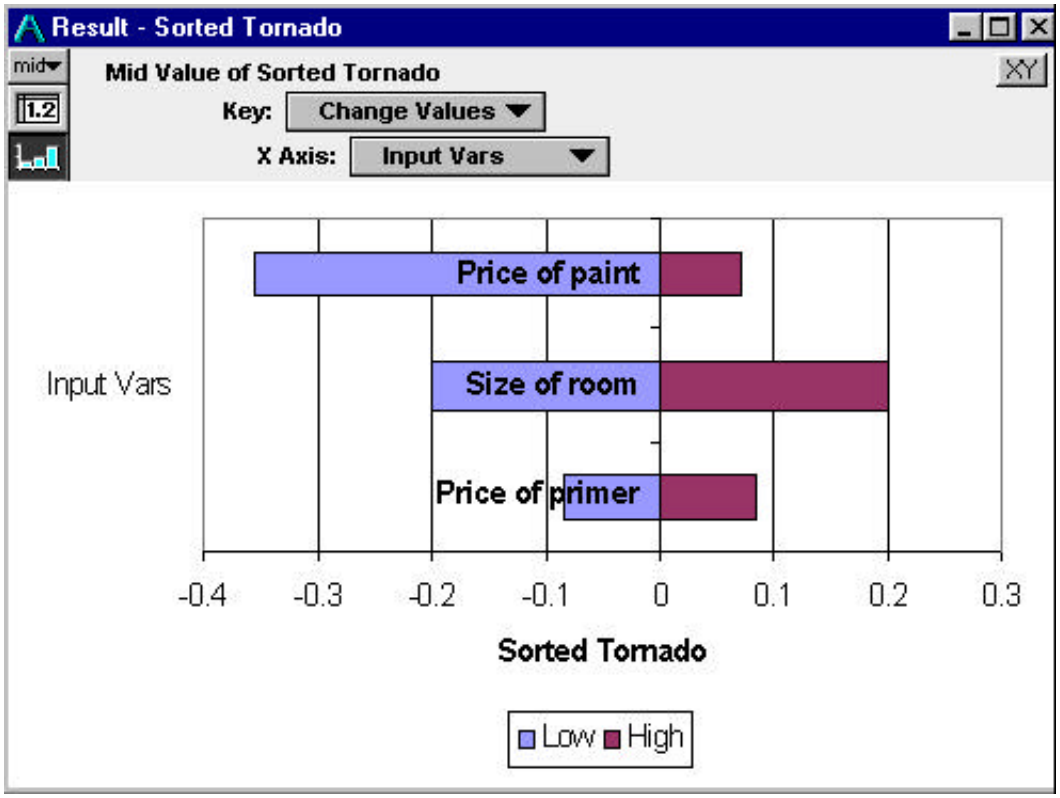
Example 1

Suppose *Z* is a function of *A*, *B*, and *C*, and we wish to examine the effect on *Z* when each input is varied, one at a time, by *10%* from its nominal value. Define:

```
L := [90%,110%]
V := [A,B,C]
MyTornado := WhatIfAll( Z, V, L*V )
```

Example 2: Tornado style diagrams

A Tornado diagram is a common tool used to depict the sensitivity of a result to changes in selected variables. The fundamental analysis behind a tornado diagram consists of varying only one input variable at a time, keeping all other variables at their nominal values. Typically, a "low" and a "high" value are selected for each input, and the output variable is computed while only one variable varies at a time. The result is then displayed as a special type of bar graph, with bars for each input variable displaying the variation from the nominal value. It is standard practice to plot the bars horizontally, sorted so that the widest bar is placed at the top. When drawn in this fashion, the diagram takes on the appearance of a tornado, hence its name. The figure below shows a typical tornado diagram.



Analytica contains very flexible facilities for performing Tornado-style analysis, with immense flexibility to implement custom variations on the basic analysis. Creating a tornado-style diagram in Analytica consists of two fundamental steps:

1. Performing the tornado analysis (obtaining the results when varying each variable separately).
2. Graphing the result (which involves setting up the graph settings appropriately).

Analytica's built-in graphing tool can produce only vertical bar graphs, allowing only rotated tornado diagrams to be created. The convential (non-rotated) tornado diagram can be obtained using Excel Graph from within Analytica.

Performing a Tornado Analysis

To perform a tornado analysis, you must:

1. Identify which output variable to perform the analysis on.
2. Select the input variables are to be varied.
3. Decide what the low and high values are to be for each input variable.

Note: *The input variables do not need to be chance variables. In fact, tornado analysis is often applied to models with no chance variables.*

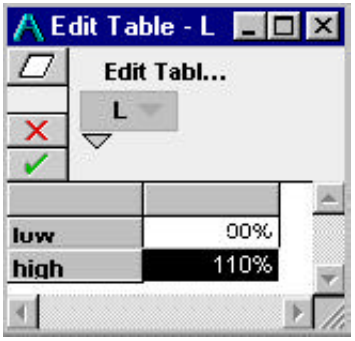
There are several options for selecting low and high values, including:

- Selecting the same absolute low and high levels for every input. This usually only makes sense if inputs are very homogeneous with identical nominal values.
- Selecting absolute low and high values separately for each input variable.
- Varying all inputs by the same relative amount, e.g., low=90% of nominal, high=110% of nominal.
- Varying all inputs between two given fractiles. This only makes sense if your inputs are uncertain variables. Example: Low=10% fractile, High=90% fractile, nominal=50% fractile.

Implementing a tornado analysis

For this example, assume we vary all inputs by the same amount

1. Create an index variable containing a list of input variable identifiers. Suppose this is called **`Vars`**.
2. Create a variable, **`L`**, and define it as a self-indexed table. (To do this, select Table from the Expression pulldown, and select self as an index.) From the edit table, set the self-index labels to read low and high. Set the value corresponding to low to 90%, and set the value corresponding to high to 110%.



3. Create a node, *Tornado_Analysis*. Assume that the output variable is X. Define Tornado as:

```
WhatIfAll( X, Vars, L * Vars )
```

4. Create a node, Sorted_Tornado_Inputs, defined as:

```
sortIndex( abs(Tornado_Analysis[L='high'] -
Tornado_Analysis[L='low']) )
```

5. Create a node, *Sorted_Tornado*, defined as:

```
Tornado_Analysis[Vars=Sorted_Tornado_Inputs]
```

Steps 4 & 5 are not necessary if you do not require your bars to be displayed from largest to smallest. If you do include steps 4 & 5, *Sorted_Tornado* will contain the results of the Tornado analysis, otherwise the result is *Tornado_Analysis*.

It is possible in Analytica to use array abstraction to produce a set of Tornado diagrams, with each Tornado itself indexed by an additional dimension. Additional dimensions are already included if your output variable is itself an array result, in which case you will have a tornado diagram for each element in the output value's array value. This flexibility is unique to Analytica; however, you should note that having multiple tornados in a single result complicates the problem of sorting the bars, since the sort order will, in general, be different for the different bars. If you have extra indexes in your tornado analysis, you will need to either skip steps 4 & 5 above, and display non-sorted Tornados, or select a single sort order based on whatever criteria fits your needs, realizing that not all tornados will display in sorted order. To display a tornado using Excel Graph, your output variable must be a scalar.

The `WhatIfAll` function typically provides the easiest method for implementing a tornado analysis in Analytica. Note that the third

parameter to `WhatIfAll` controls the method by which inputs are varied for the analysis. For example:

- For the case where you select the same absolute low and high levels for every input, `L` would be set to the absolute low and high values, and the third parameter to `WhatIfAll` would be simply `L`.
- For the case where you select absolute low and high values separately for each input variable, you would index `L` by `Vars`, fill in `L`'s table appropriately, then set the third parameter to be just `L`.
- And for the case where you vary all inputs between two given fractiles, you would set `L` to the desired fractiles, and use as the third parameter the expression: `getFract(X,L)`.

Graphing a Tornado using Analytica's built-in graphing tool

Once you have performed the Tornado analysis, you can easily graph the results as a rotated tornado diagram. To obtain the desired appearance, a few graph settings must be selected. The steps are:

1. Select **Show Result** for the *Tornado_Analysis* or *Sorted_Tornado* variable. Press the Graph button if necessary.
2. Pivot the index order (if necessary) so that Vars is on the X-axis and L is the Key.
3. Select **Graph Setup...** and then **Graph Style**.
4. Set the Line Style to the filled bar setting. Set Overlap=100%, Origin=X. (Where X is your output variable of interest). Press Apply.

Graphing a Tornado using the Excel Graph tool

Since Excel Graph can produce horizontal bar graphs (a graph type not available from Analytica's built-in graphing tool), you can use Excel Graph to produce "non-rotated" tornados. The steps are:

1. Select **Show Result** for the *Tornado_Analysis* or *Sorted_Tornado* variable. Press the Graph button if necessary.
2. Pivot the index order (if necessary) so that Vars is on the X-axis and `L` is the Key.

3. Select **Graph Setup...** and then **Excel Graph**. Click on **Excel Chart** and press OK.
4. Double click on the graph. Excel is brought to the foreground. From the Chart menu, select **Chart Type**. Click on the horizontal bar graph chart type and select the first sub-type (appears in the icon as side-by-side bars).
5. While still in Excel, depress the right mouse button while the cursor is over a bar. Select **Format Data Series....** In the dialog, select **Options**. Set Overlap=100%. Press OK.
6. In Analytica, note the result value of X (it must be a scalar).
7. In Excel, click on and select the X-axis, then select **Format X-Axis...** from the right mouse button menu. Select the **Scale** tab, and enter the value of X into the box labeled "Category (X) axis Crosses at". Press OK.
8. The graph is now set up. Return to Analytica.

X-Y results

When evaluating a Variable, you can specify another Variable to view it against, for Mid, Mean, Statistics, Probability Bands, and Sample.

To graph one Variable against another:

1. Open a Result window for the *y*- (vertical axis) Variable.
2. Click on the **XY** button located in the top right corner of the window to open the Object Finder dialog box.

3. In the Object Finder, select the *x*- (horizontal axis) Variable

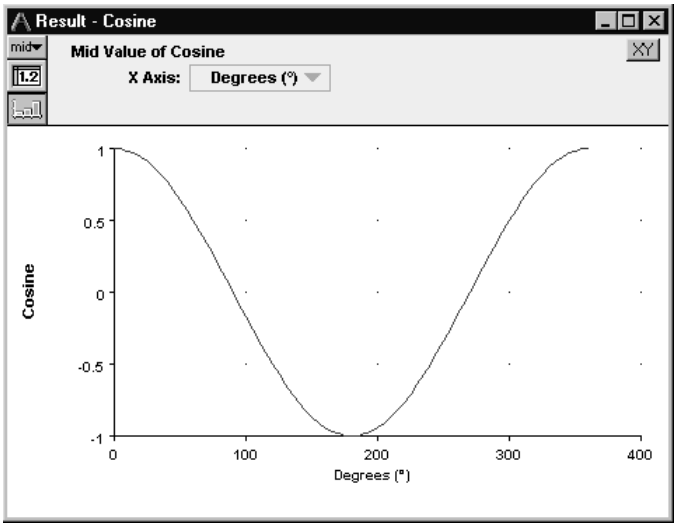
The two Variables in an XY window must share at least one index, and all indexes of *X* must also be indexes of *Y*. The popup menu in the index selection area becomes **Common Index**—only indexes of both *X* and *Y* may be selected.

Example

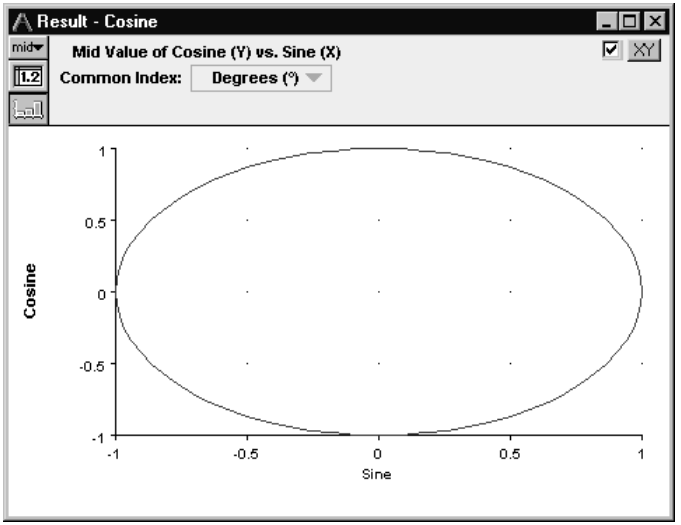
Degrees: `Sequence(0,360,10)`

Sine: `Sin(Degrees)`

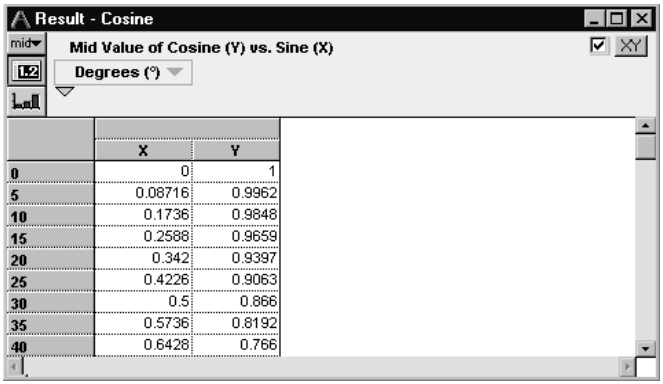
Cosine: `Cos(Degrees)` →



Click on the **XY** button. In the Object Finder dialog under Current Module select the Variable *Sine* to display:



Click on the Table View button to display:



	X	Y
0	0	1
5	0.08716	0.9962
10	0.1736	0.9848
15	0.2588	0.9659
20	0.342	0.9397
25	0.4226	0.9063
30	0.5	0.866
35	0.5736	0.8192
40	0.6428	0.766

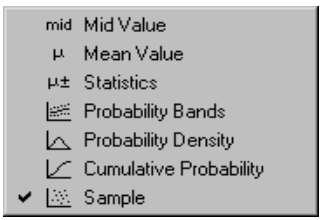
To return to the graph or table of *Cosine* vs. *Degrees*, click in the **XY** check box.

Scatter plots

A scatter plot, graphing the samples of two probabilistic Variables against each other, can provide insight into their probabilistic relationship.

To generate a scatter plot for two Variables, *X* and *Y*:

1. Open a Result window for *Y*.
2. Click on the **XY** button located in the top right corner of the window to open the Object Finder dialog box.
3. In the Object Finder, select the *X* Variable.
4. In the Uncertainty View popup menu (at the top left of the Result window), select the Sample view.



If the Variables are independent, the scatter plot points will fall randomly on the graph. If the Variables are totally dependent, the scatter plot points will fall along a single line. The strength of the relationship is indicated by the degree to which the points are

close to a line. If the line is straight, the relationship is linear; if the line is curved, the relationship is nonlinear.

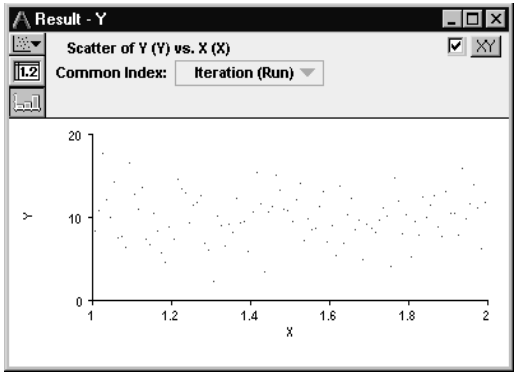
You can superimpose several scatter plots of *Y* in an array of uncertain quantities depending on *X*. The different quantities will be represented by differently colored dots or symbols.

Example

X: `Uniform(1,2)`

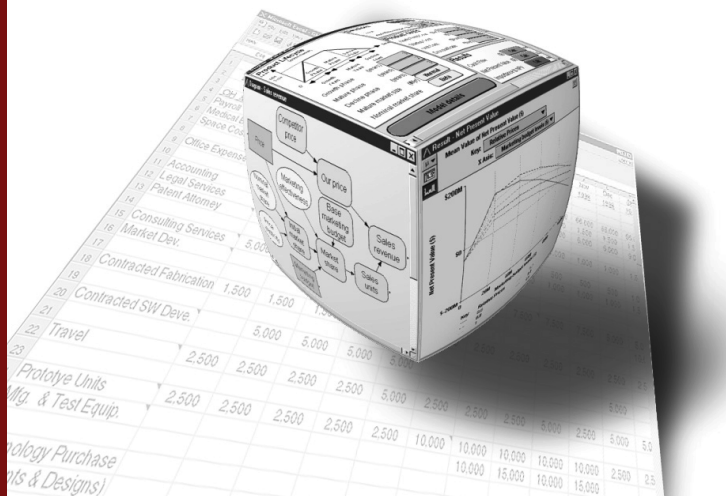
Y: `Normal(10,3)`

The resulting scatter plot, of two independent Variables, is:



Chapter 16

Modeling Changes over Time



In this Chapter

This chapter shows you how to use the system function **`Dynamic`** and the system Variable *Time.*

16: Modeling changes over time

A ***dynamic Variable*** is a quantity that changes over time—for example, the effect of inflation on car prices over a ten-year period. The system function `Dynamic()` and system Variable *Time* enable you to model changes over time.

Analytica Note: *Read Chapter 11, "Arrays and Indexes," before using these features.*

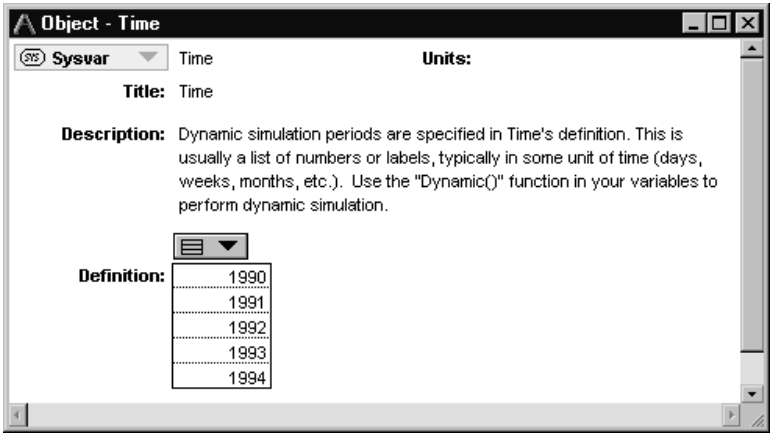
The term *dynamic* is used in this chapter to refer to the `Dynamic()` function.

The Time index

Dynamic simulation time periods are specified in the system Variable *Time*. To perform dynamic simulation, you must provide a definition for *Time*.

To edit the definition of *Time*, select **Edit Time** from the **Definition** menu to open the Object window for *Time*.

Time is defined by default as a list of three numbers 0, 1, and 2. You may want to define *Time* as a list of years, as in the following example:



Time becomes the index for the array that results from the `Dynamic()` function.

Analytica Note: *A model can have only one definition Time—that is, one set of time periods for* `Dynamic()` *functions. Any number of Variables in the model can be defined using* `Dynamic()`.

Using the Dynamic function

Dynamic (*initial1, initial2..., initialn, Expr*)

Performs dynamic simulation, calculating the value of its defined Variable at each element of *Time*. The result of `Dynamic()` is an array, indexed by *Time*.

Initial1, ...initialn are the values of the Variable for the first *n* time periods. *Expr* is an expression giving the value of the Variable for each subsequent time period. *Expr* can refer to the Variable in earlier time periods, that is, contain its own identifier in its definition. If Variable *Var* is defined using `Dynamic()`, *Expr* can be a function of `Var[Time-k]` or `Self[Time-k]`, where *k* is an expression that evaluates to an integer between 1 and t, and t is the time step at which Expr is being evaluated.

Analytica Note: *Square brackets ([ ]) are necessary around* `Time-t`.

The `Dynamic()` function must appear at the topmost level of a definition. It cannot be used inside another expression.

When a dynamic Variable refers to itself, it appears in its own list of inputs and outputs, with a symbol for cyclic dependency:  .

Library

Special

When to use

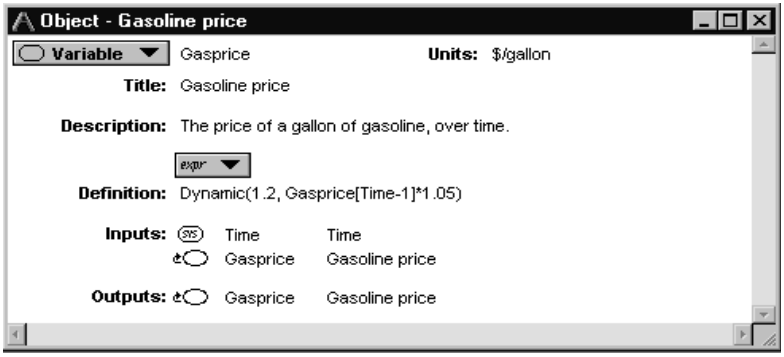
Use `Dynamic()` for defining Variables that are cyclically dependent. This is the only function in Analytica that permits reference to the same Variable, or other dynamic Variables, at earlier time periods.

Example

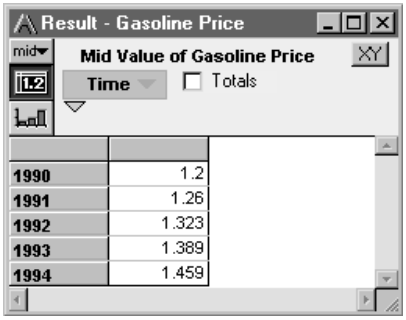
`Dynamic()` can be used to calculate the effect of inflation on the price of gasoline in the years 1990 to 1994.

If the initial value is $1.20 per gallon and the rate of inflation is 5% per year, then *Gasprice* can be defined as:

`Dynamic(1.2, Gasprice[Time-1] * 1.05)` or `Dynamic(1.2, Self[Time-1] * 1.05)`.



Clicking on the Result button and viewing the mid value as a table displays the following results:



For 1990, Analytica uses the initial value of *Gasprice* `(1.2)`. For each subsequent year, Analytica multiplies the value of *Gasprice* at `[Time-1]` by 1.05 (the 5 percent inflation rate).

x [*Time-k*]

Given a Variable *x* and brackets enclosing *Time* minus an integer *k*, returns the value for *x, k* time periods back from the current time period. This function is only valid for Variables defined using the `Dynamic()` function.

Library Special

More about the Time index

Reference to earlier time

`Time-k` in the expression `var[Time-k]` refers to the position of the elements in the *Time* index, not values of *Time*.

For example, if *Time* equals `[1990,1994,1998,2002,2006]`, then the value of `Gasprice[Time-3]` in year 2006 would refer to the price of gasoline in 1994, not 2003. When you refer to the *Time* Variable directly, not as an index, the expression refers to the values of *Time*. For example, the expression `(Time-3)` in 2006 is 2003.

The offset, *k,* may be an expression, and may even be indexed by *Time*. When *k* is indexed by *Time*, then the offset varies at different points in *Time*. However, *Slice(k,Time,t)* must be between 1 and *t-1*. It must be positive since the expression is not allowed to depend on values in the future (that have not yet been computed). It must be less than *t-1* since the expression cannot depend on values "before the beginning of time."

Defining time

There are three ways to define the *Time* index, each of which has different advantages:

- Sequence (the preferred method)
- List (numeric)
- List of labels (text)

Time as a sequence

Using the `Sequence()` function is the easiest way to define *Time* with equal intervals (see “List vs. list of labels” on page 219 and “Creating an array with an Edit Table” on page 225). The numeric values for *Time* can be used in other expressions.

Example

Time as a list (numeric)

When *Time* is defined as a numeric list, it will usually consist of increasing numbers. The intervals between entries can be unequal, and the values for *Time* can be used in other expressions.

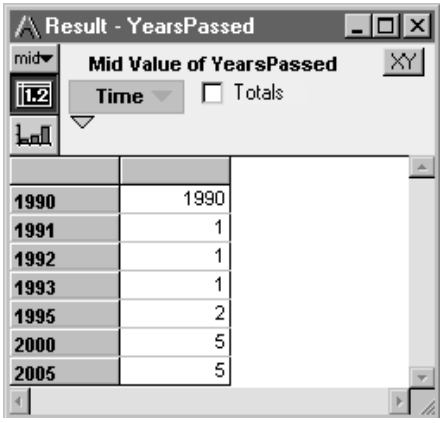
Example

Time:

1990
1991
1992
1993
1995
2000
2005

When you use time periods that differ by a value other than 1, typing `(Time-1)` won't provide the value of the previous time period. You can use the syntax `x[Time-1]` if you want to utilize a Variable indexed by *Time*, but if you want to perform an operation that depends on the difference in time between the current time period and the last one, you must first create a node that uncumulates the *Time* index:

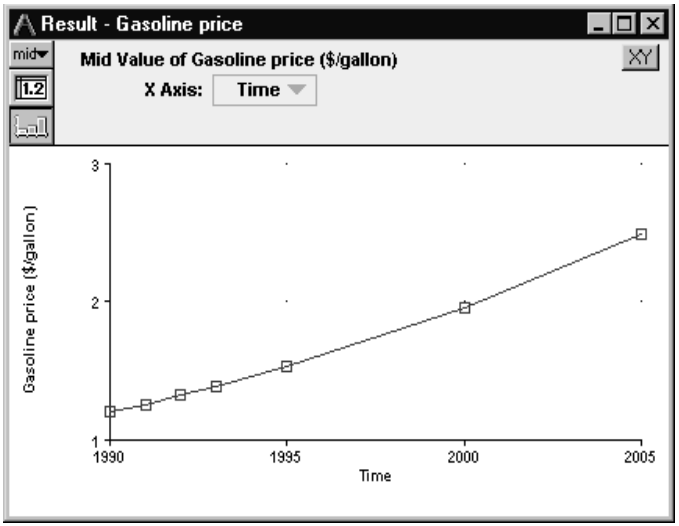
YearsPassed: `Uncumulate(Time)`



1990	1990
1991	1
1992	1
1993	1
1995	2
2000	5
2005	5

Now you can include this node in a dynamic expression that depends on the time between time periods. The following definition is equivalent to the one on page 362 but allows for changes in time period increments:

Gasprice: `Dynamic(1.2, Gasprice[Time - 1] * 1.05 ^ YearsPassed)` →



Time as a list of labels (text)

When *Time* is defined as a list of labels, *Time* values cannot be used in other expressions as numbers.

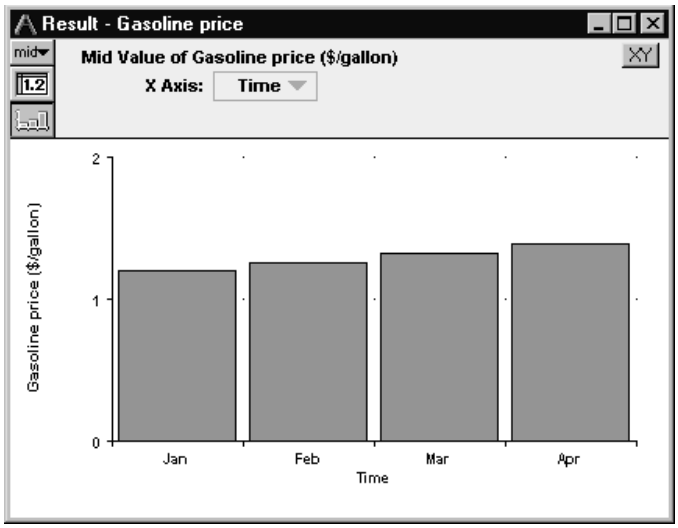
The resulting graph of any `Dynamic()` function, with the *x*–axis set to *Time*, will show the labels at equal *x*–axis intervals.

Example

Time:

Jan
Feb
Mar
Apr

Gasprice: `Dynamic(1.2, Gasprice[Time-1] * 1.05)` →



Using Time in a model

You can use *Time* like any index Variable; you can change only its title and definition. To include the *Time* node on a diagram:

1. Open the Object window for *Time* by selecting **Edit Time** from the **Definition** menu.
2. Select **Make Alias** from the **Object** menu (see “Alias nodes” on page 84).

When the *Time* node displays on a diagram, arrows from *Time* to all dynamic Variables display by default.

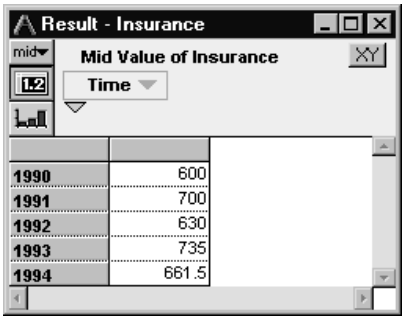
Initial values for Dynamic

A dynamic definition of *var* usually includes the expression `Self[Time-k]` or `var[Time-k]`, where *k* is the number of time periods to subtract from the current *Time* value. You must supply at least *1* initial value.

As an example, when *k* in `[Time-k]` is greater than 1, suppose your car insurance policy depends on the premium you paid two years ago. To calculate your payments in 1992, you must refer to the amount paid in 1990. A dynamic Variable representing such a rate for insurance needs two initial values for *Time*, such as:

Insurance:

```
Dynamic(600, 700, Insurance[Time - 2] * 1.05) →
```

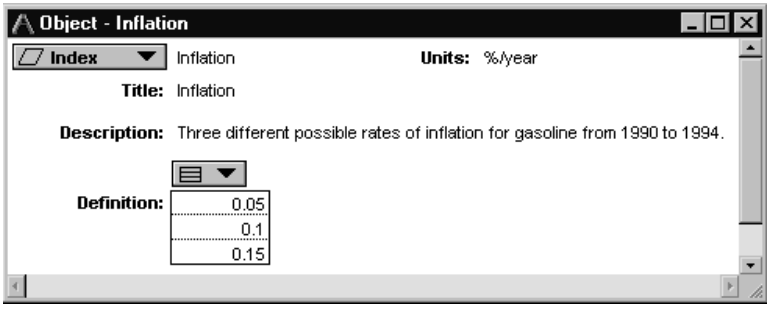


Using arrays in Dynamic

The initial value of a dynamic Variable—that is, the first parameter to the `Dynamic()` function—can be a number, Variable identifier, or other expression that evaluates to a single number, list, or array. Analytica evaluates a dynamic Variable starting from each initial value, in each time period, so the result is a correctly dimensioned array.

Example

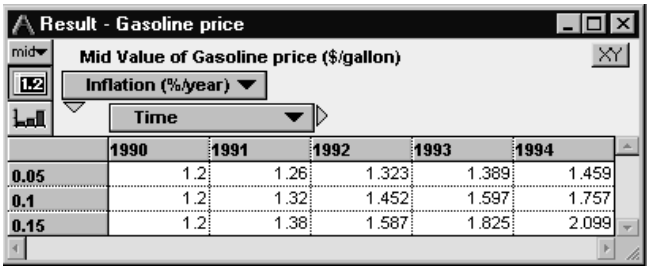
Expanding the example (see “Using the Dynamic function” on page 362), suppose the inflation rate of gasoline is uncertain. Instead of providing a single numerical value, you could define the inflation rate as a list:



Using the new *Inflation* Variable in the definition for *Gasprice*, the results show three different rates of increases in gasoline prices from 1990 to 1994:

Gasprice:

```
Dynamic(1.2,Gasprice[Time - 1] * (1 + Inflation)) →
```



	1990	1991	1992	1993	1994
0.05	1.2	1.26	1.323	1.389	1.459
0.1	1.2	1.32	1.452	1.597	1.757
0.15	1.2	1.38	1.587	1.825	2.099

Dependencies with Dynamic

All Variables with dynamic inputs are evaluated dynamically—that is, their results are arrays indexed by *Time*.

Example

A series of dynamic definitions produce equations for distance, velocity, and acceleration:

Acceleration: `-9.8`

Dt: `0.5`

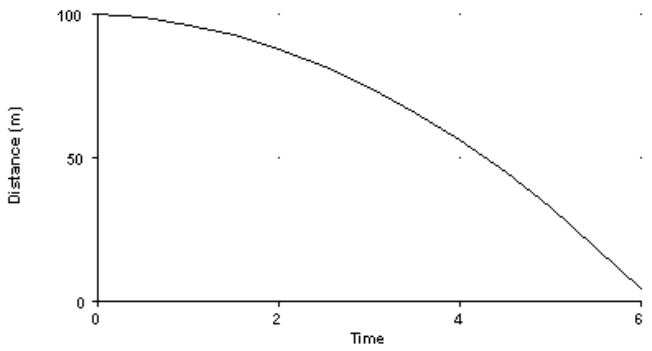
Time: `Sequence(0, 6, Dt)`

Velocity:

```
Dynamic(0, Self[Time-1] + Acceleration * Dt/2)
```

Distance:

```
Dynamic(100, Self[Time-1] + Velocity * Dt) →
```



Dynamic dependency arrows

If a Variable is dynamically dependent on another Variable, a gray arrow is drawn between the Variables.

To show or hide dynamic dependency arrows:

1. Select **Set Diagram Style...** from the **Diagram** menu to open the Diagram Style dialog box (see “Diagram Style dialog box” on page 121).
2. Click in the **Dynamic** checkbox to show dynamic arrows (or uncheck it to hide the arrows).
3. Click **OK** to accept the change.

Expressions inside dynamic loops

A dynamic loop is a sequence of Variables beginning and ending at the same Variable, with each consecutive Variable dependent on the previous one. At least one Variable in a dynamic loop is defined using the *dynamic* function.

When the definition of a Variable in a dynamic loop is evaluated, the definition is repeatedly evaluated in the context of *Time=t* (as *t* increments through the values of *Time*). The value for any identifer that appears in an expression is implicitly sliced at *Time=t* (unless it is explicitly offset in *Time*). As an example, suppose *A* is indexed by *Time*, and *X* is defined as:

```
dynamic(0, self[Time-1] + Max(A,Time) )
```

During evaluation, *A* would be a scalar at any given time point since it is implicitly sliced across *Time.* When *A* is not indexed by *Time*, *Max(A,Time)* simply returns *A,* so that the above expression is equivalent to

```
dynamic( 0, self[Time-1] + A )
```

To add the greatest value of *A* along *Time* in this expression, you must introduce an extra Variable to hold the maximum value, defined simply as *Max(A,Time)*, and ensure that the two Variables do not occur in the same dynamic loop.

If you attempt to operate over the *Time* dimension from within a dynamic loop, Analytica issues the warning: "Encountered application of an array function over the Time index from within a dynamic loop. The semantics of this operation may be different than you expect."

Uncertainty and Dynamic

Uncertain Variables propagate uncertainty samples during dynamic simulation. If an uncertain Variable is used in a dynamic simulation, its uncertainty sample is calculated only once, in the initial time period.

Example

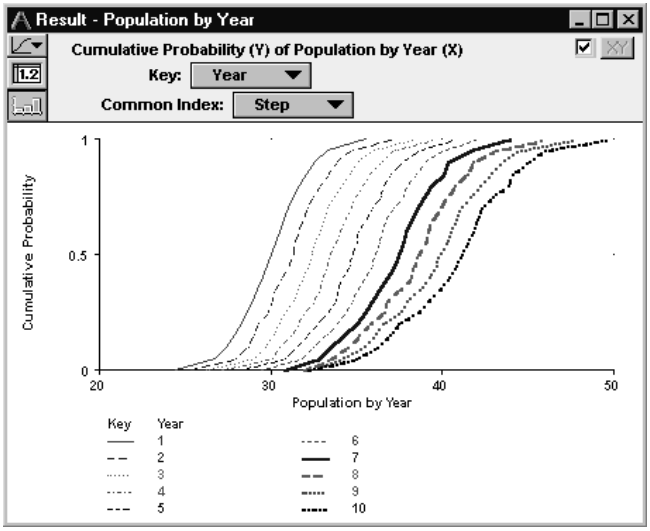
The following definitions model population changes over time:

Population: `Normal(30, 2)`

Birthrate: `Normal(1.2, .3)`

Time: `Sequence(1, 10, 1)`

Pop_by_year: `Dynamic(Population, Self[Time-1] + Birthrate)`



The uncertainty samples for *Population* and *Birthrate* are each calculated once, at the initial time period. The same samples are then used for each subsequent time period.

Resampling

If you want to create a new uncertainty sample for each time period (that is, resample for each time period), place the distribution in the last parameter of the `Dynamic()` function. For example, replace *Birthrate* with its definition in *Pop_by_year*:

Pop_by_year: `Dynamic(Population, Self[Time - 1] + Normal(1.2, .3))`

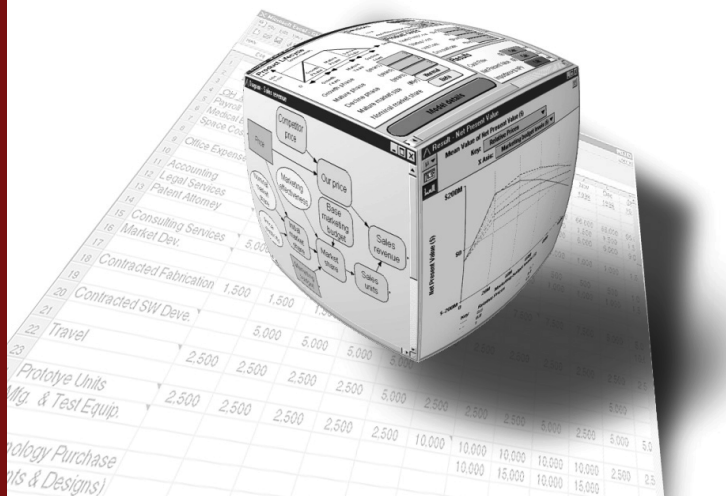
An alternative way to create a new uncertainty sample for each time period is to make *Birthrate* a dynamic Variable.

Birthrate: `Dynamic(Normal(1.2, .3), Normal(1.2, .3))`

Pop_by_year: `Dynamic(Population, Self[Time-1] + Birthrate)`

Chapter 17

Importing, Exporting, & OLE Linking Data



In this Chapter

This chapter shows you how to exchange data between Analytica and other applications.

OLE linking makes it possible to link data to and from external applications. With OLE linking, changes to inputs or results are automatically and instantaneously propagated between applications.

17: Importing, exporting, & OLE linking data

This chapter describes how to exchange data between Analytica and other applications. The primary methods are:

- Using the standard **Copy** and **Paste** commands
- Using OLE Linking
- Using the **Import** and **Export** commands

Copying and pasting

You can use the standard **Copy** and **Paste** commands with any modifiable Attribute of a Variable, module, or function.

Pasting data from a spreadsheet

To paste tabular data from a spreadsheet into an Analytica table:

1. Select a group of cells in a spreadsheet.
2. Select **Copy** from that program's **Edit** menu, to copy the data to the clipboard.
3. Bring the Analytica model to the front and open the Edit Table window you want to paste the data into.
4. Select a top-left cell or the same number of cells that you originally copied.
5. Select **Paste** from the **Edit** menu (Ctrl-V).

Analytica Note: *When copying a row of data from a spreadsheet into a one-dimensional table, transpose the data first so that you are copying it as a column of cells, not a row of cells.*

Pasting data from another program

To paste data from a program other than a spreadsheet:

- Use tab characters to separate items, and return characters to separate lines.

- Use numbers in floating point or exponential format. You can use the suffixes that Analytica recognizes (including K, M, and m; see page 177 for a comprehensive list). Dollar signs ($) and commas (thousands separators) are not permitted.

Copying a diagram

To copy an Influence Diagram, including the objects represented by the nodes:

1. Select the group of nodes you wish to copy.
2. Select **Copy** from the **Edit** menu (Ctrl-C). The objects that the nodes represent, as well as a picture of the selected nodes with all of the relevant arrows between the selected nodes, are copied to the clipboard.

To copy an entire Influence Diagram window, select **Copy Diagram** from the **Edit** menu. The entire Influence Diagram is copied as a Picture representation without copying the objects that the nodes represent.

Copying an Edit Table or Result Table

To copy data from an Edit Table or Result Table:

1. Open the window containing the table.
2. Select cells and choose **Copy** from the **Edit** menu (Ctrl-C).

To copy all the elements of a table in addition to the index elements, select **Copy Table** from the **Edit** menu. The entire multidimensional array is copied as a graphic and as a list of two-dimensional tables in a special text format (see “Edit Table data import/export format” on page 388).

Copying a Result Graph

To copy or export a Result Graph:

1. Open the Result window containing the graph.
2. Select **Copy** from the **Edit** (Ctrl-C) menu to copy a PICT representation of the graph.

Using OLE to link results to other applications

OLE (Object Linking and Embedding) is a widely used Microsoft technology that enables objects in two applications to be hot-linked, so that changes to the Object in one application cause the same changes in the other application. For example, by linking an array in Analytica to a table in a Microsoft Excel spreadsheet, any change to the array in Analytica model will automatically be reflected in the spreadsheet.

By using OLE linking, results from Analytica models can be linked into OLE compliant applications like Word and Excel. Linking data can save a great deal of work because it saves you from performing repeated copy and paste operations between Analytica and other applications whenever your model results change. Without OLE, if you copied Result tables from Analytica, pasted them into a Word document, and later you tweak your model results, you would need to re-copy and re-paste all those Result tables. However, if you link those tables using OLE, all the data in the Word document will update either automatically, or if you prefer, when you explicitly decide to update the data.

You may link any of the Result table views, *i.e.,* Mid, Mean, Statistics, Probability Density, Cumulative Probability and Sample table views. You may link any two-dimensional slice of a multi-dimensional table with the regular **Copy** command. For Result tables with more than two dimensions, you may decide to link the entire table as a series of two-dimensional tables using the **Copy table** option from the **Edit** menu. You may also link a rectangular region of cells that are a subset of a a two-dimensional table. However, you cannot link non-table data such as the information that is contained in the Object window or Attribute panel.

Linking procedure

Steps for linking result data from your Analytica model to an external OLE-compliant application are as follows. For concreteness, we'll assume here that the other application is Microsoft Excel.

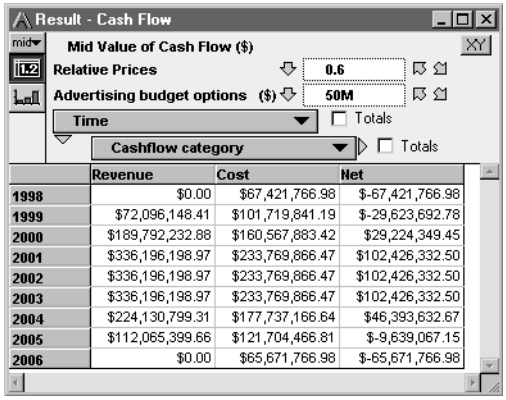
1. In the Analytica Result window, select the cells you want to link and choose **Copy** from the **Edit** menu.
2. From Excel, select the cells where you would like the Analytica data linked.

3. From Excel, choose **Paste Special…** from the **Edit** menu.
4. The Paste Special dialog box will appear.
5. In this box, choose the option **Paste Link**, select **Text** from the **As** list, and click the **OK** button.

You're done. Any changes to the source Result table will be propagated to the linked data in Excel. The procedure for linking Analytica model results to other OLE-compliant applications will be similar to the above steps.

Analytica Note: *The external application must support OLE-linking of tab-delimited textual data. Applications that do not support this format will not display "Text" as an option in Step 5 above, or will disable the* ***Paste Special...*** *menu item in Step 3.*

Detailed example of linking Analytica results



	Revenue	Cost	Net
1998	$0.00	$67,421,766.98	$-67,421,766.98
1999	$72,096,148.41	$101,719,841.19	$-29,623,692.78
2000	$189,792,232.88	$160,567,883.42	$29,224,349.45
2001	$336,196,198.97	$233,769,866.47	$102,426,332.50
2002	$336,196,198.97	$233,769,866.47	$102,426,332.50
2003	$336,196,198.97	$233,769,866.47	$102,426,332.50
2004	$224,130,799.31	$177,737,166.64	$46,393,632.67
2005	$112,065,399.66	$121,704,466.81	$-9,639,067.15
2006	$0.00	$65,671,766.98	$-65,671,766.98

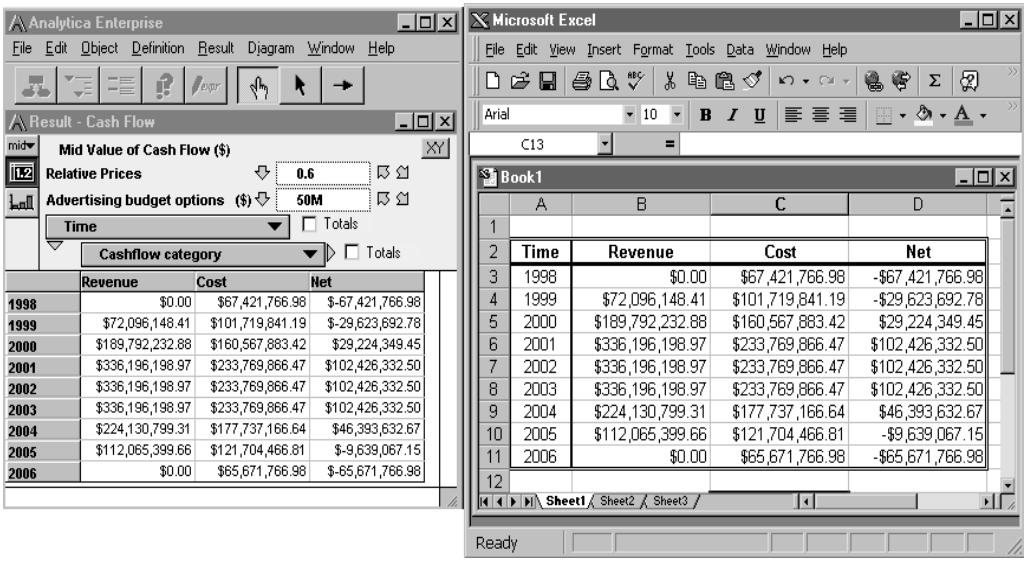
This example will itemize detailed steps for linking an Analytica Result table into an Excel spreadsheet. Suppose you would like to link the model results displayed above into an Excel spreadsheet. You can start by linking the column and row headers. Go to the node titled *Cashflow Category* and evaluate its result. Notice the result of node *Cashflow Category* is displayed as a column of cells, but you would like to have them linked into Excel as a row. Unfortunately you may not link this data as a row with a single Copy/Paste Special operation since Excel will not let you transpose the linked data from a column to a row. However, you can easily work around this limitation. Link the values into an unused portion of your spreadsheet or to a blank sheet using the linking procedure described in the previous section. In the cells where

you actually would like the labels to appear as a row, simply reference the linked cells. In other words, define the cells that will comprise the column headers for the linked table you are creating using the names of the corresponding linked cells.

Now it's time to link the values of *Time* as the row headers in your linked table. *Time* is an Analytica system Variable and one of the elementary ways to copy its values for linking is to create a node called *Time* and give it the definition *time*. Evaluate this node and then link the values displayed in the Result table using the linking procedure described in the previous section.

Linking the body of the table is just a straightforward application of the linking procedure. The number format of the cells will be preserved in fixed point format, but you may want to use Excel formatting to get the dollar sign and thousand separator displayed. Excel may switch to the exponential number format or display '########' if your columns are not wide enough.

The body of the table and its indexes (the row and column headers) are linked. For instance, if your Analytica model results change and you decide also to change the value of 'cost' to 'expense', these changes will be reflected in you're linked table in Excel (see the figure below).



Important notes about linking to Analytica results

Changing file locations

When moving linked files from one drive partition to another on the same machine or between two different computers, keep the relative paths the same. The simplest way to do this is to keep the linked model files and the other application files to which they are linked in the same folder.

Automatic vs. manual updating

OLE links are set for *automatic* updating by default, but you may change this setting to *manual*. We recommend this if the data is linked from an Analytica model with a lengthy re-computation time or to an application with a lengthy re-computation time.

To change a link's setting to *manual* in Word:

1. On Word's **Edit** menu, select **Links…**.
2. In the Links box that appears select the link(s) you're interested in adjusting.
3. Click on the radio button labeled **manual** and click the **OK** button.

In other OLE-compliant applications the steps for switching from *automatic* to *manual* updating should be very similar to the ones listed above.

You may also decide to set all your OLE links to be updated manually using a preference setting in Analytica. From the **Edit** menu, select **Preferences…**, then in the **Preferences** dialog box, uncheck the check box located on the bottom right labeled **Auto recompute outgoing OLE links**.

Using Indexes

Array-valued results that are to be linked should not have local indexes (created using the **Index..Do** construct). All indexes should correspond to index nodes in your diagram.

Number formatting

When linking data into OLE compliant applications, the number format will be the same as Analytica's format at the time of link

creation. However, if the linked Analytica data uses the default Suffix number format, the linking will convert the format to *Exponential,* which is more universally recognizable in other applications. In programs that have their own number formatting settings such as Excel, the number format will likely be adjusted according to the settings for the cells you are pasting into. However you must still be careful about losing significant digits (see next paragraph).

Precision is another important issue in number formatting. Before linking from Analytica, you should first adjust the number format so that it displays all the significant digits you would like to have in the other OLE-savvy application to which you are linking.

Refreshing links when Analytica model is not running

If you refresh the links between an Analytica model and another OLE-savvy application when the Analytica model is not running, the following events will occur:

1. A new instance of Analytica is launched
2. It automatically loads the Analytica model
3. It evaluates the Variables upon which the links are dependent,
4. It reactivates the links, and
5. It updates the linked data.

There are two ways to refresh the links this way. The first case occurs when a file with links is opened while the model file to which it is linked is closed, and you answer 'Yes' to the dialog box prompting you to update the linked data. The other way is if you are working with a file containing links to a model that is not running and you explicitly update the links. To explicitly update the links in Excel, you would select **Links…** from the **Edit** menu. Then in the **Links** dialog box, select the links you would like to refresh and click the **Update** button.

Linking data from other applications into Analytica

Using OLE linking, you may incorporate data originating in OLE-compliant applications as the input for nodes in your Analytica model. You accomplish this by linking the external data to Edit

tables in Analytica. Once again, this removes the need to perform numerous copy and paste operations each time the source data in the other application changes.

When linking data into Analytica, you may link data into any Edit table with less than three dimensions. When linking data in Edit tables you must link all the contents of the table; linking a subset of an Edit table is not supported. You may not link data from other applications to anywhere else than an Edit table in Analytica including the diagram windows, Object windows, and the Attribute panel.

Linking procedure

Steps for creating a linked Edit table in Analytica with data from an Excel spreadsheet:

1. In Excel, select the cells you want to link to Analytica and choose **Copy** from the **Edit** menu.
2. In Analytica, make the Edit table where you want the Excel data linked the front most window.
3. From the **Edit** menu or the right mouse button pop-up menu, choose **Paste Special…** and the Paste Special dialog box will appear.
4. In this box, choose the option **Paste Link**, select **Text** from the **As** list, and click the **OK** button.

The process for linking data from Word or other OLE compliant applications will be analogous to the steps just outlined.

Example of linking a table into Analytica

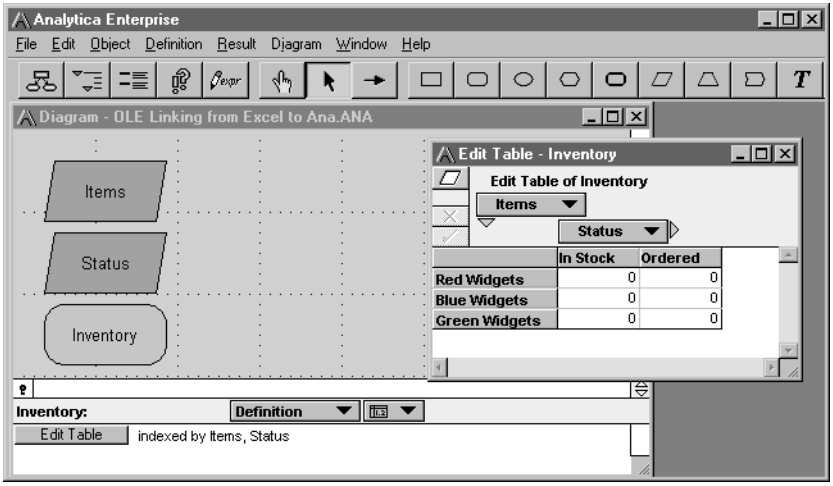
This section will itemize detailed steps for linking a table from Excel into Analytica by creating a node with a ‘Linked Table’ definition. Specifically, suppose you desire to link the Excel table displayed in the following figure into Analytica.

Start by creating two indexes in Analytica to store the row and column headers. Title the first index *Items* and the second *Status*. Select the node *Items* and then click the **Show definition** button on the toolbar (this is the button with the pencil icon) or right mouse menu. In the Attribute panel or Object window that appears, click on the *expr* popup menu and choose **List of labels**. Press the *down arrow* or *Return* key three times. This will give you three cells—*item 1*, *item 2* and *item 3*. In Excel, copy the three cells used as the row headers (*i.e.,* '*Red Widgets*', '*Blue Widgets*' and '*Green Widgets*'); return to Analytica and do a regular paste into the three cells of the definition for the index node 'Items'.

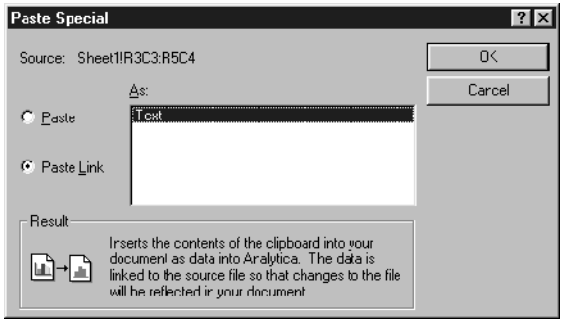
Now you need to copy the values of the column headers (*i.e.,* '*In Stock*' and '*Ordered*') into the definition for the index node *Status*. Since Analytica enforces strict dimension checking (*i.e.,* you cannot paste a 3 x 1 array of cells into a 1 x 3 array of cells), you are required to first convert the row into a column. You can accomplish this easily by copying the row, moving to an unused portion of the spreadsheet or onto a blank sheet, and choosing **Paste special...** from Excel's **Edit** menu. The **Paste Special** dialog box will appear and you need only select the **Transpose** check box on the bottom right. Click the **OK** button and you have converted the column header cells from a row into a column. Now copy this column, go back to Analytica, select the *Status* node, and click the **Show definition** toolbar button. Select the first cell '*item 1*' and choose **Paste** from the Analytica's **Edit** menu.

Since you've finished creating the indexes, you're ready to start on the node that will contain the linked table. Create a Variable node in Analytica and title it *Inventory*. With this node selected, click the **Show definition** button on the toolbar. In the Attribute panel or Object window that appears, click on the ***expr*** popup menu and choose **Table**. The **Indexes** dialog appears. In this dialog, select *Items* and click the ▼ button. This will move *Items* to

the **Selected Indexes** section. You also want to select *Status* and then click the ▼ button to make it a selected index as well. Click the **OK** button and an Edit table will appear as follows.

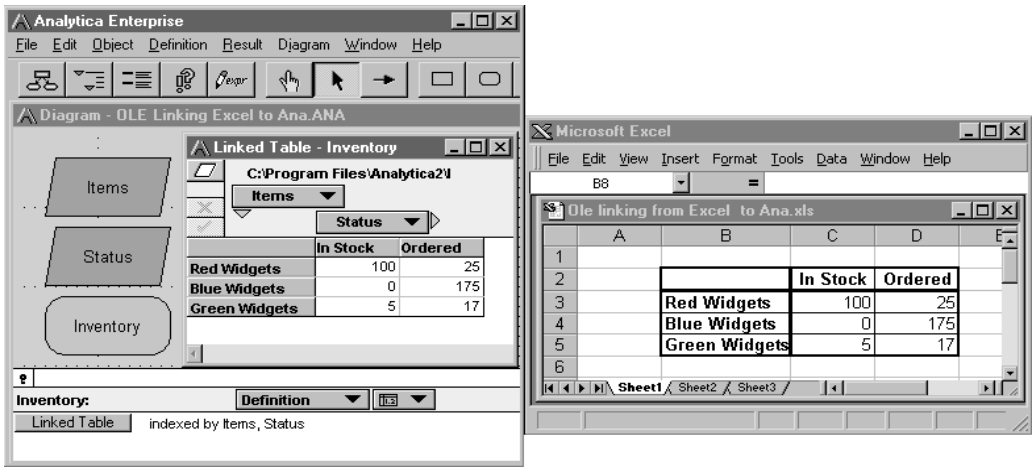


Go to Excel and select the numerical values displayed in the table and choose **Copy** from the **Edit** menu. Return to Analytica (while in Edit mode), click anywhere in the Edit table grid and choose **Paste Special…** from either the **Edit** menu or the right mouse menu. Now choose **Paste special…** from the **Edit** menu and the **Paste Special** dialog box comes into view. You want the settings in the box to be **Paste Link** and **Text** which are the default settings (see below). Click **OK**.



The caption for the table changes from 'Edit Table' to 'Linked Table' and you're done. If you arrange the applications windows so that you can see the source table in Excel and the Linked table in Analytica (see Figure 6), you can readily demonstrate that the link is activated. Change the value for 'Green Widget's Ordered'

from 2 to say 17. The corresponding value in Analytica's Linked table will change accordingly.



Analytica Note: *the data within the table is linked and will be updated automatically when altered, but the row and column headers are not linked and any changes to their values will have to be propagated using the standard cut and paste operations. Perform this by copying to the indexes used by the table, not to the table itself.*

Important notes about linking into Analytica Edit Tables

Changing file locations

When moving linked files on the same machine or between two different computers, keep the relative paths the same so that the files can locate each other. The simplest way to do this is to keep the linked model file(s) and the other application file(s) to which it is linked in the same folder.

Automatic vs. manual updating

OLE links are set for ‘automatic’ updating by default, but you may change this setting to ‘manual’. This may be desirable if the linked data is used in a model with a lengthy computation time. To change a link’s setting to ‘manual’ updating:

1. On Analytica's **Edit** menu, select **OLE Links…**.
2. In the Edit Analytica Links box that appears select the link(s).
3. Click on the radio button labeled **manual** and click the **OK** button.

Terminating links

You may want to terminate a link to a source file for a number of reason including if you do not have the source file or if you would like to edit the values in a Linked table. To break a link, bring up the **Edit Analytica Links** dialog, by choosing **OLE Links…** from the **Edit** menu. Select the link you would like to terminate and click the **Break Link** button.

Activating the other application

If you have linked data from an external application into Analytica, after loading Analytica you can make the other application visible using the **Open Source** button on the **OLE Links...** dialog, accessed through the **Edit** menu. If you implement a portion of your model in Analytica and a portion in an external application, with OLE links in both directions, you can make both applications simultaneously visible on the screen by loading the Analytica model first, then pressing the **Open Source** button to open the external application.

Importing and exporting

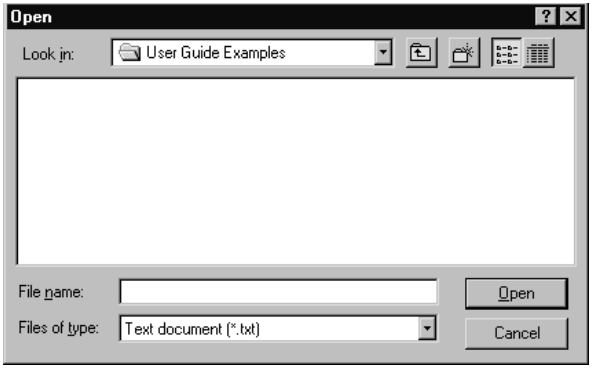
Importing a definition

To import a definition from a text file into expression format:

1. Select the definition field of the Variable in either the Object window or Attribute panel definition view.

   If the Variable is defined as a List, List of Labels, or Edit Table, select the cell(s) in which to import.

2. Select **Import...** from the **File** menu. A dialog box prompts you for the file name from which to import.



Importing into an Edit Table

To import data from a tab-delimited text file into an Edit Table:

1. Open the window containing the table.
2. Select cells and choose **Import…** from the **File** menu.

   A dialog box prompts you for the file name from which to import.

To import all the elements of a multidimensional table including the index elements, a special text format is required (see “Edit Table data import/export format” on page 388). This is also the format in which an Edit Table or Result Table is exported. The indexes of the table must have been previously created as nodes.

Exporting

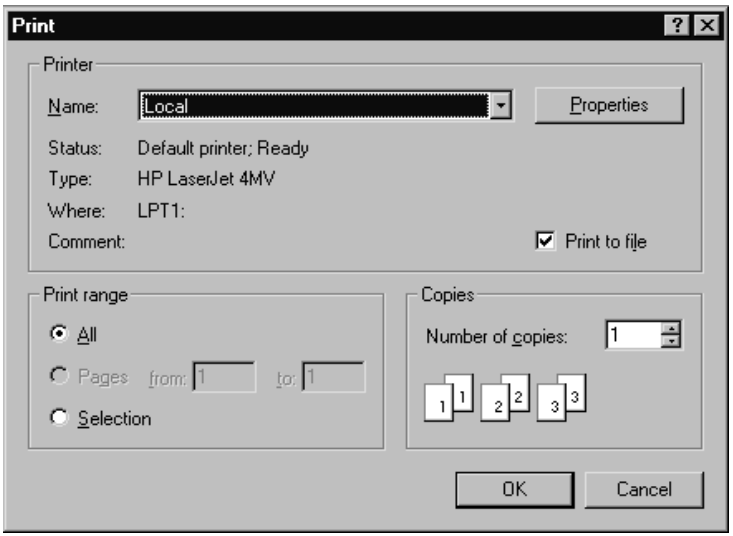
To export a Variable’s definition or result table to a text file, first be certain that the text file is closed.

1. Select the Variable to be exported from and open either the Object window, definition in the Attribute panel or Result window.
2. Select the definition field, list cell(s), or table cell(s) for exporting.
3. Select **Export** from the **File** menu. A dialog box prompts you for the file name to export to.

Printing to a file

Another way of exporting any Diagram window, Object window, or Result window to a file is to print to a file:

1. Select **Print** from the **File** menu.
2. Select **Print to File** and press *Enter* or click **OK**.



3. Enter the name of the file and the format for the file in the dialog box that appears.

Edit Table data import/export format

Multidimensional data being imported or copied into an Edit Table must be in a text file with the special format described in this section. This is also the format in which an Edit Table or result table is exported.

- **`TextTable`** is a keyword.
- **`Attribute`** is the name of the Attribute into which the data is to be pasted (usually Definition).
- **`Variable identifier`** is the identifier of the Variable node into which the data is to be pasted.
- **`Index`** identifier is the identifier of the index for this Variable. This node must already exist in the model.

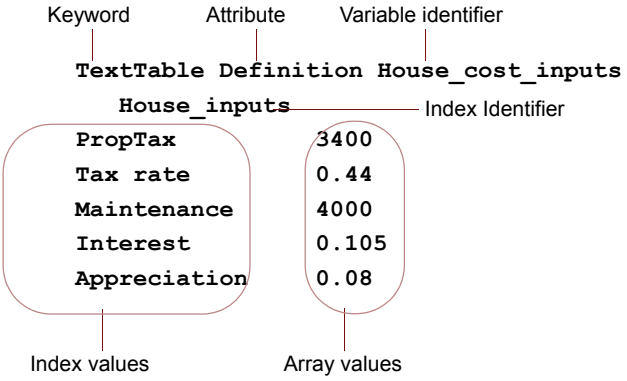
- Each index value and array value pair must be separated by tab characters.

One-dimensional array

The format for a one-dimensional array is:

```
TextTable <Attribute> <Variable identifier>
<line break>

<tab><Index identifier><line break>

<Index value><tab><Array value><line break>
```

Example

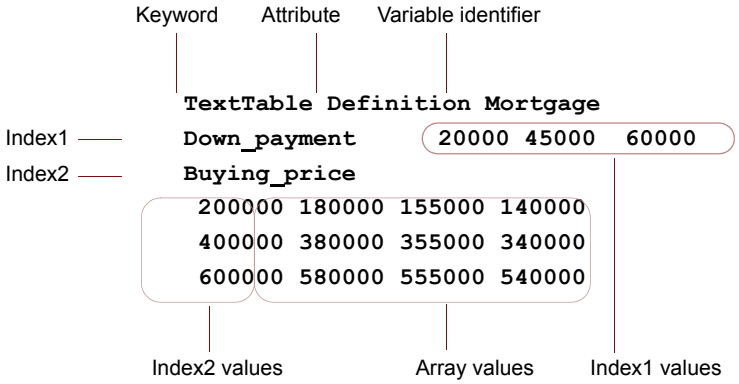


Two-dimensional array

The format for a two-dimensional array is:

```
TextTable <Attribute> <Variable identifier>
<line break>

<Index1 identifier>< tab><Index1 values separated by
tabs><line break>

<Index2 identifier><line break>

<Index2 value1><tab><Array values separated by
tabs><line break>

<Index2 value2><tab><Array values separated by
tabs><line break>

<Index2 valueN><tab><Array values separated by
tabs><line break>
```

Example



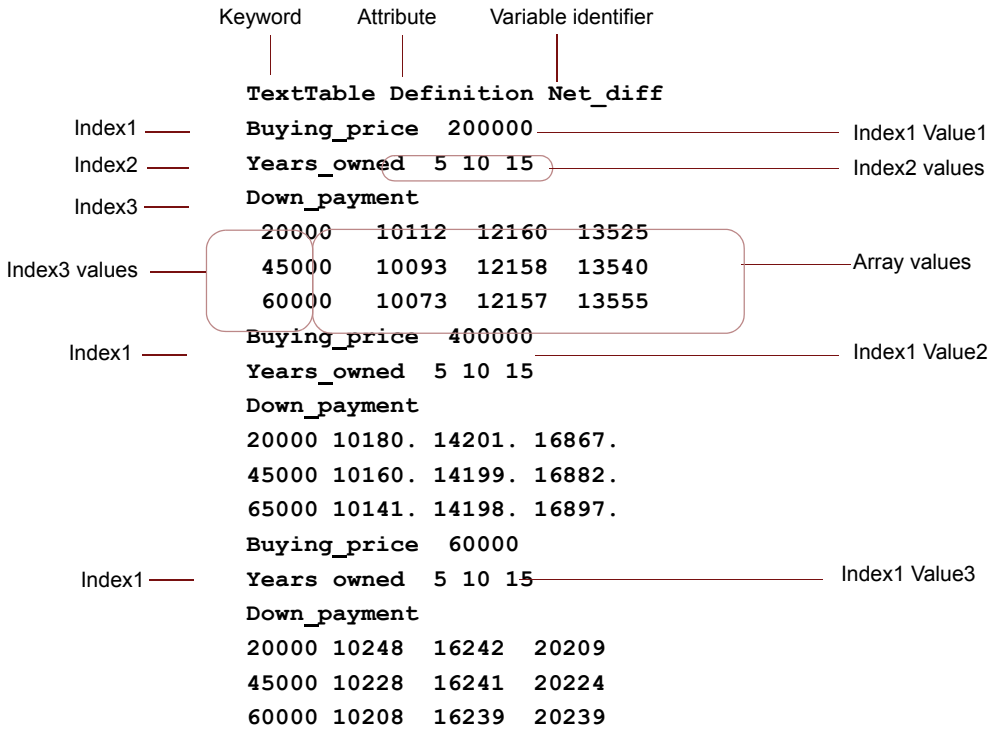
Three-dimensional array

The format for a three-dimensional array is:

```
TextTable <Attribute> <Variable identifier> <line
break>

<Index1 identifier><tab><Index1 Value1><line break>

<Index2 identifier><tab><Index2 values separated by
tabs><line break>

<Index3 identifier><line break>

<Index3 value1><tab><Array values separated by
tabs><line break>

<Index3 value2><tab><Array values separated by
tabs><line break>

<Index3 valueN><tab><Array values separated by
tabs><line break>

<Index1 identifier><tab><Index1 Value2><line break>

<Index2 identifier><tab><Index2 values separated by
tabs><line break>

<Index3 identifier><line break>

<Index3 value1><tab><Array values separated by
tabs><line break>

<Index3 value2><tab><Array values separated by
tabs><line break>

<Index3 valueN><tab><Array values separated by
tabs><line break>
```

and so on for each value of **Index1**.

Example



Number format

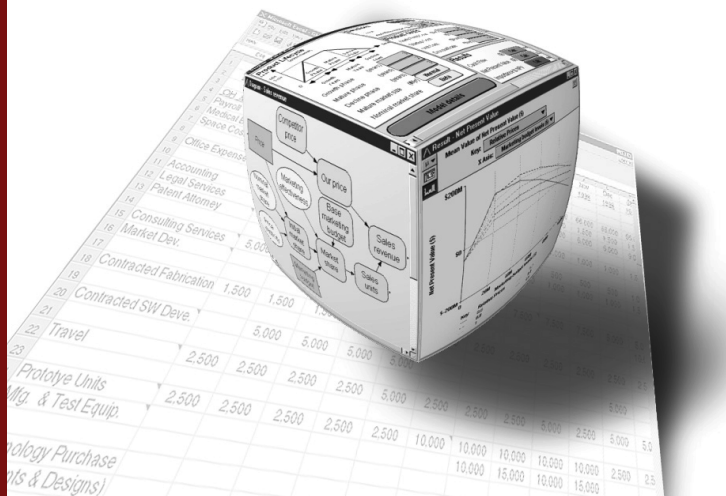
Numerical data can be imported in any format recognized by Analytica (see “Number Format dialog box” on page 135).

Numerical data will be exported in the format set for the table, with these exceptions:

- Suffix format numbers will be exported in scientific exponential format.
- Fixed decimal point numbers of more than 9 digits will be exported in scientific exponential format.
- If a date format begins with the day of the week, e.g., "Saturday, January 1, 2000", the weekday is suppressed: "January 1, 2000".

Chapter 18

Working with Large Models



Chapter 18

In this Chapter

This chapter shows you how to:

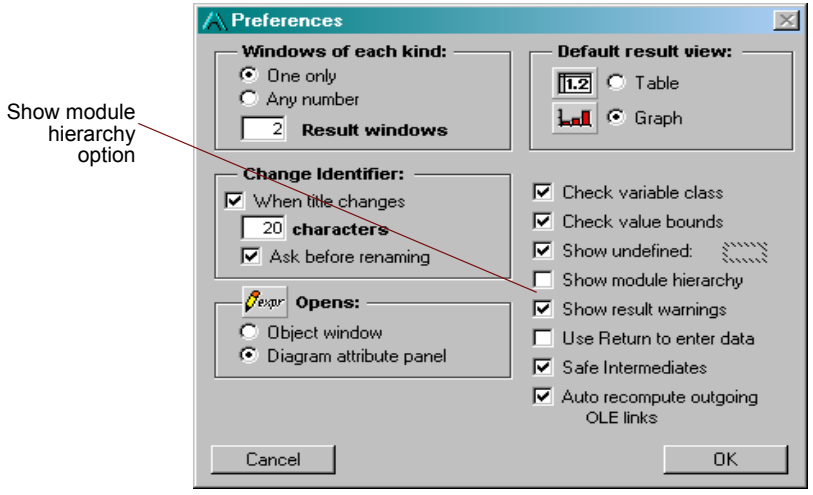
- Navigate large models
- Combine existing models into an integrated model

18: Working with large models

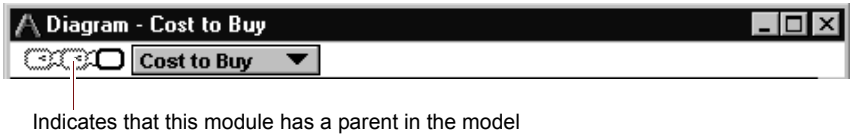
Large models, which include many Variables organized into multiple modules at several levels of hierarchy, can be challenging to find your way around. The first part of this chapter describes how to navigate larger models, using the hierarchy preference, the Outline window, and Variable input and output attributes. The second part of this chapter describes how to combine existing models into an integrated model.

Show module hierarchy preference

Often a large model has many layers of hierarchy. You can see the hierarchy depth of each module at the top of its Diagram window by setting a preference. Select **Preferences...** from the **Edit** menu to display the Preferences dialog box.

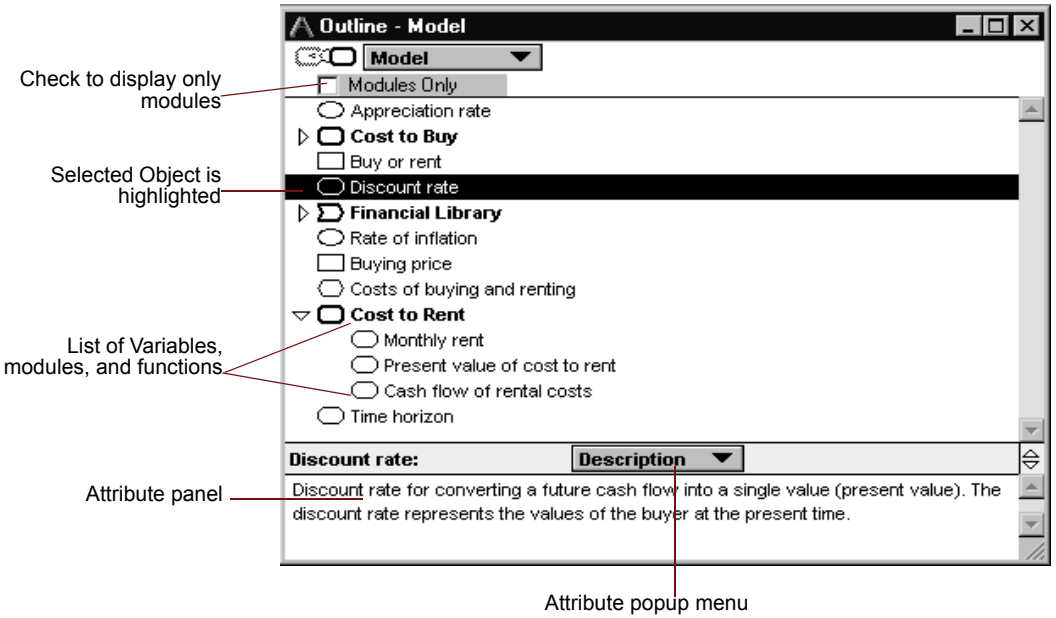


If you check the **Show module hierarchy** box, the top of the active Diagram window displays one or more module node shapes to indicate its hierarchy depth.



The Outline window

The Outline window displays a listing of the nodes inside a model. It can also show the module hierarchy as an indented list of modules. It provides an easy way to orient yourself in a large model and to navigate within it.



Opening the Outline window

To open the Outline window, click on the **Outline** button in the tool palette ( ).

The Outline window highlights the entry for the selected module or Variable.

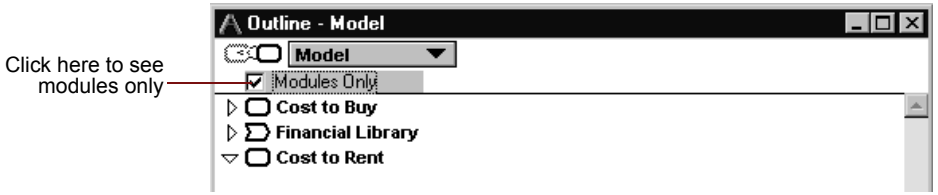
Opening details from an outline

To display a module's Diagram window, double-click on its entry in the outline.

To display a Variable's Object window, double-click on its entry in the outline.

Expanding and contracting the outline

By default, the outline lists all nodes in the model. Check the **Modules Only** box to list only the modules (exclude Variables and functions).



In the outline, each module entry has a triangle icon (▷ or ▽) to let you display or hide the module's contents.

▷ Indicates that the module's contents *are not* shown in the Outline window. Click on this icon to display the module's contents.

▽ Indicates that the module's contents *a*re shown as an indented list. Click on this icon to hide the module's contents.

Viewing and editing attributes

The Attribute panel at the bottom of the Outline window works just like the Attribute panel available at the bottom of a Diagram window (see "Displaying the Attribute" on page 35).

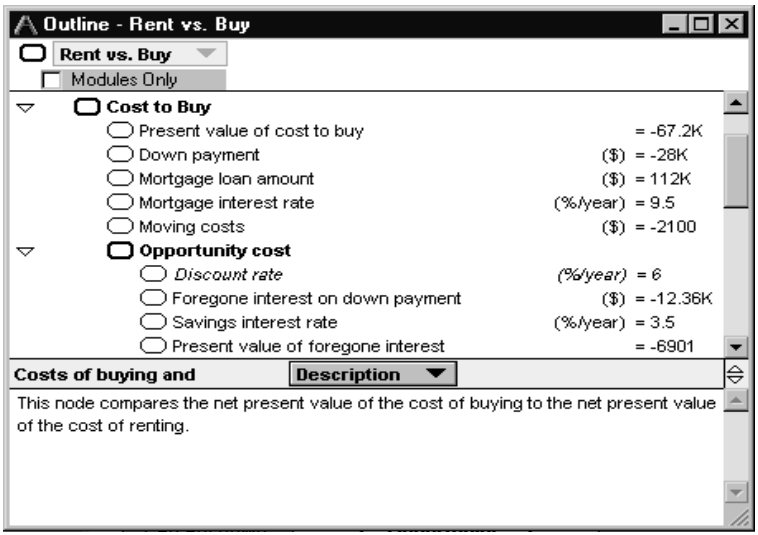
To view the attributes of a listed node:

1. Select the node by clicking on it.
2. Choose the Attribute to examine from the Attribute popup menu (see "The Attribute popup menu" on page 35).

If you edit attributes in this panel, the changes are propagated to any other Attribute panels and Object windows.

Viewing values

To see the Outline window with mid values, select **Show With Values** from the **Object** menu. Each Variable whose mid value has been evaluated and is a scalar will display in the window (see "Showing mid values" on page 36).



Finding Variables

To locate a Variable in its diagram, by identifier or by title, use the Find dialog box.

Find dialog box

To display the Find dialog box:

1. Select **Find...** (Ctrl-F) from the **Object** menu.

2. Choose the Attribute to search by: **Identifier** or **Title**.
3. In the text field, enter the identifier or title for the Analytica Object for which you want to search. You can enter an incomplete identifier or title, such as "down" for "Down payment."
4. Click on the **Find** button to initiate the search.

The Diagram window containing the Object found is displayed, with the node of the Object selected.

If the name you type does not match completely any existing identifier or title (depending on which Attribute you are searching), the first identifier or title that is a partial match will be displayed.

To find the next Object that is a partial match to the last identifier or title that you entered, select **Find Next** (Ctrl-G) from the **Object** menu.

To find an Object whose identifier matches the selected text in an Attribute field (such as a definition field), select **Find Selection** (Ctrl-H) from the **Object** menu.

Managing attributes

Every node in an Analytica model is described by a collection of ***attributes***. For some models, you may want to control the display of attributes or create new attributes. Some attributes apply to all classes (Variable, module, and function). Others apply to specific classes, as listed in the following table.

Attribute	Function	Module	Variable
Author		*	
Check	÷		÷
Class	*	*	*
Created		*	
Definition	*		*
Description	*	*	*
Domain			÷
File Info		*	
Help	÷	÷	÷
Identifier	*	*	*
Inputs	÷		÷
Last Saved		*	
Outputs	÷		÷

Attribute	Function	Module	Variable
Parameters	*		
Probvalue			÷
Title	*	*	*
Units	*		*
Value			÷
User-created (up to 5)	÷	÷	÷

Key:

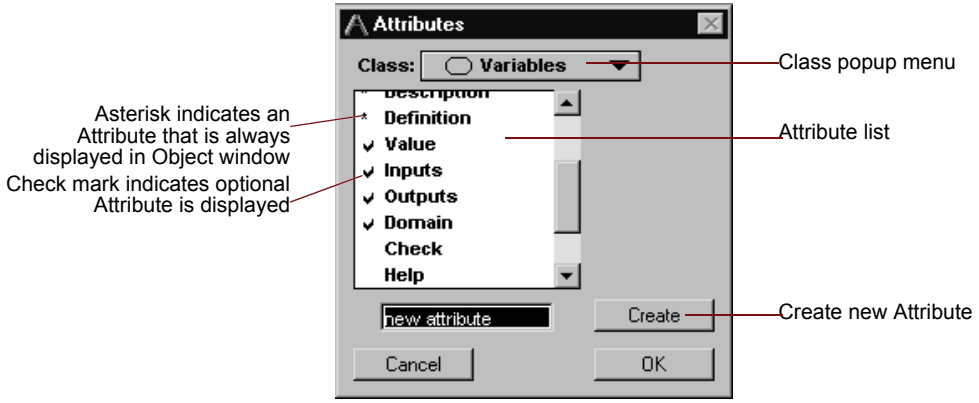
plain = modifiable by user	* = always displayed
italic = set by Analytica	√ = optionally displayed

For descriptions of the attributes, see Appendix C, “Glossary.”

Attributes dialog box

Use the Attributes dialog box to control the display of optional attributes in the Object window and Attribute panel and to define new attributes.

To open the Attributes dialog box, select **Attributes...** from the **Object** menu.



Class popup menu

Use this menu to select the Attribute list for Variables, modules, or functions.

Attribute list

This list shows attributes for the selected class. Attributes with an asterisk (*) are always displayed in the Object window and Attribute panel. Attributes with a check mark (√) are optionally displayed.

Displaying optional attributes

To display an optional Attribute in the Object window and Attribute panel, click on it once to select it, then click on it again to show a check mark.

To hide an optional Attribute, click on it once to select it, then click on it again to remove the check mark.

Creating new attributes

You can create up to five additional attributes. For example, you could use a Reference Attribute to include the bibliographic reference for a module or Variable.

To create a new Attribute in the Attributes dialog box:

1. Select **new Attribute** from the Attribute list to show the new Attribute Title field and the **Create** button.
2. Enter the title for the new Attribute in the Title field. The title can contain a maximum of 14 characters; 10 characters are the maximum recommended for visibility with all screen fonts.
3. Click on the **Create** button to define the new Attribute.

A newly created Attribute is displayed for modules, Variables, and functions. To control whether or not it is displayed for modules, Variables, or functions, select the Class popup menu in the Attributes dialog box, and turn the check mark on or off.

Renaming an Attribute

To rename a created Attribute:

1. Select it in the Attribute list. The Title field and the **Rename** button appear.
2. Edit the name of the Attribute in the Title field.
3. Click on the **Rename** button.

Referring to the value of an Attribute

Analytica includes the following function for referring to the value of an Attribute in a Variable's definition:

Attrib Of *x*

Returns the value of Attribute ***attrib*** of Object ***x***, where ***x*** may be a Variable, function, or module. For most attributes, including **Identifier**, **Title**, **Description**, **Units**, **Definition**, and user-defined attributes the result is a text value. For **Value** and **Prob-value**, the result is the value of the Variable (deterministic or probabilistic, respectively). For **Inputs**, **Outputs**, and **Contains** (an Attribute of a Module), the result is a vector of Variables.

You cannot refer to an Attribute of a Variable by naming the Variable in the definition of that Variable. Instead, refer to it as **Self**, for example:

```
Variable Boiling_point
Units: F
Definition: If (Units of Self) = 'C'
                THEN 100 ELSE 212

Boiling_point → 212
```

Library Special

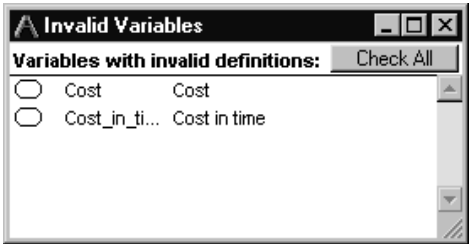
Example

```
Units of Time → 'Years'
```

Analytica Note: *Changes to attributes other than* ***Definition*** *do not automatically cause recomputation of the Variables whose definitions refer to those attributes. So, if you change Units of* `Boiling_point` *to C, the value of* `Boiling_point` *will not change until* `Boiling_point` *is recomputed for some other reason.*

Invalid Variables

To locate all Variables in a model with syntactically incorrect or missing definitions, select **Show Invalid Variables** from the **Definition** Menu.



Double-click on a Variable to open its Object window. From the Object window, you can edit the definition, or click on the Parent Diagram button ( ) to see the Variable in its diagram.

Using filed modules and libraries

Modules and libraries can be components of a model. If you are building several similar models, or if you are building a large model composed of similar components, you can create modules and libraries for reuse. (See Chapter 19, “Building Functions and Libraries” for details about libraries.)

To use a module or library in more than one model, create a ***filed module*** or ***filed library.***

Creating a filed module or library

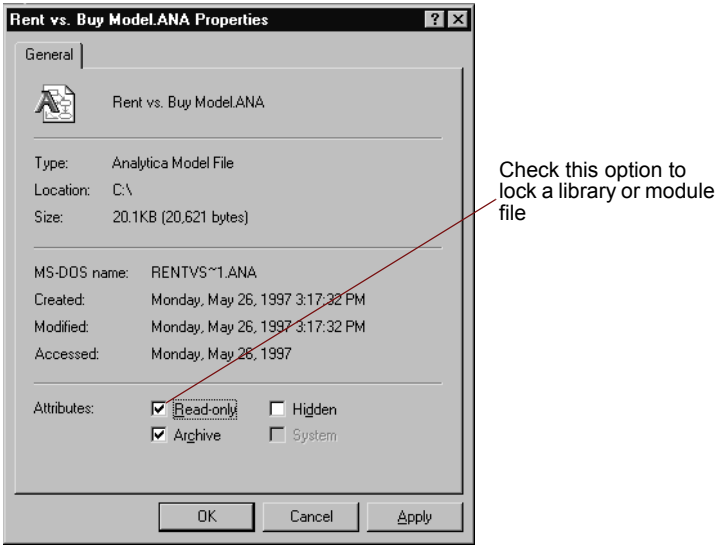
To create a filed module or library:

1. Create a module by dragging the module icon from the node palette onto the diagram, and give it a title.
2. Create functions and/or Variables in the module, or create them elsewhere and move them into the module.
3. Change the class of the module to **Module** ( ) or **Library** ( ) (see “Changing the class of a node” on page 86).
4. The Save As dialog box appears. Give the filed module or library a name and save it.
5. If you want the original model to load the new filed module or library the next time it is opened, save the model using the **Save** command.

Locking a filed module or library

To prevent a filed module or library from being modified, lock it:

1. Close the filed module or library, or close Analytica.
2. In Windows Explorer, select the filed module or library.
3. Select **Properties** from the **File** menu.



4. Check the Read-only check box.
5. Close the Properties window.

Adding a module to a model

To add a filed module to the active model, use the Add Module dialog box (see “Adding a module or library” on page 406). You can either embed a copy of the module or link to the original of the filed module.

Adding library to a model

To add a filed library to the active model, use the Add Module dialog box (see “Adding a module or library” on page 406). You can either embed a copy of the library or link to the original of the filed library.

When you select **Add Library** from the **File** menu, the Open File dialog box always opens up to fixed directory, regardless of the current directory settings or previous changes of directories. The directory is determined by a registry setting: …`/Lumina Decision Systems/Analytica/3.0/AddLibraryDir`, which is set by the Analytica installer to `INSTALLDIR/Libraries`.

Removing a module or library from a model

To remove a filed module or library from a model, first select it. Then, select **Cut** or **Clear** from the **Edit** menu. An embedded copy will be deleted; a linked original will still exist as a separate file.

Warning: *Any definitions that use a function in a deleted library or that have an input from a deleted module or library will have the deleted Object removed and will be changed to* `FunctionOf(`*remaining Variables*`)`*.*

Saving changes

After you have linked to a filed module or library, the **Save** command saves every filed module and library that has changed, as well as the model containing them, in their corresponding files.

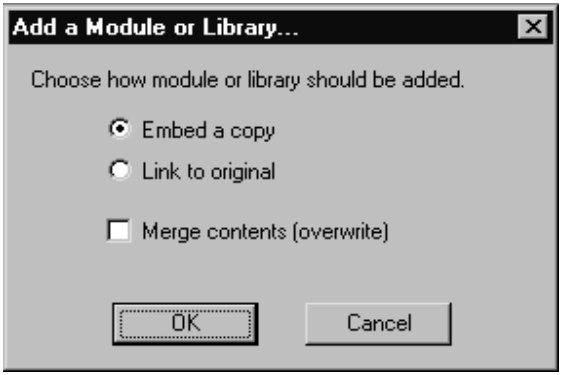
The **Save As** and **Save A Copy In** commands save only the active (top most window's) model, filed module or filed library.

Adding a module or library

Use the Add Module dialog box to add a filed module or library to the active model.

If you are adding a module, you open the Add Module dialog box by selecting **Add Module** from the **File** menu (Ctrl-L). If you are adding a library, you open the Add Module dialog box by selecting **Add Library** from the **File** menu.

The standard Open Model dialog box appears. Select the desired module in that dialog box. The following dialog box then appears:



Analytica Note: *Be sure that the selected model or module was saved with a class of* ***filed module*** *or* ***filed library****. If it was saved with a class of model, when it is linked to the root model, its preferences and uncertainty settings will overwrite the preferences and uncertainty settings of the root model.*

An added module or library may be either embedded or linked. You can optionally overwrite any nodes with the same identifiers.

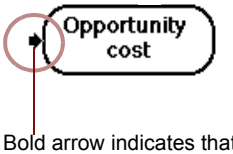
Embed a copy

Embeds a copy of the selected module or library in the active model, making it a part of, and saving it with, the model. Any changes to the copy will not affect the original filed module or library.

Link to original

Creates a link to the selected module or library, which can be separately opened and saved. If you make changes to a linked module or library from one model, the changes are saved in the original's file and any other models linked to the original will be affected by the changes.

A linked module or linked library has a bold arrow pointing into it on the diagram.



Bold arrow indicates that this is a linked module

Merge contents (overwrite)

Select this check box to overwrite existing objects in the active model with objects with the same identifiers from the added module or library. This is useful if the file being added contains updates from a previous version.

If you do *not* select this check box, and an Object in the file being added has the same identifier as one in the active model, Analytica will point that out and ask if you want to rename the Variable. If you click **Yes**, it will rename the Variable in the *existing* model, and update all definitions in the existing model to use the changed identifier. It will leave unchanged the identifier of the Variable in the module it is adding (which may contain definitions referencing that identifier that it has yet to read.) Hence, all the definitions in the existing model and added module will continue to reference the correct (original) Variables.

Combining models into an integrated model

Large models introduce a unique set of modeling issues. Modelers may want to work on different parts of a model simultaneously, or at remote locations. During construction, a large model may be more tractable when broken into modular pieces (modules), but all modules should use a common set of indexes and functions. Analytica provides the functionality required to support large-scale, distributed modeling efforts.

This section describes how to best use Analytica for large modeling projects and contains suggestions for planning a large model where responsibility for each module is assigned to different people (or teams).

Define public Variables

The first step to creating an integrated model is to define public Variables for use by all modules and agree on module linkages.

Every integrated model will have Variables that are used by two or more projects (for example, geographical, organizational, or other indexes, modeling parameters, and universal constants). These public Variables should be defined in a separate module, and distributed to all project teams. Each team uses **Add Module** (see “Adding a module or library” on page 406) to add the public Variables module to its model at the outset of modeling. Using a common module for public Variables avoids duplication of Variables and facilitates the modules’ integration.

Source control over the public Variables module must be established at the outset so that all teams are always working with the same public Variables module. Modelers should not add, delete, or change Variables in the public Variables module unless they inform the source controller, who can then distribute a new version to all modelers.

If multiple teams will be working on separate projects, it is essential that the teams agree upon inputs and outputs. Modelers must specify the input Variables, units, and dimensions that they are expecting as well as the output Variables, units, and dimensions that they will be providing. The indexes of these inputs and outputs should be contained in the public Variables module.

Create a modular model

By keeping large pieces of a model in separate, or filed modules, modelers can work on different parts of a model simultaneously. You can break an existing model into modules, or combine modules into an integrated model. In both cases, the result is a top-level model, into which the modules are added.

To save pieces of a large model as a set of filed modules, see “Using filed modules and libraries” on page 404.

To combine existing models into a new, integrated model:

1. Create or open the model that will be the top level of the hierarchy. This is the model to which all sub-models will be added.
2. Using **Add Module** (see “Adding a module or library” on page 406), add in the sub-models. Be sure to check the Merge option in the Add Module dialog box. Add the modules in the following sequence:
    - Any public Variable modules

- All remaining modules in order of back to front; that is:
  - first, the module(s) whose outputs are not used by any other module, and
  - last, the module(s) which take no inputs from any other module.

3. Save the entire integrated model, using the **Save** command.

The two alternative methods of controlling each module's input and output nodes so the modules can be easily integrated, are:

- Identical identifiers
- Redundant nodes

Identical identifiers

Assign the input nodes in each module the exact same identifiers as the output nodes in other modules that will be feeding into them. When you add the modules beginning with the last modules first (that is, those at the end of model flow diagram), the input nodes will be overwritten by the output nodes, thus linking the modules and avoiding duplication.

With identical identifiers, the individual modules cannot be evaluated alone because they are missing their input data. They can be evaluated only as part of the integrated model.

Redundant nodes

Place the output node identifiers in the definition fields of their respective input nodes. Due to the node redundancy, this method requires more memory than using identical identifiers, and it is therefore less desirable when large tables of data are passed between modules. However, since no nodes are overwritten and lost upon integration, this method preserves the modules' structural integrity, with both input and output nodes visible in each module's diagram.

With redundant nodes, each module can be opened and evaluated alone, using stand alone shells.

Stand alone shells

With redundant nodes, you can create a top-level model that contains one or more modules and the public Variables module plus

dummy inputs and outputs. Such a top-level model is called a ***stand alone shell*** because it allows you to open and evaluate a single module "standing alone" from the rest of the integrated model. Stand alone shells are useful when modelers want to examine or refine a particular module without the overhead of opening and running the entire model.

To create a stand alone shell for module *Mod1*, which is a filed module:

1. Open the integrated model and evaluate all nodes that feed inputs to *Mod1*.
2. Use the **Export** command (see “Importing and exporting” on page 386) to save the value of each feeding node in a separate file. Make a note of:
   - the identifier of each node and the indexes by which its results are dimensioned,
   - the identifiers of *Mod1*’s output nodes, if you want to include their dummies in the stand alone shell.
3. Close the integrated model.
4. Create a new model, to be the stand alone shell.
5. Use **Add Module** to add the public Variables module.
6. For each input node, create a node containing an Edit Table, using the identifier and dimensions of the feeding nodes you noted from the integrated model.
7. Use the **Import** command (see “Importing and exporting” on page 386) to load the appropriate data into each node's Edit Table.
8. Use **Add Module** to add *Mod1* into the stand alone shell.
9. To include output nodes at the top level of the hierarchy, create nodes there and define them as the identifiers of *Mod1*’s outputs.
10. Save the shell.

The shell now has all the components necessary to open and evaluate *Mod1*, without loading the entire model. As long as modelers do not make changes to the dimensions or identifiers of module inputs and outputs, they can modify a module while using the stand alone shell, and the resulting module will be usable within the integrated model.

Cautions in combining models

Identifiers

Every Object in a model must have a unique identifier. The identifiers of filed libraries and filed modules that you add to a model, as well as their Variables and functions, cannot duplicate identifiers in the root model. See “Merge contents (overwrite)” on page 408.

Created attributes

When you combine models with created attributes, the maximum number of defined attributes is five (see “Managing attributes” on page 400).

Location of linked modules and libraries

If the model will eventually be distributed to other computers, all modules and libraries should be on the same drive as the root model prior to being added to the root model. When the model is distributed, distribute it with all linked modules and libraries.

Managing windows

An Analytica model can potentially display thousands of Diagram, Object, and Result windows. To prevent your screen from becoming cluttered, Analytica limits the number of windows of each type that can be open at once. The default limits are:

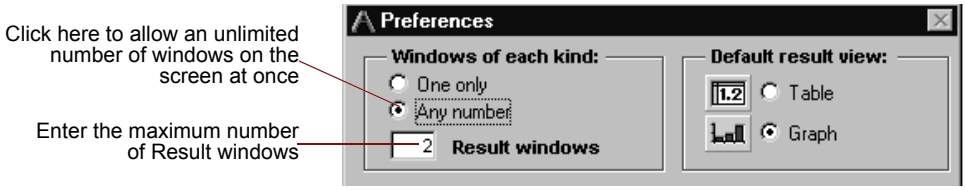
- The top-level Diagram window and not more than one Diagram window for each lower level in the hierarchy
- One Object window
- Two Result windows

The oldest window of the same type is deleted whenever you display a new window that would otherwise exceed these limits.

Overriding the limits on the number of windows

To display more windows of the same type, override the default limits in one of the following ways:

- Open a second Object window, or open a Diagram window without closing an existing Diagram window at the same level, by pressing the Control key (Ctrl) while you click or double-click to open the new window.
- Use the Preferences dialog box (see “Preferences dialog box” on page 88) to change the limits. Select **Preferences...** from the **Edit** menu.



In the "Windows of each Kind" area, select **Any number** instead of **One only**.

To display more Result windows and keep the limit on Diagram and Object windows, enter the maximum number of Result windows.

Optimization and speed-up

Numerous optimizations in the Analytica 3.1 engine result in a substantial increase in speed over Analytica 2.0. We have found a factor of between 1.5 and 4 reduction in the time to evaluate a model, depending on the functions and dimensionality of the model.

One example improvement is in **Subscript**. For example, the evaluation of `A[I=J]`, or equivalently `Subscript(A,I,J)`, is now approximately linear in the size of `J`, rather than proportional to the product of the sizes of `I` and `J`, as it used to be.

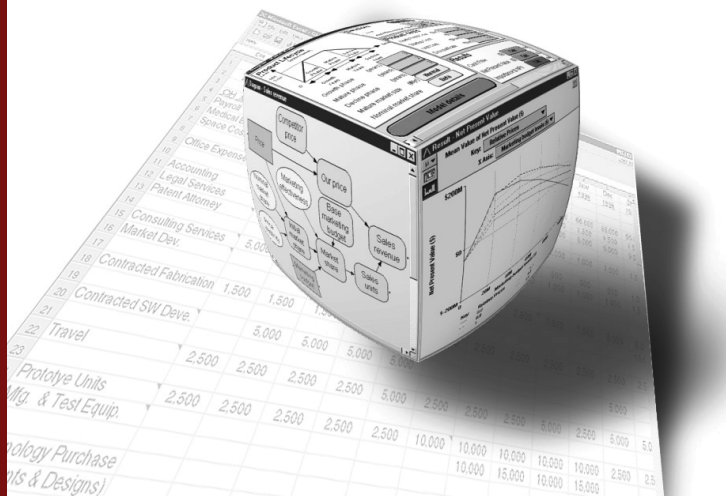
Rectangularization of intermediate results

In the most general case, intelligent array abstraction requires an extra internal, but somewhat costly, step during evaluation to make sure all intermediate arrays are fully rectangular. Skipping this step seldom has an impact on the final result, but can speed things up dramatically for certain models, especially those using dynamic simulation extensively. Unfortunately, in the very rare cases where it does make a difference, skipping the step can lead in incorrect results. By default, Analytica 3.1 uses the safe

but slower setting, for the system Variable, Rectangularize_inter, that controls this. You can clear this setting in the **Preferences** window for faster execution. See “Safe Intermediates” on page 92.

Chapter 19

Building Functions and Libraries



Chapter 19

In this Chapter

This chapter shows you how to:

- Create your own functions
- Create your own function libraries

19: Building functions and libraries

You can create your own functions to perform calculations you use frequently. A function has one or more parameters; its ***definition*** is an expression that uses these parameters. You can specify that the function check the type or dimensions of its parameters, and control their evaluation by using various ***parameter qualifiers***.

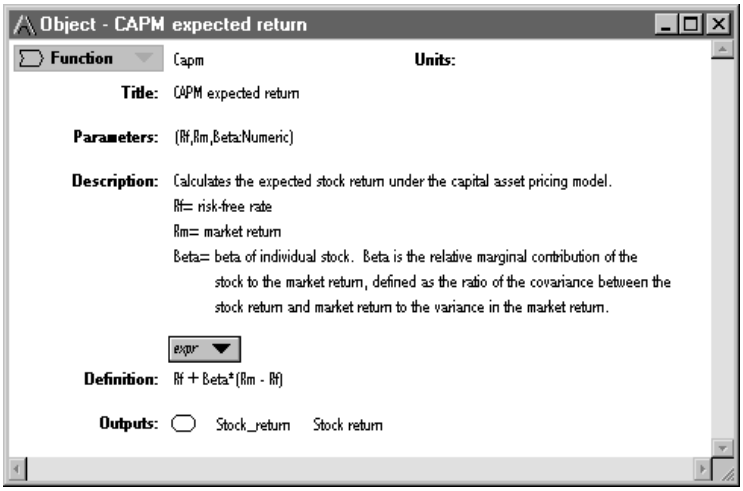
A ***Library*** is a a collection of functions grouped in a library file to extend Analytica's built-in functions for use for particular types of application. more than one model. Analytica is distributed with an initial set of libraries, available in the Libraries folder inside the Analytica folder on your hard disk. If you add a Library to a model, it will appear with its functions in the **Definition** menu, and these functions will appear almost the same as the built-in functions.

You may want to look at these libraries to see if they provide functions useful for your applications. You may also look at library functions as a starting point or inspiration for writing your own functions.

Analytica experts may create their own function libraries for particular domains. Other Analytica users can benefit from these libraries.

Example function

The following function, `Capm()`, computes the expected return for a stock under the capital asset pricing model.



Parameters The three parameters, *Rf*, *Rm*, and *Beta*, are qualified to be numeric.

Definition The definition is a simple expression using *Rf*, *Rm*, and *Beta*.

Sample usage You use the `Capm()` function in a definition in the same way you would use Analytica's built-in functions. For example, if the risk free rate is 5%, the expected market return is 8%, and *StockBeta* is defined as the beta value for a given stock, we can find the expected return according to the capital asset pricing model as*:*

Stock_return: `Capm(5%,8%,StockBeta)`

This definition functions equally well when *StockBeta* is an array of beta values. In this case, the result will be an array of expected returns.

Using a function

Position-based calling syntax

Analytica uses the standard *position-based syntax* for using, or *calling*, a function. You simply list the actual parameters after the function name, within parentheses, and separated by commas, in the same sequence in which they are defined. For example,

```
Capm(5%,8%,StockBeta)
```

evaluates function `Capm(Rf, Rm, Beta)` with `Rf` set to 5%, `Rm` set to 8%, and `Beta` set to `Stockbeta`.

Name-based calling syntax

Analytica also supports a much more flexible *name-based* calling syntax, using the names of the parameters:

```
Capm(beta: StockBeta, rf: 5%, rm: 8%)
```

In this case, we name each parameter, and put its actual value after a colon ":" after the parameter name. The name-value pairs are separated by commas ",". The parameters can be specified in any order, provided all the required parameters are mentioned. This method is much easier to read when the function has many parameters. It is especially useful when there are many parametes and some are optional. See page 428 for how to qualify parameters as optional.

You can mix positional and named parameters, as in:

```
Fu1(1, 2, D:4, C:3)
```

But, you may not list a positional parameter after a named parameter:

```
Fu1(1, D:4, 2, 3)
```

will display an error message.

This *name-based calling syntax* is analogous to Analytica's *name-based subscripting* for arrays to obtain selected elements of an array. In each case, you do not need to remember the particular sequence of parameters or indexes to understand how the model works.

Analytica Note: *Name-based calling syntax works for all user-defined functions. It also works for some of the built-in functions, including the Financial library, Text functions, Optimizer functions, EigenDecomp, and MatrixMultiply. We recommend that you do not use it for other built-in functions.*

Creating a function

To define a function:

1. Make sure the Edit tool is selected and you can see the node palette (see page 73).
2. Drag the Function node icon from the node palette into the Diagram area.

3. Title the node, and double-click on it to open its Object window.
4. Enter the new function's attributes (described in the next section).

Attributes of a function

Like other objects, a Function is defined by a set of attributes. Many of these attributes are the same as the attributes of Variables, including identifier, title, units, description, and definition, inputs, and outputs. It possesses one unique Attribute, **Parameters**, which specifies the parameters available to the function.

Identifier

If you are creating a library of functions, make a descriptive identifier. This identifier appears in the function list for the library under the **Definition** menu, and is used to call the function. Analytica makes all characters except the first one lower case.

Title

If you are creating a library of functions, limit the title to 22 characters. This title appears in the Object Finder dialog box to the right of the function.

Units

If desired, use the units field to document the units of the function's result. The units are not used in any calculation.

Parameters

The parameters to be passed to the function must be enclosed in parentheses, separated by commas. For example: `(x,y,z)`

The parameters may have type qualifiers (see the next section).

If you are creating a library of functions, use descriptive abbreviations for the parameters and give them a logical sequence. The parameters will appear in the Object Finder dialog box and they will be pasted when the function is pasted from its library in the **Definition** menu.

Analytica Note: *You can define up to 32 parameters and local variables in a Function.*

Description

The Description should first document what the function returns, and explain each of its parameters. If the Definition is not immediately obvious, the second part of the Description should explain how it works. The Description text for a function in a Library will also appears in a scrolling box in the bottom half of the Object Finder dialog.

Definition

The definition of a function is an expression or compound list of expressions. It should use all of its parameters. When you select the definition field of a function in Edit mode, it shows the **Inputs** pull-down menu that lists the parameters as well as any other Variables or functions that have been specified as inputs to the function. You can specify the inputs to a function in the same way as for a Variable—by drawing arrows from each input node into the function node.

Parameter qualifiers

Parameter qualifiers are keywords you may use in Parameters to specify for each parameter how, or whether, it should be evaluated when the function is used (called), and whether it should have a particular type of value, such as number, text value, or other. Other qualifiers specify whether a parameter should be an array, and if so, which indexes it expects.You can also specify whether a parameter is optional. By using qualifiers properly, you can help make functions easier to use, more flexible, and more reliable.

For example, consider this Parameters Attribute:

```
(A: Numeric ArrayType[I,J]; I, J: IndexType; C; D:
Optional Atomic TextType)
```

It defines five parameters, `A`, `I`, `J`, `C`, and `D`. `A` should be an array of numbers, indexed by parameters `I` and `J`. `I` and `J`, being separated by commas "," rather than semicolons ";" are subject to the same qualifier, `IndexType`. `C` has no qualifiers, and so can be of any type, or dimensions. The semicolon ";" between `C` and `D` means that the qualifiers following `D` do *not* apply to `C`. `D` has three qualifiers, specifying that it is `Optional`, `Atomic`, and a text value. See below for details.

Evaluation mode qualifiers

Evaluation modes control how or whether, Analytica evaluates each parameter when a function is used (called). The Evaluation Mode Qualifiers are:

`Context` Evaluates the parameter deterministically or probabilistically according to the current context. For example, consider

```
Function Fn1(x)
Parameters: (x: Context)

Mean(Fn1(x))
```

`Mean()` is a statistical function that always evaluates its parameter probabilistically. Hence, the evaluation context for `x` is probabilistic, and so `Fn1` will evaluate `x` probabilistically.

`Context` is the default evaluation mode used when no evaluation mode qualifier is mentioned. So, strictly, `Context` is redundant, and you can omit it. But, it is sometimes useful to specify it explicitly to make clear that the function should be able to handle the parameter whether it is deterministic or probabilistic.

Synonym: `Expr`

`DetermType` Evaluates the parameter determinstically, or in Mid mode, using the Mid (usually median) of any explicit probability distribution.

Synonyms: `Determ`, `Mid`, `DetermMode`, `MidMode`

`ProbType` Evaluates the parameter probabilistically, *i.e.,* in Prob mode, if it can. If you declare the dimension of the parameter, include the dimension `Run` in the declaration if you want the variable to hold the full sample, or omit `Run` from the list if you want the variable to hold individual samples. *E.g.*:

```
( A : ProbType[ In1, Run ] )
```

Synonyms: `Probabilistic`, `Prob`, `Uncertain`

`Sample` Evaluates the parameter probabilistically, *i.e.,* in Prob mode, if it can. If you declare the dimension of the parameter, include the dimension `Run` in the declaration if you want the variable to hold the full sample, or omit `Run` from the list if you want the variable to hold individual samples. *E.g.*:

```
( A : Sample[ In1, Run ] )
```

Synonym: **Samp**

IndexType The parameter must be an Index Variable, or a Dot-operator expression, such as **A.I**. You can then use the parameter as a local index within the function definition. This is useful if you want to use the index in a function that requires an index, for example **Sum(X, I)** within the function.

Synonym: **Index**

VarType The parameter must be a Variable, or the identifier of some other Object. You can then treat the parameter name as equivalent to the Variable, or other Object name, within the function definition. This is useful if you want to use the Variable in one of the few expressions or built-in functions that require a Variable as a parameter, for example, **WhatIf**, **DyDx**, and **Elasticity**.

Synonyms: **Variable**, **Var**, **VariableType**

Array qualifiers

An Array Qualifier can specify that a parameter is an array with specified index(es) or no indexes, in the case of Scalar.

Scalar The parameter expects a single number, not an array. Means the same as **Numeric Atomic**.

Synonyms: **Scalars**, **ScalarType**

Atomic If the actual parameter is an array, the function is called separately on each atomic element of the array. The results of all the calls to the function are reassembled into an array with the same indexes as the original parameter, which is returned as the overall result.

You might be tempted to use Atomic to qualify parameters of every function, just in case. Functions with Atomic parameters may take

longer to execute because they have to do all that disassembly of the array-valued parameters, multiple evaluations, and reassembly of the results. So, avoid using it in time-consuming functions except when really necessary.

ArrayType Dimensionality declaration, when present, forces Analytica to perform horizontal array abstraction over the parameters when additional dimensions are present. For example, if **Fu1** has the parameter declaration:

```
(A: ArrayType[Time])
```

and if A, when evaluated, contains dimensions other than Time, Analytica will loop over the other dimensions, ensuring that within the function A contains no dimension other than Time.

A dimensionality declaration usually the following the form:

```
ArrayType [ In1, In2, ... ]
```

Zero or more indexes can be specified between the square brackets. A zero- index declaration means that the value will be atomic when the function body is evaluated, and in this case the synonymous keyword Atomic may be used (synonyms Atom, AtomicType, AtomType). ArrayType also has the synonyms Array and Arr and is itself optional (e.g., one could write Numeric[In1] rather than Numeric ArrayType[In1].

Each index identifier listed inside the brackets may be either a global index Variable or another parameter explicitly qualified as an IndexType. For example the Parameters Attribute:

```
(A: ArrayType[Time, J]; J: IndexType)
```

specifies that parameter **A** must be an array indexed by **Time** (a built-in Index Variable) and by the index Variable passed to parameter **J**.

In the absence of an Array Qualifier, Analytica will accept an array-valued parameter for the function, and passes it into the function Definition for evaluation with all its dimensions (indexes). This kind of *vertical array abstraction* is usually more efficient for those functions that can handle array-valued parameters.

All Forces the parameter to have or be expanded to have all the Indexes listed. For example, in

```
X: ArrayType All [I, J]
```

the **All** qualifier forces the value of **X** to be an array indexed by the specified Index Variables, **I** and **J**. If **X** is a single number, not an array, **All** forces Analytica to convert it into an array with indexes, **I** and **J**, repeating the value of **X** in each element. Without **All** Analytica would simply pass the scalar value **X** into the function definition.

Type checking qualifiers

Type Checking Qualifiers make Analytica check whether the value of a parameter (each element of an array-valued parameter) has the expected type—such as, numerical, text, or reference. If any values do not have the expected type, Analytica gives an evaluation error at the time it tries to use (call) the function. The Type Checking Qualifiers are:

Numeric A number, including **+INF**, **-INF**, or **NaN**

Synonyms: **Number**, **Numbers**, **NumberType**, **NumericType**, **Num**, **Real**

Positive A number greater than zero, or **INF**.

TextType A text value

Synonyms: **Text**, **Textual**, **TexType**, **String**, **Strings**

ReferenceType A reference to a value, created with the **** operator.

Synonyms: **Ref**, **RefType**, **Reference**

Coerce

If you accompany a Type checking qualifier by the **Coerce** qualifier, it will try to convert, or *coerce*, the value of the parameter to the specified type. For example:

```
A : Coerce TextType ArrayType[I]
```

will try to convert the value of **A** to an array of text values. It will give an error message if any of the coercions are unsuccessful.

If the conversion cannot occur, an error is issued. The following coercions are allowed:

Undef to text	(blank)
Null to text	("Null")
Number to text	(uses number format for caller)
Text to Positive	(date vs. number based on number format)
Text to Number	(date vs. number based on number format)
Undef to Reference	(\Undefined)
Null to Reference	(\Null)
Number to Reference	(\X)
Text to Reference	(\Text)

Other coercions, including **Undef** or **Null** to **Number** or **Positive** will result in an error that the coercion is not possible.

Ordering qualifiers

The ordering qualifiers, **Ascending** or **Descending**, check that the parameter value consists of numbers in the specified order. Ordering also works for text values. **Ascending** means alphabetical order, and **Descending** means the reverse.

Ordering is not strict: That is, it allows successive elements to be the same. For example, [1,2,3,3,4] and [‘Anne’, ‘Bob’, ‘Bob’, ‘Carmen’] are both considered ascending.

If the value of the parameter does not have the specified ordering, or it is a scalar (not array) value, Analytica will issue an evaluation error when trying to evaluate the function call.

If the parameter has more than one dimension (other than Run), you should specify the index of the dimension over which to check the order, thus:

```
A : Ascending ArrayType[I]
```

Optional parameters

Any parameter may be optional if declared as optional by including the keyword **Optional**, or its synonym **Opt**, within the declaration. Optional parameters may appear anywhere within the declaration—they are not limited to the final parameters. So, for example, one could declare the parameters for Fu1 as

```
(A: Optional; B; C: Optional; D; E: Optional)
```

To call Fu1, you could use any of these examples:

```
Fu1(1,2,3,4,5)
Fu1(1,2,,4)
Fu1( ,2,,4)
Fu1( ,2,3,4,5)
```

Or you could use named-based calling syntax:

```
Fu1(B:2, D:4)
```

which is clearer and simpler.

When middle parameter are omitted, an empty space between commas must be included when the function is called. If all parameters following the last parameter provided are optional, placeholder commas are not necessary.

When a parameter is omitted, the value will have a special internal value such as **undef** (or a different internal value if a VarType is omitted). You can detect this inside the function definition using function **IsNotSpecified**. For example, the first line of the body of the function might read:

```
If IsNotSpecified(A) then A := 0;
```

Take care with omitted VarType and IndexType parameters. Some built-in functions may crash if passed an unspecified IndexType or varType parameter.

If a function with an omitted parameter value passes it as a parameter to a second function whose parameter is not optional, it displays a warning message. For example:

```
Fu1(A : optional) := Fu2(A)
Fu2(B) := B
```

```
Val := Fu1( )
```

will display the message

```
Error: Parameter B is not optional in function
Fu2.
```

Libraries

When you place functions and Variables in a library, the library becomes available as an extension to the system libraries. Its functions and Variables also become available. Up to eight user libraries can be used in a model.

There are two types of user libraries (see also “Changing the class of a node” on page 86):

- A library () is a module within the current model.
- A filed library () is saved in a separate file, and can be shared among several models.

Creating a library

To create a library of functions and/or Variables:

1. Create a module by dragging the module icon from the node palette onto the diagram, and give it a title.
2. Change the class of the module to library or filed library (see “Changing the class of a node” on page 86).
3. Create functions and/or Variables in the new library or create them elsewhere in the model and then move them into the library.

Functions and Variables in the top level of the library can be accessed from the **Definition** menu or Object Finder. Use modules within the library to hold functions and Variables (such as test cases) that will not be accessible to models using the library.

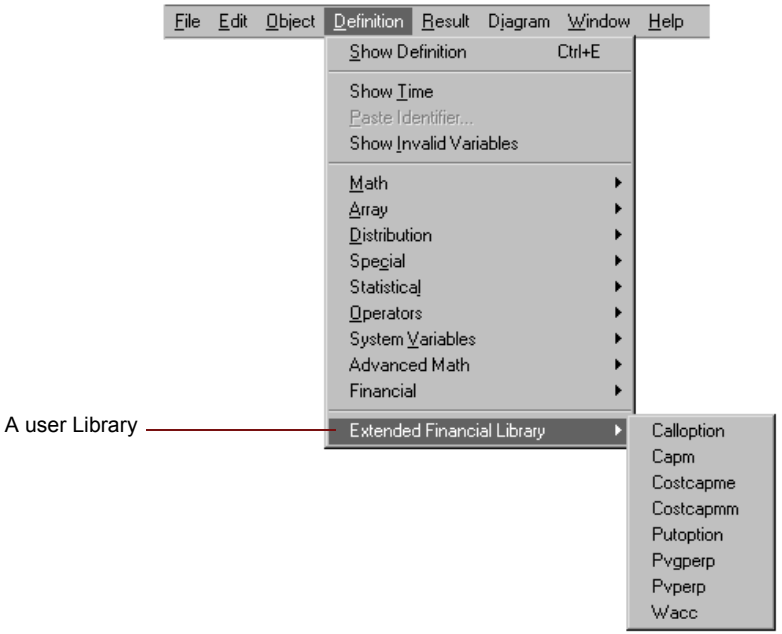
Adding a filed library to a model

Add a filed library to a model using the Add Module dialog box (see “Adding a module or library” on page 406).

Using a library

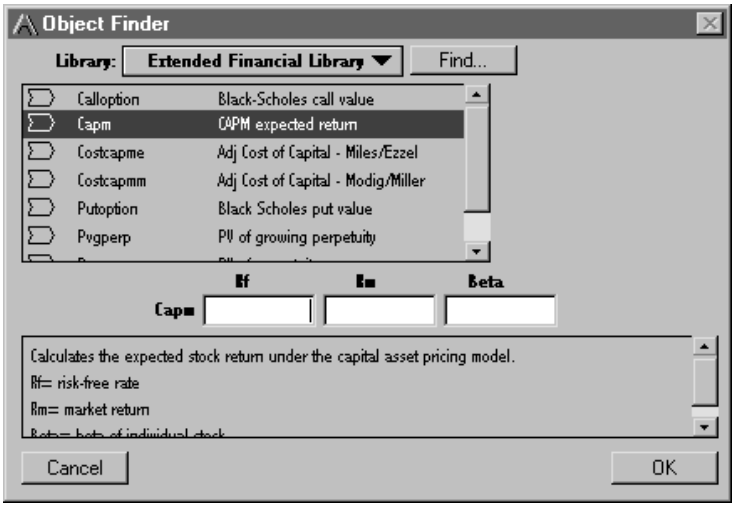
When defining a Variable, you can use a function or Variable from a library in any of the following ways:

- Type it in.
- Select **Paste Identifier** from the **Definition** menu to open the Object Finder.
- Select **Other** from the Expression popup menu to open the Object Finder.
- Paste from the library under the **Definition** menu.



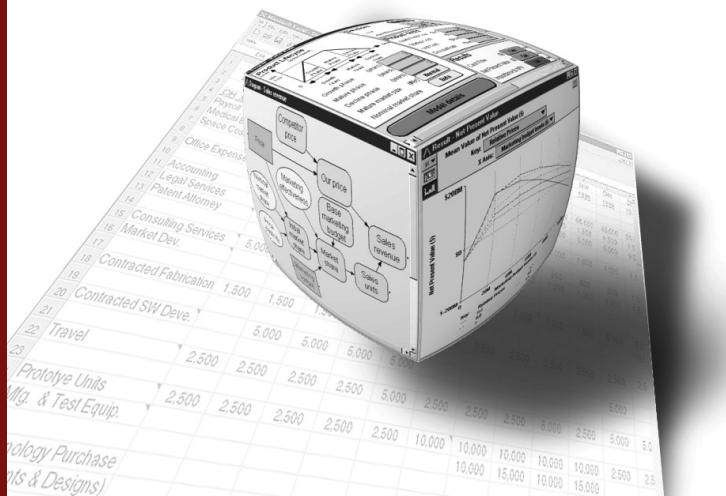
Example

Compare the way the `Capm()` function is displayed in the Object window (page 429) to the way it is displayed in the Object Finder:


Object Finder
Library:
Extended Financial Library
Find...
Calloption
Black-Scholes call value
Capm
CAPM expected return
Costcapme
Adj Cost of Capital - Miles/Ezzel
Costcapmm
Adj Cost of Capital - Modig/Miller
Putoption
Black Scholes put value
Pvgperp
PV of growing perpetuity
Rf
Rm
Beta
Capm
Calculates the expected stock return under the capital asset pricing model.
Rf= risk-free rate
Rm= market return
Cancel
OK

Chapter 20

Procedural Programming



In this Chapter

This chapter shows you how to use the procedural features of the Analytica modeling language.

20: Procedural programming

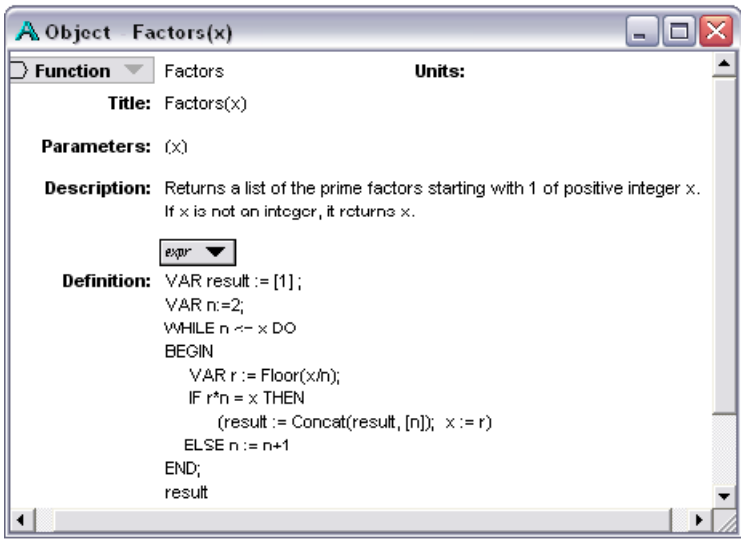
A ***procedural program*** is sequence of instructions to a computer. Each instruction tells the computer what to do, or in what sequence to execute the instructions. Most Analytica models are ***non-procedural***—that is, they consist of an unsequenced set of definitions of Variables. Each definition is a simple expressions that contain functions, operators, constants, and other Variables, but no procedural constructs controlling sequence of execution. In this way Analytica is like standard spreadsheet application, in which each cell contains a simple formula with no procedural constructs. Analytica determines the sequence in which to evaluate Variables based on the dependencies among them, somewhat in the same way spreadsheets determine the sequence to evaluate their cells. Non-procedural languages free you from having to think about it. Non-procedural models or programs are easier to write and understand because you can understand each definition (or formula) without worrying about the sequence of execution. Procedural programs, such as most programs in Fortran, Visual Basic, or C++, are much harder to write and understand.

However, procedural languages enable you to write more powerful functions that are hard or impossible without their procedural constructs. For this reason, Analytica 3.0 introduced a set of such constructs, providing it a general procedural programming language for those who want it. Together, these constructs provide much greater power and flexibility for creating definitions, and especially, for defining new functions in Analytica.

These constructs may only be used within the definition of a Variable or function to control the flow of execution within that Variable or function. They cannot affect other Variables or functions directly, and do not affect the flow of execution in other Variables or functions. Thus, the availability of these constructs does not affect the simple nonprocedural relationship among Variables and functions.

An example of procedural programming

The following function, Factors, computes the prime factors of an integer `x`. It illustrates many of the key constructs of procedural programming.



See below for an explanation of each of these constructs, and cross-reference to where they are.

Numbers identify features below:

```
Function Factors(x)
Definition:
```

1.
```
   VAR result := [1];
```

2.
```
   VAR n := 2;
```

3.
```
   WHILE n <= x DO
```

4.
```
   BEGIN
```

2.
```
      VAR r := Floor(x/n);
      IF r*n = x THEN
```

5.
```
         (result := Concat(result, [n]);
```

6.
```
         x := r)
      ELSE n := n + 1
```

4, 7.
```
   END; /* End While loop */
```

7, 8.
```
   result /* End Definition */
```

This definition illustrates these new features:

1. Var *x* := *e* construct defines a local Variable *x*, and sets an initial value *e*. See page 439 for more.

2. You can group several expressions (statements) into a definition by separating them with ';'s (semicolons). Expressions can be on the same line or successive lines. See page 439.
3. While *Test* Do *Body* construct tests condition *Test*, and, if true, evaluates *Body*, and repeats until condition *Test* is `False`. See page 446.
4. Begin *e1* ; *e2*; … End groups several expressions separated by ';'s—in this case as the body of a While loop. See page 439.
5. (*e1* ; *e2*; …) is another way to group expressions—in this case, as the action to be taken in the Then case. See page 439.
6. *x* := *e* lets you assign the value of an expression *e* to a local Variable *x* or, as in the first case, to a parameter of a function. See page 440.
7. A comment is enclosed between /* and */ as an alternative to { and }.
8. A group of expressions returns the value of the last expression—here the function `Factors` returns the value of `result`—whether the group is delimited by Begin and End, by '(' and ')', or, as here, by nothing.

Summary of programming constructs

Construct	Meaning	For more see:
e1 ; ***e2***; … ***ei***	Semicolons join a group of expressions to be evaluated in sequence.	page 439
Begin ***e1*** ; ***e2***; … ***ei*** **End**	A group of expressions to be evaluated in sequence.	page 439
(***e1*** ; ***e2***; … ***ei)***	Another way to group expressions.	page 439
m .. n	Alternative for **Sequence(m, n)**.	page 452
Var ***x*** := ***e***	Define local Variable ***x*** and assign initial value ***e***.	page 439
Index ***i*** := ***e***	Define local index ***i*** and assign initial value ***e***.	page 450
x := ***e***	Assigns value from evaluating ***e to*** local Variable ***x***. Returns value ***e***.	page 440
While ***Test*** Do ***Body***	While ***Test is*** `True`, evaluate ***Body*** and repeat. Returns last value of ***Body***.	page 446
/* ***comments*** */	Brackets enclose comments in definitions, alternative to curly braces { and }.	page 437
"***text***"	Double quotes enclose text values, as an alternative to single quotes, ’ and ’.	page 179
For ***x***[] := ***a*** DO ***e***	Assigns to local Variable ***x***, successive scalar values from array ***a*** and repeats evaluation expression ***e*** for each value. Returns an array of values of ***e*** with the same indexes as ***a***.	page 457
For ***x***[***i***, ***j…***] := ***a*** DO ***e***	Same, but each successive value assigned to ***x*** is indexed by the indices, [***i***, ***j*** …].	page 457
\ ***e***	Creates a reference to the value of expression ***e***.	page 458
\ [***i***, ***j …***] ***e***	Creates an array indexed by any indexes of ***e*** other than ***i***, ***j … of references*** to subarrays of ***e*** each indexed by ***i***, ***j …***.	page 462
# ***r***	Returns the value referred to by reference ***r***.	page 458

Programming constructs

Begin-End, ( ), and ';' for grouping expressions

As illustrated above, you can group several expressions (statements) as the definition of a Variable or Function simply by separating them by ';'s (semicolons). To group several expressions as a condition or action of **If *a* Then *b* Else *c*** or **While *a* Do *b***, or, indeed, anywhere a single expression is valid, you should enclose the expressions between **Begin** and **End**, or between '**(** 'and '**)**.

The overall value of the group of statements is the value from evaluating the last expression. For example,

```
(VAR x := 10; x := x/2; x - 2) → 3
```

Analytica will also tolerate a ';' after the last expression in a group. It still returns the value of the last expression. For example,

```
(VAR x := 10; x := x/2; x/2; ) → 2.5
```

The statements can be grouped on one line, or over several lines. In fact, Analytica does not care where new-lines, spaces, or tabs occur within an expression or sequence of expressions—as long as they are not within a number or identifier.

Var *v* := *e* : Defining a local Variable

This construct creates a local Variable ***v*** and initializes it with the value from evaluating expression ***e***. You can then use **v** in subsequent expressions in within this **context**—that is, in following expressions in this group, or nested within expressions in this group. You cannot refer to a local Variable outside its context—for example, in the definition of another Variable or function.

Analytica Note: *You can define up to 31 local variables in the Definition of a Variable.*

If ***v*** has the same identifier (name) as a global Variable, any subsequent mention of ***v*** in this context refers to the just-defined local Variable, not the global.

Examples

Instead of defining a Variable as:

```
Sum(Array_a*Array_b,N)/
```

```
(1+Sum(Array_a*Array_b,N))
```

define it as:

```
VAR t := Sum(Array_a*Array*b, N);
    t/(1+t)
```

To compute a correlation between *Xdata* and *Ydata*, instead of:

```
Sum((Xdata-Sum(Xdata,Data_index)/Nopts)*(Ydata-
    Sum(Ydata,Data_index)/Nopts),Data_index)/
    Sqrt(Sum((Xdata-Sum(Xdata,Data_index)/
    Nopts)^2, Data_index) * Sum((Ydata -
    Sum(Ydata,Data_index)/Nopts)^2,Data_index))
```

define the correlation as:

```
VAR mx := Sum(Xdata, Data_index)/Nopts;
VAR my := Sum(Ydata, Data_index)/Nopts;
VAR dx := Xdata - mx;
VAR dy := Ydata - my;
Sum(dx*dy,Data_index)
 /Sqrt(Sum(dx^2,Data_index)*
                  Sum(dy^2,Data_index))
```

The latter expression is faster to execute and easier to read.

The correlation expression in this example is an alternative to Analytica's built-in `Correlation()` *function (see "Correlation (X, Y)" on page 336) when data is dimensioned by an index other than the system index* `Run`.

Analytica Note: *The* **Var** *construct replaces the* **Using** ***x*** **Do** ***e*** *construct available prior to release 3.0. We encourage you to use the* **Var** *construct instead. Analytica 3.1 still supports the* **Using** ***x*** **Do** ***e*** *for upward compatibility.*

Assigning to a local Variable: *v* := *e*

The ':=' (assignment operator) sets the local Variable ***v*** to the value of expression ***e***. The assignment expression also returns the value of ***e***, although it is usually the effect of the assignment that is of interest.

The equal sign, '=', does not do assignment. It tests for equality between two values.

Within the definition of a function, you can also assign a new value to any of its parameters. Such changes to a parameter will

not affect any global Variables used as actual parameters in the call to the function.

Analytica Note: *You can assign only to local Variables previously defined using* ***Var*** *or* ***Index*** *constructs in the* ***current context****—that is, at the current or higher level in this Definition. You cannot make assignments to a global Variable—that is, a Variable created as a diagram node. This constraint prevents unexpected side effects: In Analytica, you can tell how any global Variable is to be computed simply by looking at its definition, without having to worry about any possible side effects from the evaluation of another Variable or function elsewhere in the model.*

There is one exception to this restriction that you can assign only to local Variables: Assignment to non-local Variables ***only*** *is allowed in a button script, or in a user-defined function that is called directly or indirectly from a button script. See "Assignment to a non-local Variable," below.*

Assignment to a non-local Variable

In an expression, the operator "`:=`" performs assignment. For example:

```
A := sum( c*sin(x) + d*cos(x), I )
```

assigns the result of evaluating the expression on the right-hand-side of "`:=`" to `A`. In a Variable node (or objective, chance, decision, *etc.*), this is only allowed if `A` is a local Variable. However, you can assign to Object nodes from a button script. The above expression, when evaluated, would change the definition of the node `A`.

Assignment to non-local Variables is only allowed in a button script, or in a user-defined function that is called directly or indirectly from a button script.

This feature has many uses. For example, it can be used to copy a result table into an Edit Table. For example, `(A:=B)` will copy the result of `B` into an Edit Table in `A`'s definition, assuming `B` is array-valued.

If you have a portion of your model that takes a very long time to compute, and a second half that uses those results, but you don't wish to wait for the first part to evaluate every time you load the model, then you can create a button script to copy the result from the first part into a separate node, breaking the dependency.

An example is the case where a time-consuming data analysis or machine learning algorithm analyzes data to create a learned parameter set, and a second portion of the model uses those learned parameters to classify new instances. Use of the assignment operator allows the learning algorithm to be separated from the performance element, even though both may be encoded in the same model.

For information on writing button scripts, consult the *Analytica Scripting Guide*, distributed with ADE. For simple scripts, it is not necessary to know much about the details of button scripts to make use of buttons and this assignment feature, as long as you understand a few points.

- A button is created from Analytica Enterprise by dragging a button Object from the toolbar onto a diagram. The functionality of the button is encoded in the button's script Attribute.
- The script Attribute contains commands written in **Typescript**. Typescript is a different language from the expression syntax used in other Object definitions. Typescript allows one statement per line; a multi-line statement must contain a tilde (~) as the last character of each line of the statement, except for the final line of the statement.

  An easy way to make use of buttons in most applications is to specify a user-defined function with the desired functionality (including assignments) and place a call to the user-defined function in the script Attribute. A call to a function is valid in Typescript. If the button script contains only simple assignments, then it may be convenient to place these in button script directly.
- The "`:=`" operator can be used directly in Typescript, but has a slightly different functionality than it has elsewhere: In Typescript the right-hand side is not evaluated. So set the **script** Attribute to:

  ```
  Va1 := 1 + 2
  ```

  when the button is pressed, the definition of `Va1` will become the expression "`1+2`". However, if you surround the entire thing by parens, as in:

  ```
  (Va1 := 1+2)
  ```

  Then the script will be treated as an expression, rather than Typescript, the right-hand side will be evaluated, and the definition of `Va1` will be the number 3. As a general rule, if you

wish to use expressions in Typescript, you should always surround the entire expression with parentheses. See the *Analytica Scripting Guide*, distributed with ADE for further details. You can download the Analytica Scripting Guide *from the* Analytica Download Center:
`http://www.lumina.com/ana/support/download.htm`

Iteration loops and recursion

The two functions in this section can be used to control specialized evaluation of arrays.

For *Temp*:= *I* Do *Expr*

For each successive value of index *I*, assigns that value to Variable *Temp*, and evaluates expression *Expr*. *Expr* may refer to *I* and/or *Temp*. *Temp* is a local or temporary Variable that can be referred to only within the expression *Expr*.

The result of the *For* control function is an array indexed by *I* containing the results of evaluating *Expr*. *I* must be an index Variable, or be defined as a list or `Sequence()`.

If you make appropriate use of the intelligent array features described earlier in this and preceding chapters, you will rarely need to use *For* structure (unlike in conventional computer languages, which require extensive use of For loops and related control structures for handling arrays). `For` is sometimes useful in these specialized cases:

- To avoid the attempted evaluation of out-of-range values by nesting an `If-Then-Else` inside a `For`.
- To apply an Analytica function that requires a scalar or one- or two-dimensional array input to a higher-dimensioned array.
- To reduce the memory needed for calculations with very large arrays by reducing the memory requirement for intermediate results.

Library Special

Examples **Avoiding out-of-range errors:** Consider the following expression:

```
If X<0 Then 0 Else Sqrt(X)
```

The **If-Then-Else** is included in this expression to avoid the warning "Square root of a negative number." However, if *X* is an array of values, this expression may not avoid the warning since **Sqrt(X)** is evaluated before **If-Then-Else** selects which elements of **Sqrt(X)** to include. To avoid the warning (assuming *X* is indexed by *I*) the expression can be rewritten as

```
For j:=I do
    If X[I=j]<0 then 0 else Sqrt(X[I=j])
```

or as (see next section):

```
Using y:=X in I do
    If y<0 Then 0 else Sqrt(y)
```

Situations like this can often occur during slicing operations. For example, to shift *X* one position to the right along *I*, the following expression would encounter an error:

```
if I<2 then X[I=1] else X[I=I-1]
```

The error occurs when **X[I=I-1]** is evaluated since the value corresponding to *I-1=0* is out-of-range. The avoid the error, the expression can be rewritten as:

```
For j:=I do
    If j<2 then X[I=1] else X[I=j-1]
```

Out-of-range errors can also be avoided without using **For** by placing the conditional inside an argument. For example, the two examples above can be written without **For** as follows:

```
Sqrt(if X<0 then 0 else X)
X[I=(if I<2 then 1 else I-1)]
```

Dimensionality reduction: **For** *can be used to apply a function that requires a scalar, one- or two- dimensional input to a multi-dimensional result. This usage is rare in Analytica since array abstraction normally does this automatically; however, the need occasionally arises in some circumstances.*

Suppose you have an array *A* indexed by *I*, and you wish to apply a function *f(x)* to each element of A along I. In a conventional programming language, this would require a loop over the elements of *A*; however, in almost all cases, Analytica's array abstraction does this automatically—the expression is simply: **f(A)**, the result remains indexed by *I*. However, there are a few cases where Analytica does not automatically array abstract, or it is possible to write a user-defined function that does not automatically array abstract (*e.g.*, by declaring a parameter to be of type *Scalar*, see page 422). For example, Analytica does not array

abstract over functions such as **`Sequence`**, **`Split`**, **`Subset`**, or **`Unique`**, since these return unindexed lists of varying lengths that are unknown until the function evaluates. Suppose we have the following Variables defined (note: *A* is an array of text values):

A: Index_1 ▼

1	A,B,C
2	D,E,F
3	G,H,I

Index_2:

1	2	3

We wish to split the text values in *A* and obtain a two dimensional array of letters indexed by **`Index_1`** and **`Index_2`**. Since **`Split`** does not array abstract, we must do each row separately and re-index by Index_2 before the result rows are recombined into a single array. This is accomplished by the following loop.

```
for Row:=Index_1 do
   Array(Index_2,Split(A[Index_1=Row],','))
```

resulting in

Index_1 ▼, *Index_2* ▶

	1	**2**	**3**
1	A	B	C
2	D	E	F
3	G	H	I

Reducing Memory Requirements: In some cases, it is possible to reduce the amount of memory required for intermediate results during the evaluation of expressions involving large arrays. For example, consider the following expression:

MatrixA: A two dimensional array indexed by *M* and *N*.
MatrixB: A two dimensional array indexed by *N* and *P*.

```
Average(MatrixA * MatrixB, N)
```

During the calculation, Analytica needs memory to compute *MatrixA* * *MatrixB*, an array indexed by *M*, *N*, and *P*. If these indexes have sizes 100, 200, and 300 respectively, then *MatrixA* * *MatrixB* contains 6,000,000 numbers, requiring over 60 megabytes of memory at 10 bytes per number.

To reduce the memory required, use the following expression instead

```
For L:=M Do Average(MatrixA[M=L]*MatrixB,N)
```

Each element `MatrixA[M=L]*MatrixB` has dimensions *N* and *P*, needing only 200x300x10= 600 kilobytes of memory at a time.

Analytica Note: *For the special case of a dot product (see "Dot product of two matrices" on page 272), where an expression has the form* `Sum(A*B,I)`*, Analytica performs a similar transformation internally.*

While (*Test*) Do *Body*

While evaluates *Body* repeatedly as long as *Test* <> 0. For **While** … to terminate, **Body** must produce a side-effect on a local Variable that is used by *Test*, causing *Test* eventually to equal 0. If *Test* never becomes `False` the **While** will continue to loop indefinitely. If you suspect that may be happening, type *Ctrl-.* (Control-period) to interrupt execution.

Test must evaluate to an atomic (non-array) value; therefore, it is a good idea to force any local Variable used in Test to be atomic valued. **While** is one of the few constructs in Analytica that does not generalize completely to handle arrays. But, there are ways to ensure that Variables and functions using **While** support Intelligent Arrays and probabilistic evaluation. See page 454 for details.

While returns the final value found in the last iteration of *Body* or `Null` if no iterations occur. For example:

```
(Var x := 1; While x < 10 Do x := x+1) → 10
(Var x := 1; While x > 10 Do x := x+1) → Null
```

Using **While** often follows the following pattern:

```
Var x[]:= ...;
While ( FunctionOf(x) ) Do (
    ...
    x := expr;
    ...
);
returnValue
```

Iterate(*x1, xi, bstop, maxIter, warn*)

Suppose the definition of Variable ***X*** contains a call to **Iterate**: **Iterate** initializes ***X*** to the value of ***x1***. While stopping condition ***bstop*** is **False** (zero), it evaluates expression ***xi***, and assigns the result to ***X***. Given the optional parameter ***maxIter***, it will stop after ***maxIter*** iterations and, if ***warn*** is **True**, issues a warning—unless it has already been stopped by ***bstop*** becoming **True.** If ***bstop*** is an array, it only stops when *all* elements of ***bstop*** are true.

Iterate is designed for convergence algorithms where an expression must be recomputed an unknown number of iterations. **Iterate** (like **Dynamic**) must be the main expression in a definition—it cannot be nested within another expression. But it may, and usually will, contain nested expressions as some of its parameters. **Iterate** (again like **Dynamic** and unlike other functions) may, and usually will, mention the Variable ***X*** that it defines within the expressions for ***x1*** and ***bstop***. These expressions may also refer to Variables that depend on ***X***.

If you use **Iterate** in more than one node in your model, you should be careful that the two functions don't interact adversely. In general, two nodes containing **Iterate** should never be mutual ancestors of each other. Doing so makes the nesting order ambiguous and can result in inconsistent computations. Likewise, care must be taken to avoid similar ambiguities when using interacting **Iterate** and **Dynamic** loops.

Analytica Note: *It is usually possible to write convergence algorithms more cleanly using* ***While.*** *One difference is that* ***While*** *requires its stopping condition* ***Test*** *to be a scalar, where* ***Iterate*** *allows an array-valued stopping condition* ***bstop****. Nevertheless, it is usually better to use* ***While*** *because you want it to do an appropriate number of iterations for each element of* ***bstop, rather than continue until all its elements are True.*** *But, with* ***While*** *you will need to use one of the tricks described on and after page 454 to ensure the expression fully supports array abstraction.*

Recursion

A recursive function is a function that calls itself within its definition. This is often a convenient way to define a function, and sometimes the only way. As an example, consider this definition of factorial:

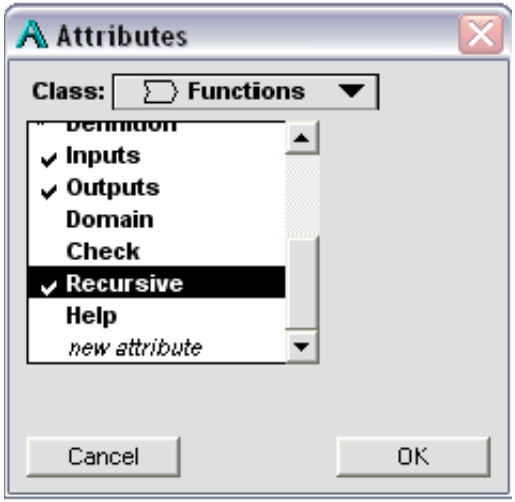
```
Function Factorial2( n : positive atomic )
Definition: IF n > 1 THEN N*Factorial2(n-1)ELSE 1
```

Thus, if its parameter, n, is greater than 1, Factorial2 calls itself if with the actual parameter (n-1). Otherwise, it simply returns 1. Like any normal recursive function, it has a termination condition under which the recursion stops.

Analytica Note: *Although the Factorial function is provided as a fully-abstractable built-in function in Analytica, we use it repeatedly as a simple example to demonstrate key ideas.*

By default, Analytica will not allow a definition of a function to refer to itself like this. It will complain about cyclic dependency. To enable recursion, you must first display the Recursive Attribute for Functions:

1. Select the **Attributes…** dialog from the **Object** menu.
2. Select **Functions** from the **Class** menu in the **Attributes** dialog.
3. Scroll down the list of attributes and click **Recursive** *twice*, to show √ which indicates that Functions will display the Recursive Attribute.
4. Check **OK** to close **Attributes** dialog.

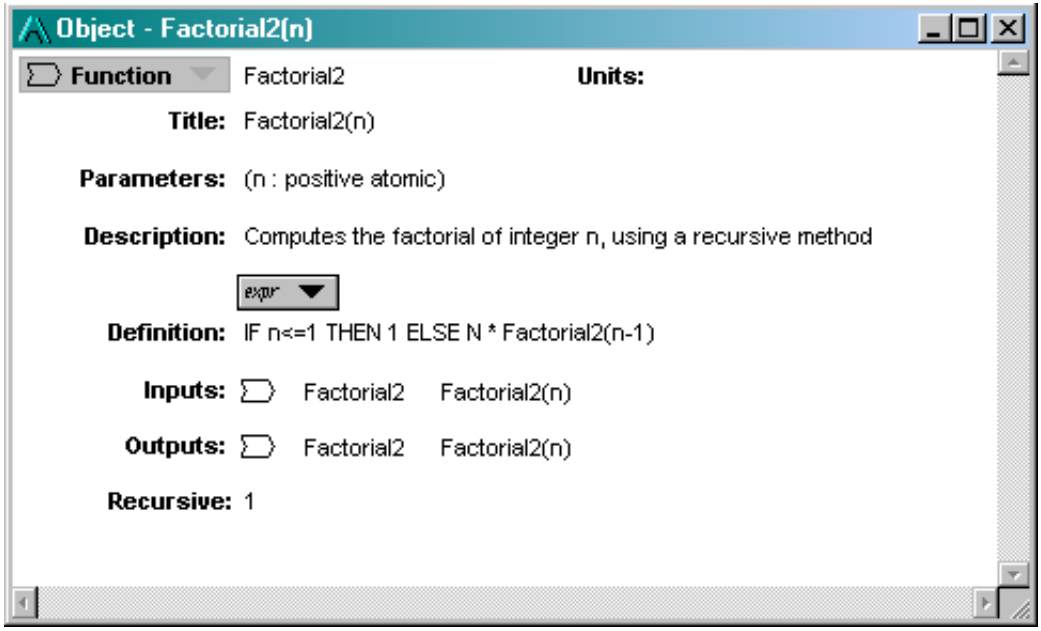


Then for each Function for which you wish to enable recursion:

5. Open the **Object Window** for the function by double-clicking on its node (or select the node and click the **Object** button)

6. Type **1** into its **Recursive** field.

The Object Window for the complete function should look like this:



As another example, consider this recursive function to compute a list of the prime factors of an integer, **x**, equal to or greater than **y**:

```
Function Factors(x, y : positive atomic)
Definition:
    Var n := Floor(x/y);
    IF n<y THEN [x]
    ELSE IF x = n*y THEN Concat([x], Factors(n, y))
    ELSE Factors(x, y+1)

Factors(60, 2) -> [2, 2, 3, 5]
```

Essentially, **Factors** says: compute **n** as **x** divided by **y**, rounded down. If **y** is greater than **n**, then **x** is the last factor, so return **x** as a list. If **x** is an exact factor of **y**, then concatenate **x** with any factors of **n**, equal or greater than **n**. Otherwise, try **y+1** as a factor.

Analytica Note: *Function calls are limited to a stack depth of 256 nested invocations*

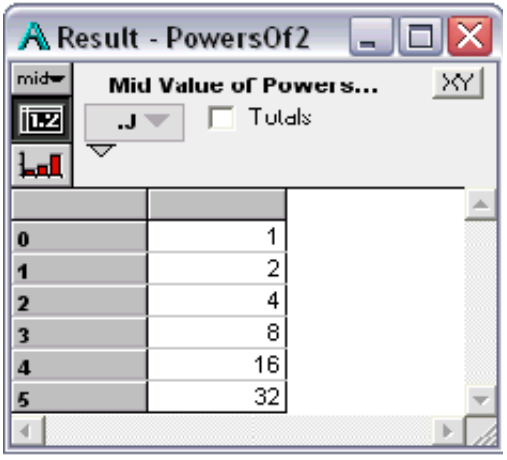
Local indexes

The construct, **Index *i* := *seqExpr*** defines an index local to the definition in which it is used. The expression ***seqExpr*** may be a sequence, literal list, or other expression that generates an unindexed array, as used to define a global index. For example:

```
Variable PowersOf2
Definition: Index J := 0..5; 2^J
```

The new Variable `PowersOf2` is an array of powers of two, indexed by the local index `J`, with values from `0` to `5`:

`PowersOf2` →



Dot operator: *a* . *i*

The dot operator in ***a . i*** lets you access a local index ***i*** via an array ***a*** that it dimensions. If a local index identifies a dimension of an array that becomes the value of a global Variable, it may persist long after evaluation of the expression—unlike other local Variables which disappear after the expression is evaluated.

Even though local index J has no global identifier, you can access it via its parent Variable with the dot operator, '**.**', for example:

`PowersOf2.J` → `[0,1,2,3,4,5]`

When using the subscript operation on a Variable with a local index, you need to include the '**.**' operator, but do not need to repeat the name of the Variable:

```
PowersOf2[.J=5] → 32
```

Any other Variables depending on `PowersOf2` may inherit `J` as a local index—for example:

```
Variable P2
Definition:  PowersOf2/2

P2[.J=5] → 16
```

Example using a local index

In this example, `MatSqr` is a user-defined function that returns the square of a matrix—*i.e.*, **A** x **A'**, where **A'** is the transpose of **A**. The result is a square matrix. Rather than require a third index as a parameter, `MatSqr` creates the local index, `I2`, as a copy of index `i`.

```
Function MatSqr(a: ArrayType; i,j: IndexType)
Definition:
    Index I2:=CopyIndex(i);
    Sum(a*a[i=I2], j)
```

The local Variable, `I2`, in `MatSqr` is not within lexical scope in the definition of `Z`, so we must use the dot operator '**.**' to access this dimension. We underline the dot operator for clarity:

```
Variable Z
Definition:
    Var XX := MatSqr(X, Rows, Cols);
    Sum(XX * Y[I=XX.I2], XX.I2)
```

Ensuring array abstraction

The vast majority of constructs in Analytica (operators, functions, and control constructs) fully support Intelligent Arrays—that is, they can handle operands or parameters with any number of indexes, and generate a result with the appropriate dimensions. So, most models automatically obtain the benefits of arrays abstraction described in the previous section with no special care.

There are, however, a few constructs that do *not* inherently enable Intelligent Arrays—or support *array abstraction*. They fall into these main types:

- Functions whose parameters must be scalar (not arrays), including **Sequence, m..n**, **SplitText**. See page 452.
- Functions whose parameter must be a vector (just one index), such as **CopyIndex**, **SortIndex**, **Subset**, **Unique**, and **Concat** (with two parameters).
- The **While** loop, which expects its termination condition to be a scalar. See page 454 .
- **If *b* Then *c* Else *d***, when condition ***b*** is an array, and ***c*** or ***d*** may give an evaluation error. See page 455.
- Functions with optional index parameters that are omitted, such as **Sum**(***x***), **Product**, **Max**, **Min**, **Average**, **Argmax, SubIndex**, **ChanceDist**, **CumDist**, and **ProbDist**. See page 456.

When using these constructs, you must take special care if you want to ensure that your model is fully array-abstractable. Analytica 3.0 and 3.1 provides a number of new features to make that much easier than before. Below we explain how to use them in each of these four cases.

Functions needing scalars and array abstraction

Consider this example:

```
Variable N
Definition: 1..3
Variable B
Definition: 1..N
B → Evaluation error:One or both parameters to
    Sequence(m, n) or m .. n are not scalars.
```

The expression `1..N`, or equivalently, `Sequence(1, N)`, will not evaluate if `N` is a vector. Otherwise, it would try to create a non-rectangular array containing lists with 1, 2, and 3 elements. Analytica does not permit such nonrectangular arrays, and so does not permit the parameters of **Sequence** to be arrays.

Similarly, a list cannot contain elements of different dimensions:

```
[1, 2, N] → Evaluation error:
```

In fact, most of the functions and expressions that can generate the definition of an index require scalar (or in some cases, vector) parameters, and so are not fully array abstractable. These include **Sequence**, **Subset**, **SplitText**, **SortIndex** (if the second parameter is omitted), **Concat**, **CopyIndex**, and **Unique**.

Why would you want array abstraction using such a function? Consider this approach to writing a function to compute a factorial:

```
Function Factorial1
Parameters: (n)
Definition: Product(1..n)
```

This function will work if `n` is scalar, but not if it is an array, because `1..n` requires scalar operands. However, this edition, with a For loop, will work fine:

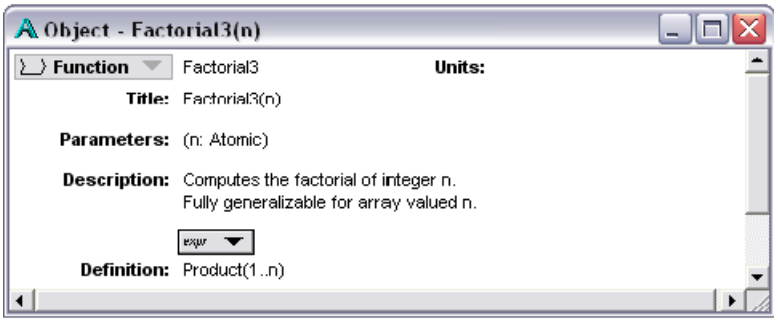
```
Function Factorial2
Parameters: (n)
Definition: For m[]:=n DO Product(1..m)
```

The For loop with empty square brackets `[]` next to the loop Variable, means that successive scalar elements of `n` are assigned to `m`, and the body `Product(1..m)` is evaluated for each one. Because `m` is guaranteed scalar, this works fine. The results of each value of each evaluation of `Product(1..m)` reassembled to create an array with the same dimensions as `n`.

Atomic parameters and array abstraction

This can allow you to assume a particular dimensionality within your function definition.The **Atomic** parameter qualifier is often the easiest approach to ensure array abstraction with a function that contains a construct that needs scalar parameters. By qualifying a parameter as **Atomic**, you are telling Analytica that the function definition requires that parameter to be scalar, for example:

```
Function Factorial3
Parameters: (n: Atomic)
Definition: Product(1..n)
```



The parameter qualifier **Atomic** does not require the actual parameter to be scalar. Rather it decomposes `n` into scalar elements, and applies the definition of `Factorial1(n)` to each one. Then, like the **For** loop, it reassembles the results, returning an array with the same indexes as `n`. It works fine even if several parameters are qualified as **Atomic**. You can also declare the dimensionality of array-valued parameters to ensure that only those dimensions appear while the body is being evaluated (see “Array qualifiers” on page 424). Doing so ensures your function will array abstract when new dimensions are added to your model, or when the function is evaluated in probabilistic mode.

While and array abstraction

The **While** ***b*** **Do** ***e*** construct requires its termination condition ***b*** to evaluate to a *scalar*—that is, a single **True** (1) or **False** (0). Otherwise, it would be ambiguous about whether it should continue. If you want a function definition using a **While** loop to array abstract, you can use the parameter qualifier **Atomic** as just described to manage Sequence and other functions that require their parameters to be scalars. In this example, the **Atomic** qualifier assures that `n` and hence the **While** termination condition **(a < n)** is a scalar during each evaluation of Factorial1:

```
Function Factorial1
Parameters: (n: Atomic)
Definition:
    VAR fact := 1;
    VAR a := 1;
    WHILE ( a < n )
    DO (a := a + 1; fact := fact * a )
```

If *b* Then *c* Else *d* and array abstraction

Consider this example:

```
Variable X =: -2..2
Sqrt(X) → [NAN, NAN, 0, 1, 1.414]
```

The square root of negative numbers -2 and -1 returns NAN (not a number) after issuing a warning. Now consider the definition of A:

```
Variable Y := (IF X>0 THEN Sqrt(X) ELSE 0)
Y → [0, 0, 0, 1 1.414]
```

When the condition of an IF construct is an array of truth values, as in this case, Analytica evaluates both THEN and ELSE parts, although it returns the expected elements of each part. So, it ends up evaluating `Sqrt(X)` even for negative values of `X`. The conditional expression defining `Y` will end up evaluating `Sqrt(X)` for every element of `X`, even those that are negative.

In this case, the final result contains no NANs, and so it generates no error message. In other words, there is no longer a problem. However, the warning will be delayed until after it has finished evaluating the expression.

A similar problem remains with functions that require particular types of parameter. Consider this array:

```
Variable Z := [1000, '10,000', '100,000']
```

This kind of array containing true numbers, e.g., 1000, and numbers with commas turned into text values, often arise when copying arrays of numbers from spreadsheets. The following function would seem helpful to remove the commas and convert the text values into numbers:

```
Function RemoveCommas(t)
Parameters: (t)
Definition: Evaluate(TextReplace(t, ',', ''))

RemoveCommas(Z) →
Evaluation Error: The parameter of Pluginfunction
TextReplace must be a text while evaluating
function RemoveCommas.
```

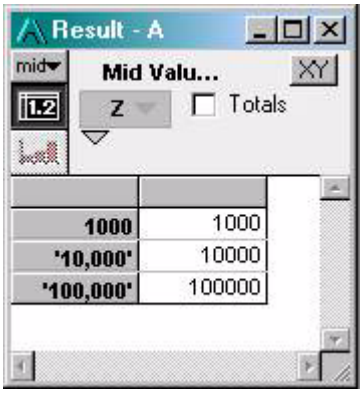
Unfortunately, `TextReplace` require its parameter to be a text value. What if we test if `t` is Text and only apply `TextReplace` when it is?

```
Function RemoveCommas(t)
```

```
Parameters: (t)
Definition: IF IsText(t)
   THEN Evaluate(TextReplace(t, ',', '')) ELSE t

RemoveCommas(Z) → (same error message)
```

This still doesn't work because the `IF` construct still applies `ReplaceText` to all elements of `t`. Now, let's add the parameter qualifier **Atomic** to **t**:

```
Function RemoveCommas(t)
Parameters: (t: Atomic)
Definition: IF IsText(t)
   THEN Evaluate(TextReplace(t, ',', '')) ELSE t
RemoveCommas(Z) →
```



This works fine because the **Atomic** qualifier means that `RemoveCommas` breaks its parameter t down into atomic (scalar) elements before evaluating the function. During each evaluation of `RemoveCommas`, `t` and hence `IsText(t)`, is scalar, either True or False. When False, the IF construct evaluates the ELSE part but not the THEN part, and so calls `TextReplace` when `t` is truly a text value. After calling `TextReplace` separately for each element, it reassembles the results into the array shown above with the same index as `Z`.

Omitted index parameters and array abstraction

A few functions have index parameters that are optional. For example, with **Sum**(***x***, ***i***), you can omit index ***i***, and call it as **Sum**(***x***). In that case, **Sum** will choose which index to sum over. But, if ***x*** has more than one index, it is hard to predict which index it will sum over. If you create the model when ***x*** has one index, its

behavior is predictable. But, if you then expand the dimensions of the model, so that ***x*** has additional indexes, its behavior is hard to predict—so, the model is not array abstractable.

There is a simple way to avoid this problem and maintain reliable array abstraction: *When using functions with optional index parameters,* ***never omit the index!*** Almost always, you know what you want to sum over, so mention it explicitly. If you add dimensions later, you'll be glad you did.

Other functions that have optional index parameters, and so to which this rule also applies, include, **Product**, **Max**, **Min**, **Average**, **Argmax**, **SubIndex**, **ChanceDist**, **CumDist**, and **ProbDist**.

Analytica Note: *When the optional index parameter is omitted, and the parameter has more than one dimension, these functions choose the* ***outer*** *index, by default. Usually, the outer index is the index created most recently when the model was built. But, this is often not obvious. We designed Intelligent Arrays specifically to shield you from having to worry about this detail of the internal representation.*

Selecting indexes for iterating with For and Var

To provide detailed control over array abstraction, the **For** loop can specify exactly which indexes to use in the iterator ***x***. The old edition of **For** still works. It requires that the expression a assigned to iterator ***x*** generate an index—that is, it must be a defined Index Variable, **Sequence**(***m***, ***n***), or ***m***..***n***. The new forms of **For** are more flexible. They work for any array (or even scalar) value ***a***. The loop iterates by assigning to ***x*** successive subarrays of ***a***, dimensioned by the indexes listed in square brackets. If the square brackets are empty, as in the second line of the table, the successive values of iterator ***x*** are scalar. In the other cases, the indexes mentioned specify the dimensions of ***x*** to be used in each evaluation of ***e.*** In all cases, the final result of executing the **For** loop is a value with the same dimensions as ***a***.

For *x* := *a* DO *e*	Assigns to local Variable ***x***, successive scalar values from index expression ***a*** and repeats evaluation expression ***e*** for each value. Returns an array of values of ***e*** indexed by ***a***.
For *x*[] := *a* DO *e*	Assigns to local Variable ***x***, successive scalar values from array ***a*** and repeats evaluation expression ***e*** for each value. Returns an array of values of ***e*** with the same indexes as ***a***.
For *x*[*i*] := *a* DO *e*	Same, but assigns to local Variable ***x***, successive subarrays indexed by ***i*** from array ***a*** and iterates evaluation of expression ***e*** for each index value of ***a*** other than ***i***.
For *x*[*i*, *j* …] := *a* DO *e*	Same, but each successive value assigned to ***x*** is indexed by the indices, [***i***, ***j*** …].

The same approach also works using **Var** to define local Variables. By putting square brackets listing indexes after the new Variable, you can specify the exact dimensions of the Variable. These indexes should be a subset (none, one, some, or all) of the indexes of the assigned value ***a***. Any subsequent expressions in the context are automatically repeated as each subarray is assigned to the local Variable. In this way, a local Variable can act as an implicit iterator, like the **For** loop.

```
Var Temp[I1,I2,...] := X;
```

References and data structures

A ***reference*** is an indirect link to a value, a scalar or an array. A Variable can contain a single reference to a value, or it can contain an array of references. Variables and arrays can themselves contain references, nested to any depth. This lets you create complex data structures, such as linked lists, trees, and non-rectangular structures. Use of references is provided by two operators:

\e is the ***reference operation***. It creates a reference to the value of expression ***e.***

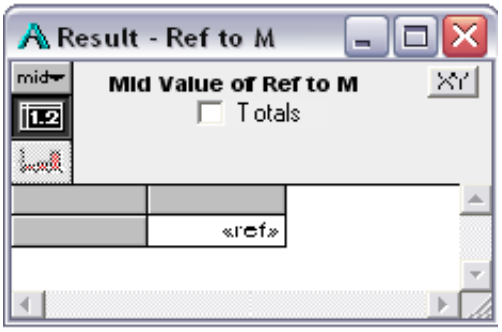
#e is the ***dereference operation***. It obtains the value

referred to by **e**. If **e** is not a reference, it issues a warning and returns **Null**.

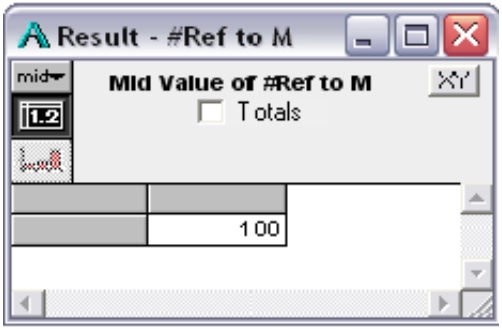
An example:

```
Variable M
Definition: 100

Variable Ref_to_M
Definition: \ M
```

The result of `Ref_to_M` looks like this:

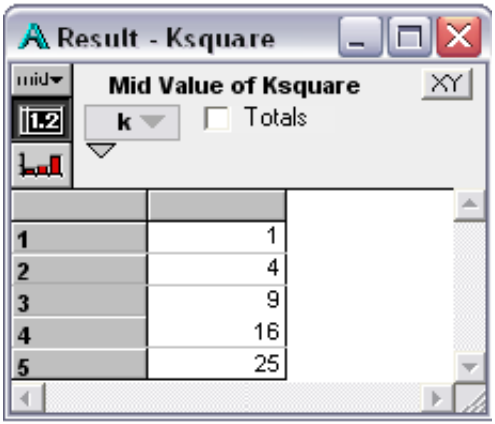


You can double-click on the cell containing «ref» to view the value referenced, in this case:

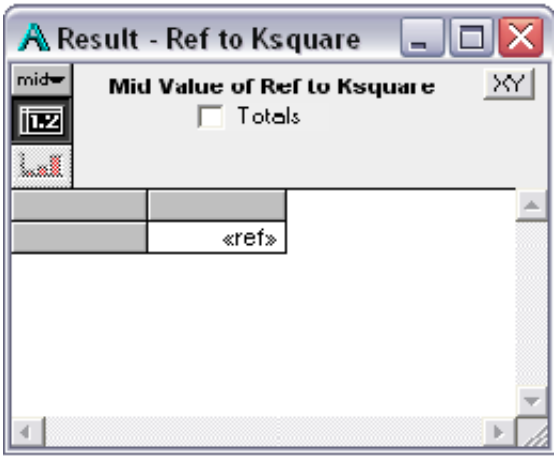


You can also create an array of references. Suppose

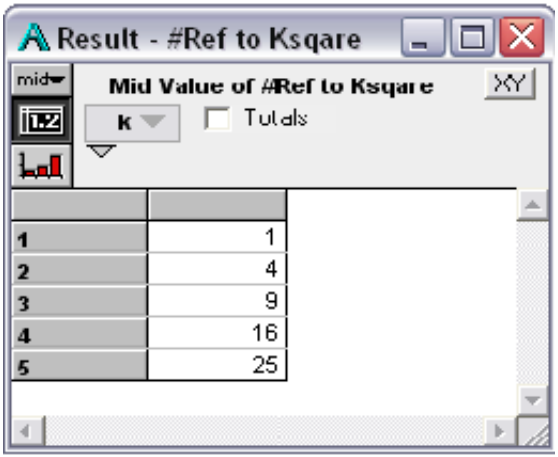
```
Index K
Definition: 1..5

Variable Ksquare
Definition: K^2

Ksquare →
```




```
Variable Ref_to_Ksquare
Definition: \ Ksquare

Ref_to_Ksquare →
```

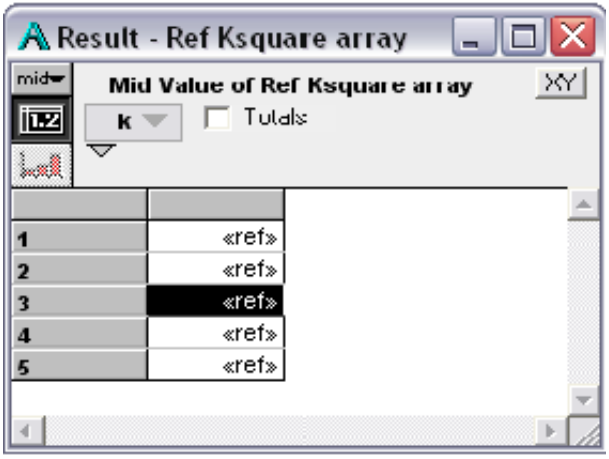


If you click on the «ref» cell, it opens:

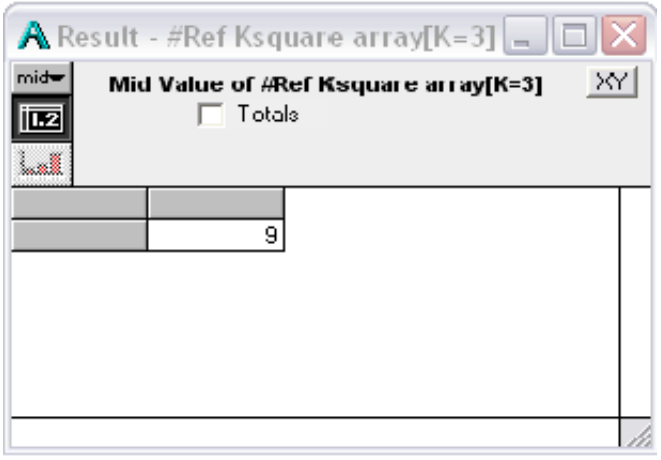


You can also create an array of references from an array, for example:

```
Variable Ref_Ksquare_array
Definition: \ [] Ksquare
Ksquare →
```



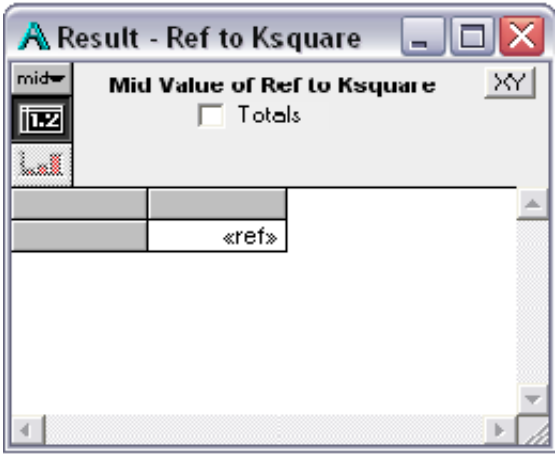
The empty square brackets '[ ]' specify that the values referred to have no indexes, *i.e.,* they are scalars. You can now click on any of these cells to see what it refers to. Clicking on the third cell, for example, gives:



Managing indexes of referenced subarrays: \ [*i, j,...*] *e*

More generally, you can list in the square brackets any indexes of ***e*** that you want to be indexes of each subarray referenced by the result. The other indexes of ***e*** (if any) will be used as indexes for the referencing array. Thus, in the example above, since there were no indexes in square brackets, the index **K** was used as an index of the reference array. If instead we write:

```
\ [K] Ksquare →
```



It creates a similar result to `\ Ksquare`, since **K** is the only index of `Ksquare`.

To summarize:

\ ***e***	Creates a reference to the value of expression ***e, whether it is a scalar or an array***.
\ **[]** ***e***	Creates an array indexed by all indexes of ***e*** ***containing references*** to all scalar values from ***e***.
\ **[*i*]** ***e***	Creates an array indexed by any indexes of ***e*** other than ***i of references*** to subarrays of ***e*** each indexed by ***i***.
\ **[*i*, *j* …]** ***e***	Creates an array indexed by any indexes of ***e*** *other than* ***i***, ***j … of references*** to subarrays of ***e*** each indexed by ***i***, ***j …***.

In general, it is better to include the square brackets after the reference operator, and avoid the unadorned reference operator, as in the first row of the table. Being explicit about which indexes to include will generally lead to expressions that array abstract as intended.

IsReference(*x*)

Is a test to see whether its parameter ***x*** is a reference. It returns **True** (1) if ***x*** is a reference, **False** (0) otherwise.

Using references for linked lists: Example functions

Linked lists are a common way for programmers to represent an ordered set of items. They are more efficient than arrays when you want often to add or remove items, thereby changing the length of the list (which is more time consuming for arrays). In Analytica, we can represent a linked list as a elements with two elements, the item—that is, a reference to the value of the item—and a link—that is, a reference, to the next item:

```
Index Linked_list
Definition: ['Item', 'Link']

Function LL_Put(x, LL)
Description: Puts item x onto linked list LL.
Definition: \Array(Linked_List,[\x,LL])

Function LL_Get_Item(LL)
Description: Gets the value of the first
   item from linked list LL.
```

```
Definition: # Subscript(#LL, Linked_list, 'Item')

Function LL_length(LL)
Parameters: (LL: Atomic)
Description: Returns the number of items in
    linked list LL
Definition: VAR len := 0;
    WHILE (IsReference(LL)) BEGIN
        LL := subscript(#LL, Linked_List, "Next");
        len := len + 1
    END;
    len

Function LL_from_array(a, i)
Parameters: (a; i: IndexType)
Description: Creates a linked list from the
    elements of array a over index i
Definition:
    VAR LL := NULL;
    Index iRev := Size(i) .. 1;
    FOR j := iRev
        DO LL := LL_Push(LL, Slice(a, i, j));
    LL
```

See **Linked List lib.ANA** for these and other functions for working with linked lists. .

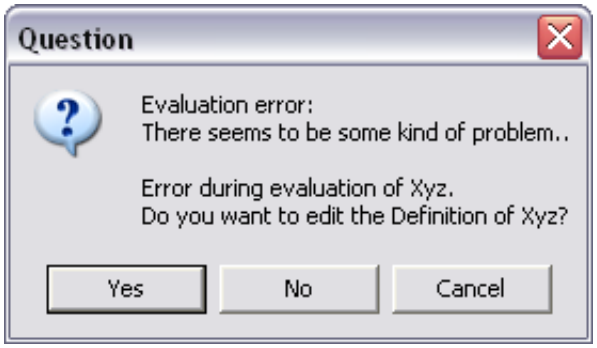
Miscellaneous functions

These functions include a variety of tools especially useful for advanced applications, including several (**Error**, **MsgBox**) useful for building interactive applications. For example, you can write wizards to automate certain modeling tasks, asking the user for options and input values.

Error(*message*)

Displays an evaluation error, with the specified ***message***, for example:

```
Variable Xyz :=
Error('There seems to be some kind of problem')
Xyz →
```



Evaluate(*t*)

Evaluate expects a text value ***t***, and evaluates it as though it were an expression in a definition. It returns the value resulting from evaluating the expression. For example:

```
Evaluate('10M /10') → 1M
```

If `t` contains any syntax errors, Evaluate will return Null; it will not flag a syntax error.

One use for Evaluate is to convert (coerce) a text representation of a number into the number itself, for example:

```
Evaluate('100M') → 100M
```

Like most other functions, it returns the deterministic (Mid) or probabilistic value, according to the context in which it is called.

Evaluate is powerful, and useful for a variety of purposes, but, it has some subtle aspects. Consider:

```
Variable A := 99
Variable B := (VAR A := 0; Evaluate('A + 1'))
B → 100
```

The Variable `A` in the evaluated text '`A + 1`' refers to the global `A`, not the local `A` defined in `B`. More generally:

- `Evaluate(t)` creates its own context for parsing t (at evaluation time), which is quite separate from the context of the expression in which the `Evaluate(t)` appears—*e.g.,* the definition of `B` above.
- So, text ***t*** cannot refer to local Variables, indexes, or function parameters defined in the context in which the Evaluate(***t***) function appears.

- If the text value of ***t*** refers to any global Variables—*e.g.*, `A` in the definition of `B` above—these will not appear as Inputs of `B`, nor will any changes to `A` cause automatic re-evaluation of `B`.
- Text t may itself define local Variables and indexes, and refer to them, but these will not be available outside ***t***.
- When `Evaluate` references another variable, Analytica will not be able to track the dependency. For example:

```
B := A+1
C := Evaluate('A+1')
```

  When `A` changes, Analytica will automatically ensure that `B` is updated, but it has no way of knowing C should also be recomputed.

Text ***t*** may itself be an expression that creates a text value to be evaluated by **Evaluate**. This text expression appears in the definition of V and is *not* subject to the above limitations, so, for example:

```
Variable V :=(Var x:= '10'; Evaluate(x & x))
V → 1010
```

FunctionOf(*e*)

FunctionOf() indicates that the definition in which it appears cannot be evaluated. Suppose a variable has a valid definition, such as

```
Variable X := A + B
```

If you then delete one of its inputs, say *A*, Analytica will substitute the following in the Definition of *X*:

```
FunctionOf( expr + B )
```

It retains the part of the definition that is still valid, in case you want to fix it.

IgnoreWarnings(*expr*)

Some warnings can be suppressed by embedding an expression inside an `IgnoreWarnings( expr )` function. `IgnoreWarnings( expr )` will evaluate *expr* and return the result. However, if a warning occurs while evaluating *expr*, the warning will not be displayed to the user, even if the `Show Result Warnings` preference is checked. (Not all warnings can be suppressed in this way).

Evaluates its parameter *expr*, and returns its value, while suppressing most warnings that might otherwise be displayed during the evaluation. It is useful when you want to evaluate an expression that generates warnings, such as divide by zero, that you know are not important in that context, but you do not want to uncheck the option **`Show Result Warnings`**, because you do want to see warnings that may appear in other parts of the model. For more on that option, see “Preferences dialog box” on page 88. For more on warnings, see “Warning” on page 527.

MsgBox(*message, buttons, title*)

MsgBox displays a standard popup model dialog box with the user-supplied ***message***, ***buttons*** (see numerical codes below), and ***title***. Analytica pauses until the user presses a button on the message box. It returns a number, depending on which button the user presses (see below).

The optional ***buttons*** parameter is a number that controls which buttons to display, as follows:

0 = OK only

1 = OK and Cancel (the default if buttons is omitted)

2 = Abort, Retry and Ignore

3 = Yes, No and Cancel

4 = Yes and No

5 = Retry and Cancel

To display an icon, add one of the following numbers to the **buttons** parameter:

16 = Critical (white X on red circle)

32 = Question

48 = Exclamation

64 = Information

MsgBox returns a number depending on which button the user presses:

1 = OK

2 = Cancel (stops any further evaluation)

3 = Abort

4 = Retry

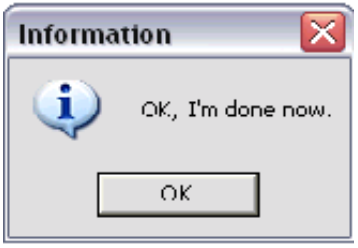
5 = Ignore

6 = Yes

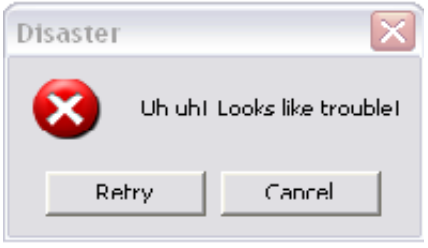
7 = No

Here are some examples:

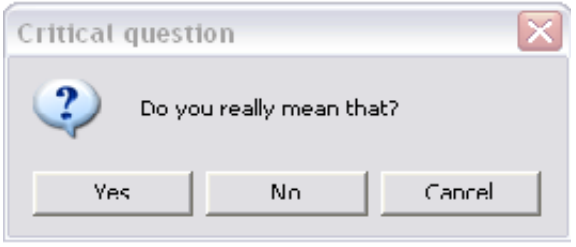
```
Msgbox('OK, I''m done now.',0+64,'Information') →
```



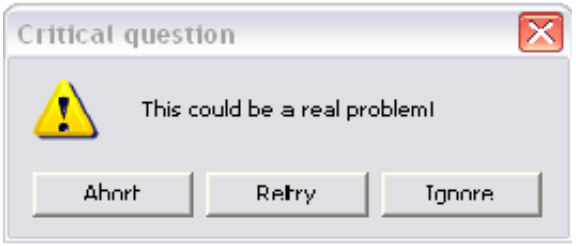

```
Msgbox('Uh uh! Looks like trouble!',5+16,
'Disaster' ) →
```




```
Msgbox('Do you really mean that?', 3+32, 'Critical
question') →
```




```
Msgbox('This could be a real problem!', 2+48,
'Critical question') →
```



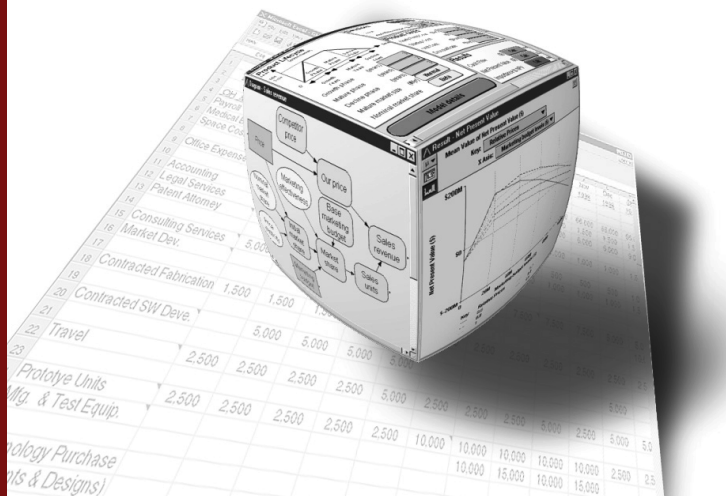
Today( )

The `today()` function returns the current date as the number of days that have elapsed since Jan 1, 1904.

Analytica Note: *When you evaluate this function, your result probably will be cached, and the cached result will eventually become out-of-date when the date changes.*

Chapter 21

Analytica Enterprise



In this Chapter

This chapter describes those features available only in the ***Enterprise*** edition of Analytica:

- Accessing external databases using ODBC and SQL.
- Creating browse-only models and other methods to protect or hide sensitive information when distributing your models to others.
- Huge Arrays, using indexes with more than 30,000 elements
- Profiling models to see CPU time and memory used by each Variable.

Note: *You must use Analytica Enterprise to create or set these features, including database access with ODBC and information hiding. However, you can run a model created in Analytica Enterprise with Analytica 2.0 or the Analytica Browser, and the model will be able to access databases and hide the information as specified in the original model.*

21: Analytica Enterprise

The Enterprise edition of Analytica extends the functionality of the Professional edition with features useful for developing and using models within a large enterprise or organization. Key areas of functionality offered by the Enterprise edition are:

- **Database access.** A collection of database functions make it possible to read and write data between external ODBC databases and your Analytica models.
- **Protecting Intellectual Property.** When you distribute your models to end-users, you can hide selected Variable definitions from the end-user, and/or prevent end-users from editing your model.

Accessing external databases

Analytica Enterprise provides several functions for querying external databases using ODBC. ODBC (**O**pen **D**ata**b**ase **C**onnectivity) is a widely used standard for connecting to relational databases, on either local or remote computers, and issuing queries in SQL (**S**tandard **Q**uery **L**anguage).

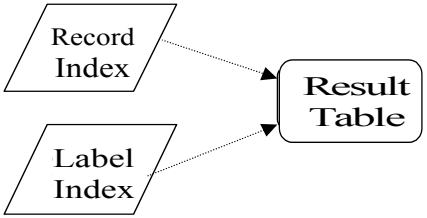
Analytica Note: *You can define and create database accesses only with Analytica Enterprise. However, you may use Analytica 2.0 or the Analytica Browser to run a model created with Analytica Enterprise, and to execute the database access features.*

Overview of ODBC in Analytica

The Standard Query Language (SQL) is a widely used language designed especially for the relational database model. In a relational database, data is organized in two-dimensional tables, where the columns of a table serve as fields or labels, and the rows correspond to records, entries, or instances. Common database terminology uses the terms ***columns*** and ***rows*** for these roles. In Analytica, it is more natural to refer to these as ***labels*** and ***records***. For instance, an address book table might have the columns or labels: LastName, FirstName, Address, City, State, Zip, Phone, Fax, E-mail, and each individual would occupy one row or record in that table.

Regardless of the underlying organization of data in a data source, the result of an SQL query is always a two-dimensional table, called a Result Table. Here the rows are the records matching whatever criteria is specified by the query, and the columns are the fields that are requested.

Analytica Enterprise provides functions that take as a parameter an SQL query, formulated with standard SQL syntax as an Analytica text value. When evaluated, these functions return the result of the query as a two-dimensional table in Analytica. A two-dimensional table in Analytica requires two indexes: the rows are indexed by a record index, and the columns are indexed by a label index. So, the basic structure of an Analytica model for retrieving a result table is:



(The dotted lines explicitly show what the Result Table is indexed by). Each of the three nodes in the above figure could require the information from the Result Table. For example, the definition of the Record Index would require knowing how many records (rows) are in the result table; the Label Index may need to read the names of the columns (although, often these will be known in advance); and of course, the Result Table needs to read the table. Thus, special functions in the Database library are used to define each of the above three Variables. These functions work in concert to perform the query only once (when the Record Index is evaluated), and share the result table between the nodes.

For the address database example above, we can obtain the record index as *Individuals*, the label index as *Address_fields*, and the resulting table as *Address_table*, as follows:

```
Individuals:=DBQuery(Data_source,'SELECT*FROM
Addresses')
Address_fields:=DBLabels(Individuals)
Address_table:=DBTable(Individuals,
Address_fields)
```

In the above example, the Record Index is defined using **`DBQuery()`**, the Label Index is defined using **`DBLabels()`**, and the

Result Table is defined using **DBTable()**. Each function is described below.

To specify a data source query, two basic pieces of information must always be known: The data source identifier, and the SQL query text. These two items are the arguments to the **DBQuery()** function, and are discussed in the following two subsections

Identifying the data source

A data source is described by a text value, which may contain the Domain Service Name (DSN) of the data source, login names and passwords, etc. Here, we describe the essentials of how to identify and access a data source. These follow standard ODBC conventions. For more details, consult one of the many texts on ODBC.

Note: *You must have a DSN already configured on your machine. If not, consult with your Network Administrator. See "Configuring a DSN" below.*

The general format of a data source identification text is (the single quotes are Analytica's text delimiters):

```
'attr1=value1; attr2=value2; attr3=value3;'
```

For example, the following data source identifier specifies the database called 'Automobile Data', with a user login 'John' and a password of 'Lightning':

```
'DSN=Automobile Data;UID=John;PWD=Lightning'
```

If a database is not password protected, then a data source descriptor may be as simple as:

```
'DSN=Automobile Data'
```

If a default data source is configured on your machine (consult your database administrator), you may specify it as:

```
'DSN=DEFAULT'
```

Some systems may require one login and password for the server, and another login and password for the DBMS. In this case, both can be specified as:

```
'DSN=Automobile Data; UID=John;
PWD=Lightning; UIDDBMS=JQR; PWDDBMS=Thunder'
```

You can use the **DRIVER** Attribute to specify explicitly which driver to use, instead of letting it be determined automatically by the data source type. e.g.,

```
'DSN=Automobile Data;DRIVER=SQL Server'
```

Instead of embedding a long data source connection text inside the **DBQuery()** statement, you can define a Variable in Analytica whose value is the appropriate text value. The name of this Variable can then be provided as the argument to **DBQuery()**. Another alternative is to place the connection information in a file data source (a .DSN file). Such a file would consist of lines such as:

```
DRIVER = SQL Server
UID = John
PWD = Lightning
DSN = Automobile Data
```

Assuming this data is in a file named MyConnect.DSN, the connection text can be specified as:

```
'FILEDSN=MyConnect.DSN'
```

In some applications, you may wish to connect directly to a driver rather than a registered data source. Some drivers may allow this as a way to access a data file directly, even when it is not registered. Also, some drivers may provide this as a way of interrogating the driver itself. To perform such a connection, use the driver keyword. For example, if the Paradox driver accepts the directory of the data files as an argument, you may specify:

```
'DRIVER={Paradox Driver};DIRECTORY='D:\CARS'
```

The specific fields used here (UID, PWD, UIDDBMS, PWDDBMS, DIRECTORY, etc.) are interpreted by the ODBC driver, and therefore depend on the specific driver used. Any fields interpreted by your driver are allowed.

If you do not wish to embed the full DSN in the connection text, a series of dialogs will pop up when the **DBQuery()** function is evaluated. For example, you can leave the UID and PWD (user name and password) out of your model. When the model is evaluated, Analytica will prompt you to enter the required information. Explicitly placing information in your model eliminates the extra dialog. A blank connection text may even be used, in which case you will need to choose among the data sources available on your machine when the model is being evaluated. Although the user can form the DSN via the graphical interface at that point, the result is not automatically placed in the definitions of your Analyt-

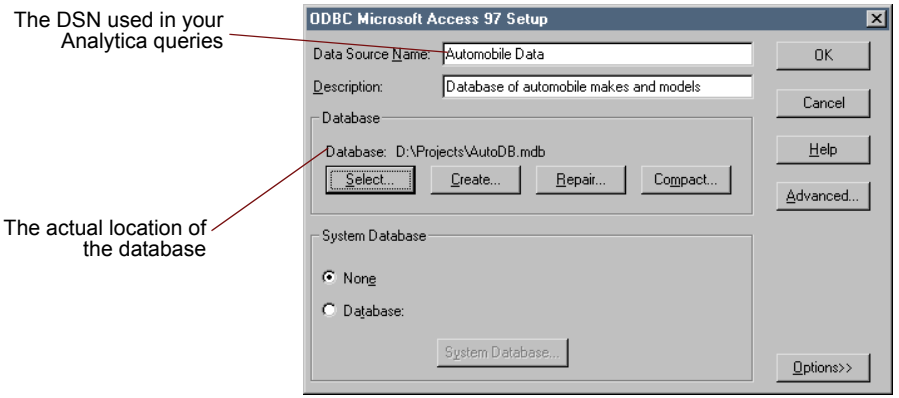
ica model. However, you may be able to store the information in a DSN file (depending on which drivers and driver manager you are using). You may also be able to register data sources on your machine from that interface.

Configuring a DSN

To access a database using ODBC, you must have a Data Source Name (DSN) already configured on your machine. In general, configuring a DSN requires substantial database administration expertise as well as the appropriate access permissions on your computer and network. To configure a data source, you should consult with your Network Administrator and/or your database product documentation. The general task of configuring a DSN is beyond the scope of this manual.

If you find you must configure a DSN yourself, the process usually involves the following steps (assuming your database already exists):

1. Select the ODBC icon from the Windows Control Panel.
2. Select the User DSN, System DSN, or File DSN tab depending on your needs. Most likely, you will want System DSN. Click the **Add** button.
3. Select the driver. For example, if your database is a Microsoft Access database, select Microsoft Access Driver and click **Finish**.
4. You will be led through a series of dialogs specific to the driver you selected. These will include dialogs that will allow you to specify the location of your database, as well as the DSN name that you will use from your Analytica model. An example is shown here:



Specifying an SQL query

You may use any SQL query as a text parameter within an Analytica database function. SQL queries can be very powerful, and may include multiple tables, joins, splits, filters, sorting, and so on. We give only a few simple examples here. The user interested in more demanding applications should consult a text on SQL, of which there are many available.

The SQL expression to select a complete table in a relational database, where the table is named **VEHICLES**, would be:

```
'SELECT * FROM vehicles'
```

Note: *SQL is case insensitive, unlike Analytica.*

To select only two columns (make and model) from this same table and sort them by make:

```
'SELECT make,model FROM vehicles ORDER BY make'
```

These examples provide a starting point. When using multiple tables, one detail to be aware of is that it is possible in SQL to construct a result table with two columns containing the same label. For example:

```
'SELECT * FROM vehicles,companies'
```

where both tables for vehicles and companies contain a column labeled 'Id'. In this case, you will only be able to access one (the first) of the two columns using **DBTable()**. Thus, you should take care to ensure that duplicate column labels do not result. This can be accomplished, for example, using the **AS** keyword, e.g.,

```
'SELECT vehicles.Id AS vid,companies.Id AS
```

```
cid,* FROM vehicles,companies'
```

For users that are unaccustomed to writing SQL statements, products exist that allow SQL statements to be constructed from a simple graphical user interface. Many databases allow queries to be defined and stored in the database. For example, from Microsoft Access, one can define a query by running Access and using the Query Wizard graphical user interface. The query is given a name and stored in the database. The name of the query can then be used where the name of a table would normally appear, e.g.,

```
'SELECT * FROM myQuery'
```

Retrieving an SQL result table

To retrieve a result table from a data source, you need:

1. The data source connection text,
2. The SQL query. These are discussed in the previous two sections. For illustrative purposes, suppose the connection text is `'DSN=Automobile Data'`, and the SQL statement is `'SELECT * FROM vehicles'`. In Analytica, you perform the following steps:

1. Create an index node. Name it RecordIndex, and specify its definition as:

```
DBQuery( 'DSN=Automobile Data',
    'SELECT * FROM vehicles' )
```

2. Create a second index node, and name it Labels. Specify its definition as:

```
DBLabels( Record_index )
```

3. Create a Variable node, name it Result Table, and specify its definition as:

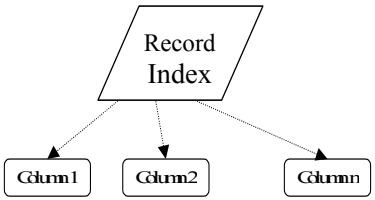
```
DBTable( Record_index,Labels)
```

You can now display Result Table to examine the results.

This basic procedure can be repeated for any result table. The structure of the model stays the same, and just the connection text and SQL query text change.

Separating columns in a model

Instead of retrieving relational data as a single two-dimensional table, as described above, it is often more convenient for further modeling in Analytica to break each column out into a separate Variable. For this to be useful, each column Variable shares the Record Index as a common index. An alternative structure is therefore:



With this model structure, the Record Index is again defined using **`DBQuery()`**, and each column is defined using **`DBTable()`**. The actual SQL query is issued only once when the Record Index is evaluated.

Suppose you wished to have *Make, Model, Year, MPG*, etc., as separate Analytica Variables, each a one-dimensional array with a common index. In this case, the steps are as follows:

1. Create an index node. Name it Record Index, and specify its definition as:

   ```
   DBQuery( 'DSN=Automobile Data',
       'SELECT * FROM vehicles' )
   ```

2. Create a Variable node named Make, and give it the definition:

   ```
   DBTable( Record_index, 'make' )
   ```

3. Create a Variable node named Year, and give it the definition:

   ```
   DBTable( Record_index, 'year' )
   ```

4. Create a Variable called CarModel, and give it the definition:

   ```
   DBTable( Record_index, 'model' )
   ```

In this case, Model is a reserved word in Analytica, so the Variable cannot be named *Model*. But, the second argument to **`DBTable()`** specifies the name of the column as stored in the database. This does not have to be the same as the name of the Variable in Analytica.

The intelligent array feature can be used in Analytica to construct a table containing only a subset of the columns in a Result Table.

For example, if vehicles has a large number of columns, a node defined by:

```
DBTable(Record_index,['make','model','year'])
```

will have only three columns. This table will be indexed by Record_index and by an implicit index (a.k.a. a null index or a self-index). The first argument to **DBTable()** must always be an indexed defined by **DBQuery()**—remember the SQL query is defined in that node, and this is how **DBTable()** knows which table is being retrieved.

DBWrite: writing to a database

You can use SQL to change the contents of the external data source from within Analytica, or from within an Analytica model. Using the appropriate SQL statements, records can be added to or deleted from an existing table (in the data source), columns may be added (if your data source driver supports this), and tables may be created or deleted.

You can not use the **DBQuery()** function to alter the data source, since it processes the SQL statement in read-only mode. To change the data source, use the **DBWrite()** function. **DBWrite()** is identical to **DBQuery()** except that the SQL statement is processed in read-write mode.

You can issue any alteration that can be expressed as an SQL statement, and that is supported by the ODBC driver you are using, from Analytica using **DBWrite()**. However, to get data from your model into the database, you must convert that data into a text value—more precisely, into an SQL statement. Doing so can be tricky, but some tools have been provided to make the process a little easier. In this section, we will show how one of the more common cases can be handled—the task of writing the data in a multi-dimensional array to a table in a database. To do this, we will make use of the ODBC_Library.ana library, which is collection of user-defined functions written in Analytica and included in the example models distributed with Analytica.

The example scenario is this: An Analytica model computes a Variable *A*, whose result is an array indexed by *I*, *J*, and *K*. We want this array to be written to a table in our database named *TableA*. Other applications can then make use of this data.

There are basically two complications with this example. The first is the fact that our array is three-dimensional, while a database table is always two-dimensional. The second complication is that,

even if you have a two-dimensional table in Analytica, how do you construct the SQL text to write the table to the database?

Our approach is to first convert the three-dimensional array *A* into a two dimensional table, which we will store in an Analytica Variable named *TableA*. *TableA* will need two indexes: *ARowIndex* and *ALabelIndex*. These three Variables are defined as follows:

```
ALabelIndex := concat(IndexNames(A),['A'])
ARowIndex := sequence(1,size(A))
TableA:= MDArrayToTable(A,ARowIndex,ALabelIndex)
```

`MDArrayToTable()` is described on page 260. *ALabelIndex* evaluates to *['I','J','K','A']*, and *ARowIndex* sets aside one row for each element of *A*. *TableA* is then a table with one row for each element of *A*, where the value of each index for that element is listed in the corresponding column, and the value of that element appears in the final column.

Next, set up *TableA* in the database with the same columns. This is most easily done using the front end provided with your database. For example, if you are using MS Access, start the MS Access program, and from there, create a new table. Alternatively, you could issue the statement

```
DBWrite(DB,'CREATE TABLE TableA(I TEXT,J TEXT,K
TEXT,A TEXT)')
```

from an Analytica expression (replacing TEXT with whatever type is appropriate for your application). Be sure that the column labels in the database table have the same names as the labels of *ALabelIndex* in the Analytica model.

Note: *It is possible (and easy) to use column labels in the database that are different from the index names in the Analytica model. To do this, define ALabelIndex to be 1-D array. The domain of the array should be the database labels, and the values of the array should be the index names (with the final value being arbitrary).*

With our data now in a 2-D table form, as needed for a database table, we must now construct the SQL text to write the table to the database. Before doing this, a few choices must be made. Should the write operation append rows to the existing database table? Or should it entirely replace the table? Or perhaps, it should replace only selected entries. The various choices made impact how the SQL statement is to be constructed. Here we will totally replace any existing data with the new data, so that after the

operation completes, the table in the database will be exactly the same as *TableA* in the Analytica model. The SQL statements for performing the write this looks like:

```
DELETE * FROM TableA
INSERT INTO TableA(I,J,K,A) VALUES
('i1','j1','k1','a111')
INSERT INTO TableA(I,J,K,A) VALUES
('i1','j1','k2','a112')
...
```

The first statement removes existing data, since we are replacing whatever is there. Then, there will be one INSERT INTO statement for each row of *TableA*. The data to the right of the VALUES keyword is replaced by the specific values for indexes *I*, *J*, *K*, and array *A* (the example above assumes the values are all text values). If your values are numeric, you should note that MSAccess is happy quoting numeric values when the column is numeric.

Since writing the table requires a series of SQL statements, we have two options: Evaluate a series of `DBWrite()` functions, or lump the series of SQL statements into one long text value and issue one `DBWrite()` statement. In Analytica, the second option is much more efficient for two reasons. First, the overhead of connecting with the database occurs only one time. Second, intermediate result tables do not have to be read from the ODBC driver, while if you issued separate `DBWrite()` statements, each one would go through the effort of acquiring the result table, only to be ignored.

Important feature (double semi-colon)

To allow multiple SQL statements in a single `DBWrite()` function (or in a single `DBQuery()` function), Analytica provides an extension to the SQL language. The double semi-colon separates multiple statements. For example, the expression

```
'DELETE * FROM TableA ;; SELECT * FROM TableA'
```

first deletes the data from the table, and then reads the (now empty) table. When ;; is used, only the last SQL statement in the series returns a result table. Most statements that write to a database return an empty result table.

We are now ready to write the Analytica expression that will construct the SQL statement to write the table to the database. The function to do this already exists in the ODBC_Library. First, use the **Add Module** item on the **File** menu to insert the ODBC_Library into your model; then use the `WriteTableSql()`

function, which returns the SQL statement (as a text value) for writing the table to the database. The function requires that *I* and *L* contain no duplicates (which should be the case anyway).

To perform the write, set the definition of a node (named *WriteIt*) to:

```
WriteIt := DBWrite(DB, WriteTableSql(A, RowIndex,
LabelIndex,'TableA'))
```

Any time *WriteIt* is evaluated, Analytica writes the table to the database.

Creating a button to initiate the write operation

The one problem with this setup is that the data doesn't get written until *WriteIt* gets evaluated. Because Analytica is entirely demand driven, *WriteIt* will not be automatically evaluated when *A* changes. A better solution is to make *WriteIt* a button (output node) on the diagram. Then a user can press the button to initiate the write-to-database operation. To create the button, select the *WriteIt* node and select the **Make Output Node** command on the **Edit** menu.

In most cases, it is the side effect, and not the result, of evaluating `DBWrite()` that is of interest. Therefore, there is no reason to force a user to view an empty result window when the *WriteIt* button is pressed. This can be avoided by having *WriteIt* evaluate to a text value, which will then show in place of the button when the most recently computed result has been written to the database. To accomplish this, the definition for *WriteIt* can be changed to

```
WriteIt :=
using dummy:=
    DBWrite(DB, WriteTableSql(A, RowIndex,
              LabelIndex,'TableA'))
    do
    'Done'
```

Database functions

The **Database** Library on the **Definition** menu contains five functions for working with ODBC databases.

Note: *These functions are available only in Analytica Enterprise.*

DBLabels(*dbIndex*)

Returns a list of the column labels for the result table. This statement may be used to define an index which can then be used as the second argument to `DBTable()`. The first argument, dbIndex, must be defined by a `DBQuery()` statement.

DBQuery(*connectionString, sql*)

Used to define an index Variable. The definition of the index should contain only one `DBQuery()` statement. *ConnectionString* specifies a data source (*e.g.,* `'DSN=MyDatabase'`). *SQL* defines an SQL query.

When placed as the definition of an index Variable, `DBQuery()` will be evaluated as soon as the definition is complete. When it is evaluated, the actual query is performed. The resulting result table is cached inside Analytica, to subsequently be accessed by `DBTable()` or `DBLabels()`.

`DBQuery()` returns a sequence *1..n*, where *n* is the number of records (rows) in the result table.

`DBQuery()` should appear only once in a definition, and if it is embedded in an expression, the expression must return a list with n elements.

`DBQuery()` processes the sql statement in read-only mode, so that the data source cannot be altered as a result of executing this statement. To alter the data source, use `DBWrite()`.

DBTable(*dbIndex, column*)
DBTable(*dbIndex, columnList*)
DBTable(*dbIndex, columnIndex*)

`DBTable()` is used to get at the data within a result table. The first argument, *dbIndex*, must be the name of a Variable (normally an index) in your Analytica model that is defined with a `DBQuery()` statement. If the second argument, *column*, is a text value, it identifies the name of a column label in the result table, in which case `DBTable()` returns a 1-D array (indexed by *dbIndex*) with the data for that column. If the second argument is a list of text values (the *columnList* form), then `DBTable()` returns a 2-D table with records indexed by *dbIndex*, and columns implicitly indexed (*i.e.*, self-indexed/null-indexed). If the second argument is the

name of an Analytica Variable (usually an index) whose value evaluates to a list of text values, those text values become the column headings for a 2-D table with columns indexed by *columnIndex*, and rows indexed by *dbIndex*. With this last form, *columnIndex* may be defined as `DBLabels(dbIndex)`.

DbTableNames(*connectionString, cat, sch, tab, typ*)

Connects to an ODBC data source and returns catalog data for the data source. *ConnectionString* specifies a data source (e.g., `'DSN=MyDatabase'`). *Cat* (catalog names), *sch* (schema names), *tab* (table names), and *typ* (table types) may be patterns if your ODBC driver manager is ODBC 3 compliant. Use '%' as a wildcard in each field to match zero or more characters. Underscore, '_', matches one character. Most drivers use backslash ('\') as an escape character, so that the characters '%', '_', or '\' as literals must be entered as '\%', '_', or '\\'. *Typ* may be a comma-delimited list of table types. Your data source and ODBC driver may or may not support this call to varying degrees.

Examples

To get all valid catalog names in *My db*:

```
DBTableNames('DSN=My db',’%’,’’,’’,’’)
```

To get all valid schemas in *My db*:

```
DBTableNames('DSN=My db',’’,’%’,’’,’’)
```

To get all valid table names in *My db*:

```
DbTableNames('DSN=My db',’’,’’,’%’,’’)
```

To get all valid table types:

```
DbTableNames('DSN=My db',’’,’’,’’,’%’)
```

DBWrite(*connectionString, sql*)

This function is identical to `DBQuery()` except that the query is processed in read-write mode, making it possible to store data in the data source from within Analytica.

SqlDriverInfo(*driverName*)

Returns a list of Attribute-value pairs for the specified driver. If *driverName*`=''` (an empty text value), returns a list of the names of the drivers. `driverName` must be a text value—it cannot be a list of text values or an index that is defined as a list of text values. This statement would not normally be used in a model, but

may be helpful in understanding the SQL drivers that are available.

Protecting intellectual property

Note: *These features are available only from Analytica Enterprise.*

When you are ready to let others use the models you have created, you may want to protect your hard work either by preventing the end-user from changing your model, or by limiting the end-user's access to selected definitions. Analytica Enterprise provides features for this purpose.

Using these features to protect intellectual property involves these steps:

1. Hide selected definitions.
2. Backup your master model file (and any linked submodules) to a safe place.
3. Save a protected (obfuscated) copy of your model.

   "Obfuscated" refers to the fact that the model file is scrambled into a non-human-readable form.
4. Distribute the "obfuscated" copy to your end-users.

The third step permanently locks your model so that hidden definitions can never again be viewed in that copy. It is therefore recommended that you save a protected *copy* of your model, and leave your original model as a master (unprotected) copy. Until the model is stored in an "obfuscated" form (step 3), an end-user is not prevented from unhiding your definitions, or from viewing them by other means (e.g., by loading the Analytica model file into a text editor).

Warning: *An obfuscated model file cannot be un-obfuscated, even by the original author. If it is locked as Browse-only, it can never again be edited. If definitions are hidden, they can never again be viewed or edited.* ***Always place a master copy of your model (and any submodules) in a safe place before making an obfuscated copy!***

Hiding definitions

By hiding your definitions, you are essentially hiding the algorithms and data in your Analytica model from the eyes of your end-user.

When a definition is hidden, the definition Attribute displays as:

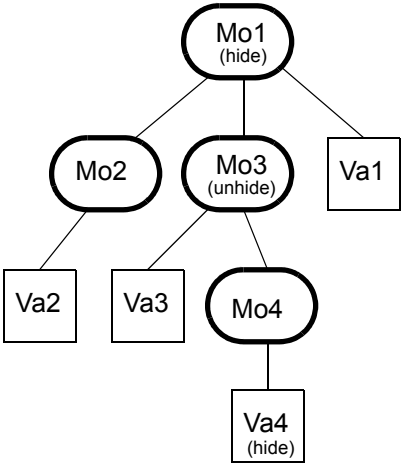
```
[Definition is Hidden]
```

However, the Variable can still be evaluated and its result viewed.

Note: *Variables associated with input nodes are always visible.*

Inheritance of definition hiding

Whether a definition is hidden or not is determined through an inheritance over the module hierarchy. Using this inheritance, you can easily hide individual definitions or all definitions within a given model or module (and its submodules). You can also exclude selected nodes or modules from being hidden. For example, you can specify that the definitions for all Variables in module *A* or any of its submodules are to be hidden, except for those in submodule *B*.



As an example, consider the module hierarchy shown above. A flag to hide definitions is set for module *Mo1* and Variable *Va4*. A flag to unhide definitions is set for module *Mo3*. With these settings, all Variables under *Mo1*, except those under *Mo3*, have hidden definitions. Thus, the definition of *Va3* is visible, while the definitions for *Va1, Va2,* and *Va4* are all hidden.

Hiding and unhiding definitions

To hide all definitions within a model or a module, bring the module window to the front and click in the diagram background so that no nodes are selected. To hide a single definition, select the Variable's node (only a single node should be selected when toggling the hide/unhide status). Then examine the **Object** menu.

Checkmarks next to **Hide Definition(s)** and **Unhide Definition(s)** indicate the cloaking status for the current module or selected node. If no checkmarks appear, the module or node inherits its cloaking status from its parent module. If a checkmark appears next to **Hide Definition(s)**, then the definition for this Variable, or for all Variables within the selected module, are hidden. If a checkmark appears next to **Unhide Definition(s)**, then the cloaking status of the current node's parent is overridden so that definitions under the current node are to be visible. To toggle the cloaking status, simple select **Hide Definition(s)** or **Unhide Definition(s)** as appropriate.

Remember that the definitions of Variables with associated input nodes are always visible regardless of cloaking status.

After you have selected the desired cloaking status, you can browse your model to verify that your definitions are hidden or visible as desired. However, until you lock these setting in place by saving your file in an obfuscated format, your end-user may still be able to view your definitions, for example, by toggling the cloaking status, or by viewing your model file in a text editor.

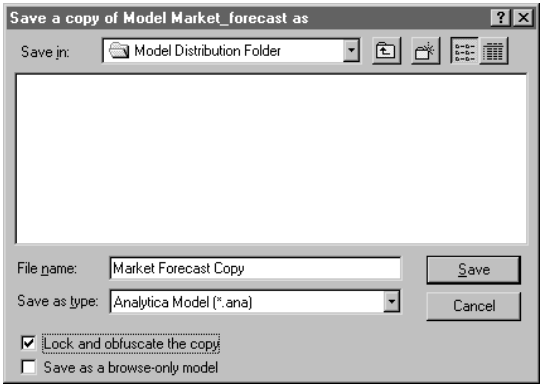
The **Hide Definition(s)** and **Unhide Definition(s)** menu options are disabled in the following circumstances:

- If the current model (or any of its linked submodules) has been loaded from an obfuscated model file.
  In this case, you cannot toggle the cloaking status since obfuscation has locked it in place.
- If more than one node is selected.

Saving an obfuscated copy of your model

There are two main reasons you might wish to create an obfuscated model file. The first is to lock the cloaking status of definitions so that an end-user cannot unhide or otherwise gain access to your proprietary definitions (see the previous section). The second reason is to permanently lock your model into a browse-only mode so end-users cannot edit the model (see the next section).

When you decide to save an obfuscated copy of your model, you should do so with extreme caution since obfuscation is irreversible. This point cannot be emphasized enough. Obfuscation protects your intellectual property from the eyes of others, but if you replace the master copy of your model with an obfuscated one, you will find obfuscation to be equally effective in protecting your intellectual property from your own eyes! *We recommend always placing a master copy of your model and all its linked submodules in a safe place before making an obfuscated copy*. In this case, it is better to be safe than sorry.



When you are ready to save an obfuscated copy of your model, select **Save a Copy In...** from the **File** menu. Enter a filename that is different than the filename of your master copy, and mark the **Lock and obfuscate the copy** checkbox at the bottom of the dialog and press the **Save** button.

To protect you from accidentally obfuscating your master copy, these checkboxes appear only on the **Save a Copy In...** dialog.

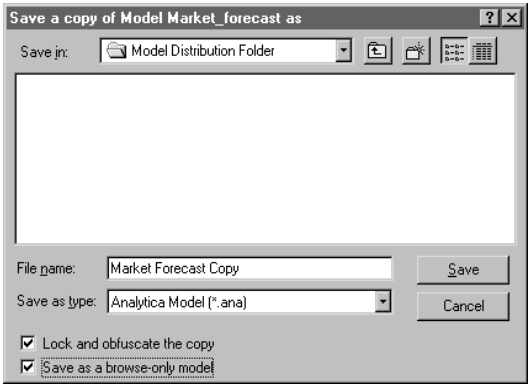
Saving a browse-only copy

You may wish to distribute a copy of your model to others, without allowing them to edit your model. This is accomplished by saving a browse-only copy. The end-users will be able to change Variables with associated input nodes, but will not be able to change other Variables in the model or add or delete Variables. The user will not be able to leave browse mode while the model is loaded.

If your end-user has only the Analytica Browser, which can be freely distributed, you do not need to create a browse-only copy. However, if your end-user obtains a registration number (either a licensed copy of Analytica or a trial registration number), they will

be able to make changes unless you have given them a browse-only copy.

If your model uses Enterprise database functionality, and you wish to share this model with others who have a different edition of Analytica, you *must* save the copy you give them as browse-only. Otherwise, the database functions cannot be evaluated by these users.



To save a browse-only copy, select **Save a Copy In...** from the **File** menu, enter a file name that is different from the name of your master copy, and check the **Save as a browse-only model** checkbox at the bottom of the dialog.

When a browse-only model (saved as such from Enterprise) is loaded into Analytica Professional, Analytica Lite, or Analytica Professional, it runs it in Power Player mode

Warning: *The browse-only flag cannot be reset and browse-only models are obfuscated.* ***Take extreme care to ensure that you have placed a master copy of your model in a safe place before saving a browse-only copy.*** *In general, you should take the same precautions emphasized in "Saving an obfuscated copy of your model" above.*

Obfuscation and linked submodules

If you plan on distributing an obfuscated version of your model, it is preferable to embed submodules rather than linking submodule files into your model. In so doing, you minimize the chances of accidentally obfuscating a submodule file, or of leaving a sub-

module file unobfuscated. This only a recommendation and not a requirement.

If you do have linked submodules, their files are not obfuscated when you save your main model using the **Save a Copy In...** command. To obfuscate these files, open the Object window for each submodule, and with the window in the foreground, select **Save a Copy In...**. Notice that the module name will appear in the title bar of the dialog. From there, the obfuscation option can be selected. Again, take extreme caution to avoid replacing your master copy. To ensure that the links between file name are maintained in your copy, you will need to use the same file name as the original linked file name, but you will want to store the copies in a new directory, keeping the relative paths the same.

When any obfuscated model file is loaded into Analytica, even if it is a sub-module, Analytica will treat the entire model as if it has been obfuscated. If any module file is browse-only, the entire-model will be browse-only. If any submodule is dirtied and saved at that point, it will be saved in an obfuscated form.

Warning: *Never import an obfuscated submodule into your master model. Doing so could cause your master model to become obfuscated next time it is saved.*

Huge arrays

Analytica Enterprise edition and ADE can manage Indexes and Arrays of up to 100 Million elements in any dimension. The only practical limit on model sizes is the amount of memory.Huge Arrays means they can also handle Sample Size for probabilistic simulation up to this size. (You can set this in the **Uncertainty Setup** dialog from **Result** menu.) Huge Arrays also let you read in large datasets from databases, using the ODBC functions.

Analytica Note: *All editions of Analytica other than Enterprise and ADE are limited to dimensions and sample sizes of up to 30,000 elements.*

Result tables and Edit tables will let you scroll over the first 30,000 elements (ending in '…'). Result graphs will plot the complete graph correctly.

Selecting a small window onto a Huge Array

To explore a Huge Array, it is often useful to create a small manageable window into it. In this example, X is a Huge Array indexed by J, with 100,000 elements. `Xview` selects a sample of 100 (`JViewSize`) elements from X starting at J = 50K (`JView-Start`):

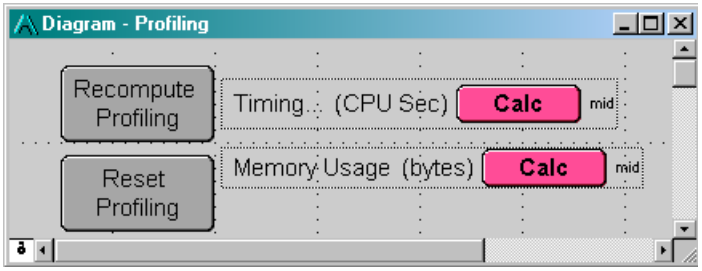
```
Index J := 1..100K
Variable X := (j-30K)^2
Variable JViewStart := 50K
Variable JViewSize := 100
Index JView:=JviewStart..(JviewStart+JviewSize)
Variable Xview := X[J = Jview]
```

To make it easy to select and display windows, you could create Input Nodes for **`JViewStart`** and **`JViewSize`**, and an output node to display the selected **`XView`**.

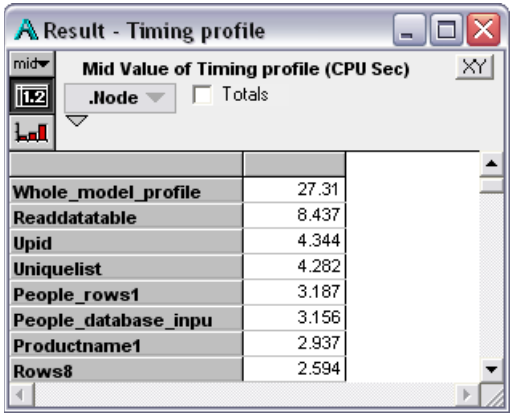
Time profiling

As experienced programmers know, it is often comes as a surprise to find out where a complex computer program spends most of its time. They use time profiling to find out which parts of a program are taking the lion's share of the CPU cycles.Once they have this information, they know where to focus their efforts to speed up the model. Analytica Enterprise 3.1 provides time profiling to help modelers optimize large Analytica models. These facilities trace how long it takes to evaluate each Variable in a model.

First add the **Profiling library** from the **Libraries** folder as a module of your model, selecting the **Embed as copy** option. After you have evaluated key nodes in your model, pen the Profiling library and click on the **Calculate** button for **Timing** output button to view how many CPU seconds have been spent on each node.



The profiler displays a list of all the model Variables whose values have been computed, with the time to compute each in seconds, in descending order.



Whole_model_profile	27.31
Readdatatable	8.437
Upid	4.344
Uniquelist	4.282
People_rows1	3.187
People_database_inpu	3.156
Productname1	2.937
Rows8	2.594

After you evaluate additional Variables of the model, click the button **Recompute timings** to update the timings. Click **Reset timings** if you want to reset them all to zero to examine a new set of computation times.

Once you have identified the largest contributors to evaluation time, you can concentrate your attention on those Variables if you want to speed up the model.

Time profiling makes use of two special read-ony attributes:

- **EvaluationTime** returns the time in seconds to evaluate this Variable, not including the time to evaluate any of its inputs (or their inputs, etc.).
- **EvaluationTimeAll** returns the time in seconds to evaluate this Variable, and any of its inputs that needed to be evaluated (and their inputs, and so on.).

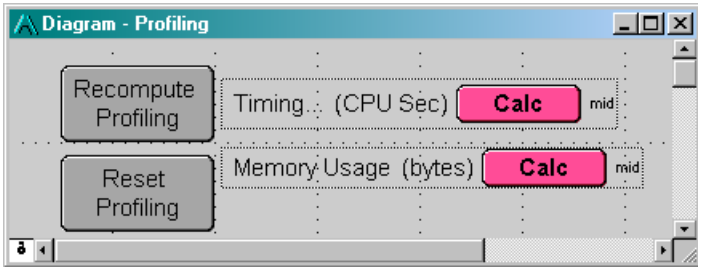
The command **ResetElapsedTimings** sets all these attributes back to zero.

Analytica Note: *Time profiling is only available for the Enterprise edition and ADE.*

Memory profiling

In addition to time profiling, Analytica Enterprise 3.1 and ADE 3.1 also support memory profiling. This allows you to determine which parts of your program are using the most memory and how much memory they use.

To profile memory use, add the **Profiling library** from the **Libraries** folder as a module of your model, selecting the **Embed as copy** option. After you have evaluated key nodes in your model, open the Profiling library and click on the **Calculate** button for **Timing** output button to view how many CPU seconds have been spent on each node.



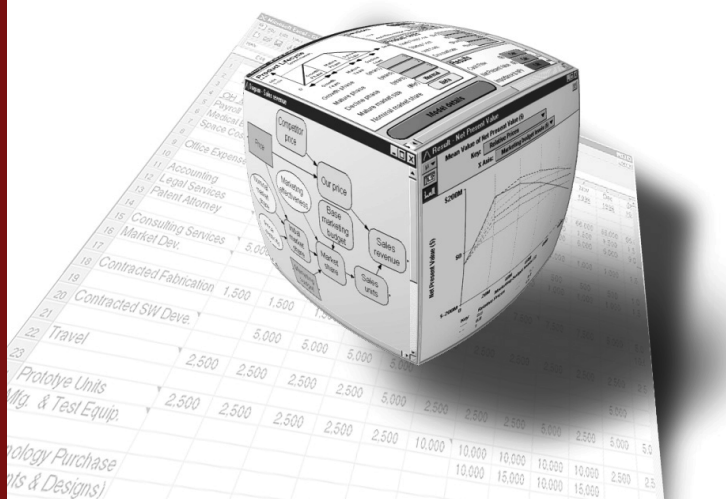
This will display a list of memory users, sorted in descending order of memory use.

Analytica Note: *The memory use shown is approximate, generally showing more memory used than is actually used, because it does not account for shared blocks of memory. That is, some blocks of memory may be shared, either within a single result, or between two different results. These shared blocks, when they exist, will get counted twice by the memory profiler.*

There is also a special memory usage function: **`MemoryInUseBy( var1 )`**. This function returns the number of bytes in use by the cached result(s) for ***`Var1`*** (with the same disclaimer that shared memory may be counted more than once). This the result reported by **`MemoryInUseBy()`** includes the memory used by mid-Value and prob-Value cached result, but it doesn't force either to be computed if the results are not already cached.

Analytica Note: *Memory profiling is only available for the Enterprise edition and ADE.*

Appendices



In the Appendices

The following appendices shows you:

- How to select an appropriate sample size
- The complete set of Analytica Menus
- The specifications for Analytica
- How to allocate and monitor memory usage in Analytica
- How to obtain the list of reserved words in Analytica
- The types of error messages you may see when you run Analytica
- The list of superseded and renamed functions
- Books and papers that are referred to in this manual or that are useful as background material

Appendix A

Appendices

A. Selecting the sample size

Each probabilistic value is simulated by computing a random sample of values from the actual probability distribution.

You can control the sampling method and sample size by using the Uncertainty Setup dialog box (see “Uncertainty Setup dialog box” on page 291). This appendix briefly discusses how to select a sample size.

Choosing an appropriate sample size

There is a clear trade-off for using a larger sample size in calculating an uncertainty Variable. When you set the sample size to a large value, the result is less noisy, but it takes a longer time to compute the distribution. For an initial probabilistic calculation, a sample size of 20 to 50 is usually adequate.

How should you choose the sample size *m?* It depends both on the cost of each model run, and what you want the results for. An advantage of the Monte Carlo method is that you can apply many standard statistical techniques to estimate the precision of estimates of the output distribution. This is because the generated sample of values for each output Variable is a random sample from the true probability distribution for that Variable.

Selecting the sample size: uncertainty about the mean

First, suppose you are primarily interested in the precision of the mean of your output Variable *y.* Assume you have a random sample of *m* output values generated by Monte Carlo simulation:

$$(y_1, y_2, y_3, \cdots y_m) \quad \textbf{(1)}$$

You can estimate the mean and standard deviation of *y* using in the following equations:

$$\bar{y} = \sum_{i=1}^{m} \frac{y_i}{m} \quad \textbf{(2)}$$

$$s^2 = \sum_{i=1}^{m} \frac{(y_i - \bar{y})^2}{(m-1)} \quad \textbf{(3)}$$

This leads to the following confidence interval with confidence α, where *c* is the deviation for the unit normal enclosing probability α:

$$\left(\bar{y} - c\frac{s}{\sqrt{m}}, \bar{y} + c\frac{s}{\sqrt{m}}\right) \quad \textbf{(4)}$$

Suppose you wish to obtain an estimate of the mean of *y* with an α confidence interval smaller than *w* units wide. What sample size do you need? You need to make sure that:

$$w > 2c\frac{s}{\sqrt{m}} \quad \textbf{(5)}$$

or, rearranging the inequality,

$$m > \left(\frac{2cs}{w}\right)^2 \quad \textbf{(6)}$$

To use this, first make a small Monte Carlo run with, say, 10 values to get an initial estimate of the variance of *y*—that is, s^2. You can then use Equation **(6)** to estimate how many samples will reduce the confidence interval to the requisite width *w*.

For example, suppose you wish to obtain a 95% confidence interval for the mean that is less than 20 units wide. Suppose your initial sample of 10 gives $s = 40$. The deviation *c* enclosing a probability of 95% for a unit normal is about 2. Substituting these numbers into Equation **(6)**, you get:

$$m > \left(\frac{2 \times 2 \times 40}{20}\right)^2 = 8^2 = 64 \quad \textbf{(7)}$$

So, to get the required precision for the mean, you should set the sample size to about 64.

Estimating confidence intervals for fractiles

Another criterion for selecting sample size is the precision of the estimate of the median and other fractiles, or more generally, the

precision of the estimated cumulative distribution. Assume that the sample *m* values of *y* are relabeled so that they are in increasing order,

$$y_1 \le y_2 \le \ldots y_m$$

and *c* is the deviation enclosing probability α of the unit normal. Then the following pair of sample values constitutes the confidence interval:

$$(y_i, y_k)$$

where

$$i = \lfloor mp - c\sqrt{mp(1-p)} \rfloor \quad \textbf{(8)}$$

$$k = \lceil mp + c\sqrt{mp(1-p)} \rceil \quad \textbf{(9)}$$

Suppose you want to achieve sufficient precision such that the α confidence interval for the *pth* fractile Y_p is given by (y_i, y_k), where y_i is an estimate of $Y_{p-\Delta p}$, and y_k is an estimate of $Y_{p+\Delta p}$. In other words, you want α confidence of Y_p being between the sample values used as estimates of the $(p-\Delta p)^{\text{th}}$ and $(p+\Delta p)^{\text{th}}$ fractiles. What sample size do you need? Ignoring the rounding, you have approximately

$$i = m(p-\Delta p), \quad k = m(p+\Delta p) \quad \textbf{(10)}$$

Thus,

$$k - i = 2m\Delta p \quad \textbf{(11)}$$

From Equations **(8)** and **(9)** above, you have

$$k - i = 2c\sqrt{mp(1-p)} \quad \textbf{(12)}$$

Equating the two expressions for $k-i$, you obtain

$$2m\Delta p = 2c\sqrt{mp(1-p)} \quad \textbf{(13)}$$

$$m = p(1-p)\left(\frac{c}{\Delta p}\right)^2 \quad \textbf{(14)}$$

For example, suppose you want to be 95% confident that the estimated fractile $Y_{.90}$ is between the estimated fractiles $Y_{.85}$ and $Y_{.95}$. So you have $\Delta p = 0.05$, and $c \approx 2$. Substituting the numbers into Equation **(14)**, you get:

$$m = 0.\dot{9}0 \times (1 - 0.90) \times (2/0.05)^2 = 144 \quad \textbf{(15)}$$

On the other hand, suppose you want the credible interval for the least precise estimated percentile (the 50th percentile) to have a 95% confidence interval of plus or minus one estimated percentile. Then,

$$m = 0.5 \times (1 - 0.5) \times (2/0.01)^2 = 10,000 \quad \textbf{(16)}$$

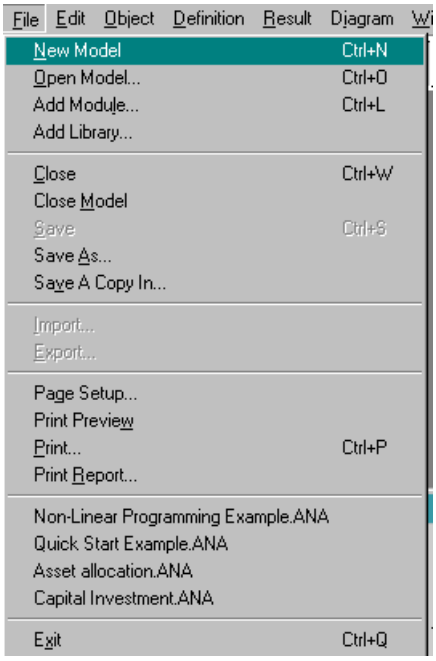
These results are completely independent of the shape of the distribution. If you find this an appropriate way to state your requirements for the precision of the estimated distribution, you can determine the sample size before doing *any* runs to see what sort of distribution it may be.

Appendix B

B. Menus

File menu

The **File** menu contains commands to open, create, and save files, as well as to import and export them. In addition, the printing and **Exit** commands are in this menu.



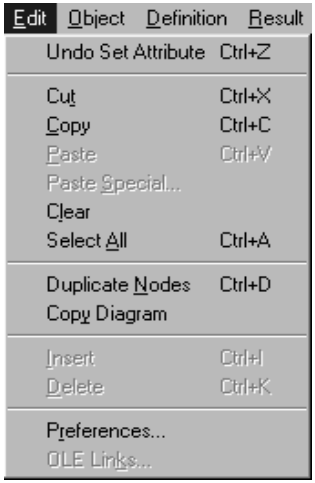
Command	Description
New Model	Starts a new model.
Open Model	Opens an existing, previously saved model.
Add Module	Adds a filed module to the active model.
Add Library	Adds a filed library to the active model.
Close	Closes the active window.

Command	Description
Close Model	Closes the active model.
Save	Saves the active model to a file, and saves each changed linked module and linked library to its own file. If the model has never been saved before, prompts for a file name and folder.
Save As	Saves the active model, filed module, or filed library as a new file. Prompts for a file name and folder.
Save A Copy In	Saves a copy of the active model, filed module, or filed library into a new file, leaving the active file name for future saves. Prompts for a file name and folder.
Import	Imports the contents of a text or data file into the selected Variable definition. See “Importing and exporting” on page 386.
Export	Exports the contents of the selected field or cells into a file. See “Importing and exporting” on page 386.
Page Setup	Displays a dialog box for selecting paper size, orientation, and scaling options for printing.
Print Preview	Displays a view showing where page breaks will occur before the current window is printed.
Print	Displays a dialog box for selecting the printer, number of copies you want to print, and other printing options.
Print Report	Displays a dialog box for printing multiple diagrams, Object windows, and result windows at the same time. See “Printing” on page 38.
Recent files	The four most recently opened Analytica models are listed on the **File** menu. Selecting one loads the corresponding model into memory.
Exit	Quits the Analytica application. Confirms if you wish to save changes to the current model.

Edit menu

The **Edit** menu contains commands to manipulate objects, text or graphics, and display the Preferences dialog.

Appendix B

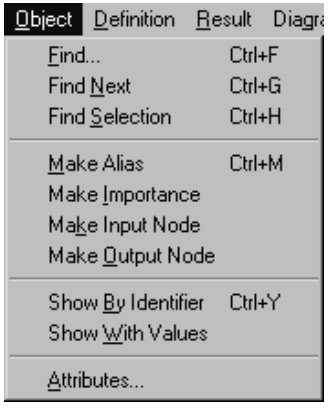


Command	Description
Undo	Undoes your last action.
Cut	Cuts the selected text, nodes, graph, or table cells into the clipboard temporarily for pasting.
Copy	Copies the selected text, nodes, graph, or table cells into the clipboard temporarily for pasting. See “Copying and pasting” on page 375.
Paste	Pastes the contents of the clipboard at the insertion point or replaces the current selection. See “Copying and pasting” on page 375.
Paste Special...	Brings up a dialog for selecting the format of data to OLE link into an Edit Table.
Clear	Deletes the selected text or node(s).
Select All	Selects all text, nodes, or table cells.
Duplicate Nodes	Duplicates the selected nodes. See “Duplicating nodes” on page 76.
Copy Diagram/ Table	When a Result table or Edit Table is active, this command is **Copy Table**, which copies the entire multidimensional Object as a tab-delimited list of tables. When a Diagram window is active, this menu command is **Copy Diagram**, which copies a picture of the diagram without copying the objects they represent. See “Copying and pasting” on page 375.

Command	Description
Insert Rows	Inserts an item in a list, or a row in a table, by copying the current item, or row. If a column in a table is selected, this command changes to **Insert Columns**. See “Editing a table” on page 228.
Delete Rows	Deletes the selected item or items in a list, or rows in a table. If a column in a table is selected, this command changes to **Delete Columns**. See “Editing a table” on page 228.
Preferences	Displays the Preferences dialog box to examine or change various options. See “Preferences dialog box” on page 88.
OLE Links...	Brings up a dialog that allows properties to be changed for OLE links from external applications into your model. Used to change links from Manual to Automatic, to update manual links, or to open an external application that is serving a link.

Object menu

The **Object** menu contains commands that find and create Analytica objects and display their attributes.

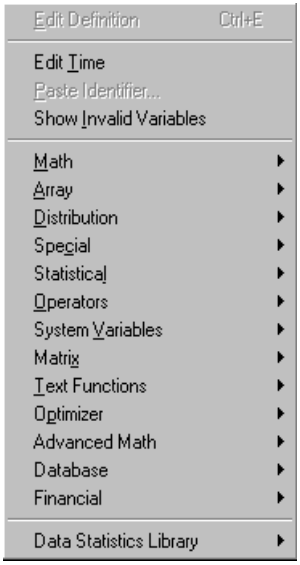


Appendix B

Command	Description
Find	Displays a dialog box to find an Object by its identifier or title. See “Finding Variables” on page 399.
Find Next	Finds the next Object that partially matches the previously defined text value. See “Finding Variables” on page 399.
Find Selection	Finds an Object by its identifier that matches the currently selected text. See “Finding Variables” on page 399.
Make Alias	Creates an alias for the selected Object. See “Alias nodes” on page 82.
Make Importance	Creates an Index and General Variable for the selected Variable to compute the uncertainty importance (rank correlation) contributions of its inputs. See “Importance analysis” on page 343.
Make Input Node	Creates an input node as an alias of the selected Object. See “Using input nodes” on page 161.
Make Output Node	Creates an output node as an alias of the selected Object. See “Using output nodes” on page 164.
Show By Identifier	Shows Variables by their identifier in the current diagram, Edit Table, Result window, or Outline view.
Show With Values	Shows the mid values of the Variable and all its inputs. See “Showing mid values” on page 36 and “The Outline window” on page 397.
Attributes	Opens the Attribute dialog box to set the visibility of attributes and define new attributes. See “Managing attributes” on page 400.
Hide Definition(s)	(Analytica Enterprise only) Marks the currently selected node or module as hidden. The definitions for all Variables contained within the selected node will be private.
Unhide Definition(s)	(Analytica Enterprise only) Unhides the currently selected node or module. This overrides any settings in parent modules to hide definitions.

Definition menu

The **Definition** menu contains commands for editing Variable definitions. It lists Analytica's built-in function libraries, as well as any user libraries that are currently open.



Command	Definition
Edit Definition	Opens the appropriate view for editing the definition of the selected Variable. If the Variable is defined as a distribution or sequence, the Object Finder opens. If it is defined as a table or probability table, its Edit Table window opens. Otherwise, an Object window or Attribute panel opens, depending on the Edit attributes setting in the Preferences dialog box. See "Preferences dialog box" on page 88.
Edit Time	Opens the Object window for the *Time* system Variable. See "The Time index" on page 361.
Paste Identifier	Opens the Object Finder dialog box for examining functions and Variable identifiers, entering function parameters, and pasting them into definitions. See "Object Finder dialog box" on page 152.
Show Invalid Variables	Displays a window listing all Variables with invalid or missing definitions. See "Invalid Variables" on page 403.

Appendix B

Command	Definition	
Math	Displays a list of the mathematical functions in the Math library. See “Math functions” on page 190.	
	`Abs()`	`Logten()`
	`Arctan()`	`Mod()`
	`Ceil()`	`Radians()`
	`Cos()`	`Round()`
	`Degrees()`	`Sin()`
	`Exp()`	`Sqr()`
	`Factorial()`	`Sqrt()`
	`Floor()`	`Tan()`
	`Ln()`	
Array	Displays a list of functions for creating and manupulating arrays. See Chapter 11, “Arrays and Indexes,” and Chapter 12, “Advanced Array Functions.”	
	`Area()`	`Min()`
	`Array()`	`Normalize()`
	`Average()`	`Product()`
	`Choice()`	`Rank()`
	`Concat()`	`Sequence()`
	`CopyIndex()`	`Size()`
	`CumProduct()`	`Slice()`
	`Cumulate()`	`SortIndex()`
	`Determtable()`	`Subscript()`
	`IndexNames()`	`Subset()`
	`Integrate()`	`Sum()`
	`Max()`	`Table()`
	`MdArrayToTable()`	`Uncumulate()`
	`MdTable()`	`Unique()`

Command	Definition	
Distribution	Displays a list of functions for creating probability distributions in the Distribution library. See Chapter 14, “Probability Distributions,” and Chapter 15, “Using Discrete Probability.”	
	`Bernoulli()`	`Logistic()`
	`Beta()`	`Lognormal()`
	`Binomial()`	`Normal()`
	`Certain()`	`Poisson()`
	`Chancedist()`	`ProbDist()`
	`ChiSquared()`	`ProbTable()`
	`Cumdist()`	`StudentT()`
	`Exponential()`	`Triangular()`
	`Fractiles()`	`Truncate()`
	`Gamma()`	`Uniform()`
	`Geometric()`	`Weibull()`
	`Hypergeometric()`	
Special	Displays a list of unusual or less commonly used functions in the Special library.	
	`Argmax()`	`IsReference()`
	`Attrib of ...`	`Istext()`
	`Cubicinterp()`	`IsUndef()`
	`Dydx()`	`Iterate()`
	`Dynamic()`	`Linearinterp()`
	`Elasticity()`	`MsgBox()`
	`Evaluate()`	`Stepinterp()`
	`For...Do`	`Subindex()`
	`x[I=....]`	`Using...Do`
	`x[time-...]`	`WhatIf()`
	`Index...Do`	`WhatIfAll()`

Command	Definition	
	`Isnan()`	`While`
	`Isnumber()`	
Statistical	Displays a list of statistical functions in the Statistical library. See “Statistical functions” on page 335.	
	`Correlation()`	`Probbands()`
	`Frequency()`	`Rankcorrel()`
	`Getfract()`	`Sample()`
	`Kurtosis()`	`Sdeviation()`
	`Mean()`	`Skewness()`
	`Mid()`	`Statistics()`
	`Probability()`	`Variance()`
Operators	Displays a list of arithmetic, comparison, logical, and conditional operators in the Operators library. See “Operators” on page 180.	
	`(- )`	`Not`
	`+`	`Or`
	`-`	`And`
	`*`	`If Then Else`
	`/`	`Ifall Then Else`
	`÷`	`Ifonly Then Else`
	`^`	`expr1; expr2`
	`<`	`A.1`
	`<=`	`first..last`
	`=`	`X := expr`
	`<>`	`\[indexes]A`
	`>=`	`# A`
	`>`	`a & b`

Command	Definition	
System Variables	Displays a list of system Variables that you can use in definitions (see the next section).	
Matrix	Displays a list of matrix functions in the Matrix library. See “Matrix functions” on page 267.	
	`Decompose()`	`MatrixMultiply()`
	`Determinant()`	`SingularValue()`
	`EigenDeComp()`	`Transpose()`
	`Invert()`	
Text Functions	Displays a list of matrix functions in the Matrix library. See “Matrix functions” on page 267.	
	`Acs()`	`TextLength()`
	`Chr()`	`TextLowerCase()`
	`FindInText()`	`TextReplace()`
	`JoinText()`	`TextSentence()`
	`SplitText()`	`TextUpperCase()`
Optimizer	Displays a list of matrix functions in the Matrix library. See “Matrix functions” on page 267.	
	`LpDefine()`	`LpSlack()`
	`LpFindIIS()`	`LpSolution()`
	`LpObjSA()`	`LpStatusNum()`
	`LpOpt()`	`LpStatusText()`
	`LpRead()`	`LpWrite()`
	`LpReducedCost()`	`LpWriteIIS`
	`LpHSSA()`	`NLPDefine()`
	`LpShadow()`	`QpDefine()`

Appendix B

Command	Definition	
Advanced Math	Displays a list of advanced and specialized mathematical and statistical functions. See “Advanced math functions” on page 192.	
	`Arccos()`	`ErfInv()`
	`Arcsin()`	`GammaFn()`
	`Arctan2()`	`GammaI()`
	`BetaFn()`	`GammaIInv()`
	`BetaI()`	`Lgamma()`
	`Combinations()`	`Permutations()`
	`Cosh()`	`Regression()`
	`CumNormal()`	`Sinh()`
	`CumNormalInv()`	`Tanh()`
	`Erf()`	
Database	Appears only in Analytica Enterprise. Contains a list of functions for accessing ODBC-compliant databases. See “Database functions” on page 484.	
	`DbLabels()`	`DbTableNames()`
	`DbQuery()`	`DbWrite()`
	`DbTable()`	`SqlDriverInfo()`

Command	Definition	
Financial	Displays a list of financial functions. See “Datatype functions” on page 200.	
	`Cumipmt`	`Pmt()`
	`Cumprinc()`	`Ppmt()`
	`Fv()`	`Pv()`
	`IPmt()`	`Rate()`
	`Irr()`	`Xirr()`
	`Nper()`	`Xnpv()`
	`Npv()`	
your libraries	Any libraries that you have defined or added to the model are listed at the bottom of the **Definition** menu, each with a submenu that lists the functions contained in the library. See Chapter 19, “Building Functions and Libraries.”	

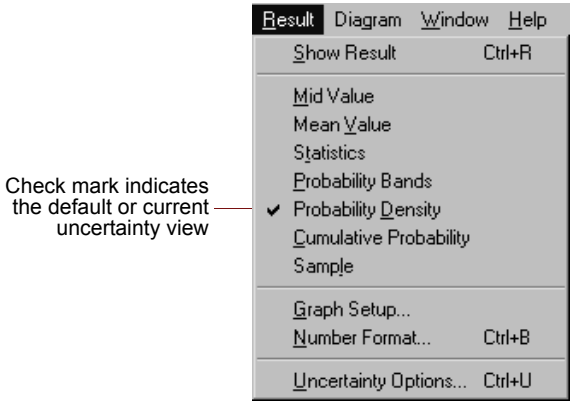
System Variables submenu

The system Variables submenu lists the Analytica Variables and constants that can be used in Variable definitions.

Command	Description
AnalyticaPlatform	In Analytica for Windows, this is 'Windows'. From Analytica for Macintosh, this is 'Macintosh', and from the Analytica Decision Engine this is 'ADE'.
AnalyticaVersion	An integer encoding the current build number of Analytica being run. In terms of the major release number, minor release number, and sub-minor release number, it is equal to $10K \cdot Major + 100 \cdot Minor + SubMinor$ For example, Analytica 3.1 subminor version 1 returns the value 31001.
False	The logical (Boolean) constant that evaluates numerically to zero.
Pi	The ratio of circumference to the diameter of a circle.
Run	The index for uncertainty sampling, defined as `Sequence(1,Samplesize)`.
Samplesize	The number of sample iterations for probabilistic simulation. See "Uncertainty Setup dialog box" on page 291.
Time	The index Variable identifying the dimension for dynamic simulation (the `Dynamic()` function). See "The Time index" on page 361.
True	The logical (Boolean) constant that evaluates numerically to nonzero.

Result menu

The **Result** menu contains commands for opening Result windows, changing the appearance of graphs and tables, and setting uncertainty options.

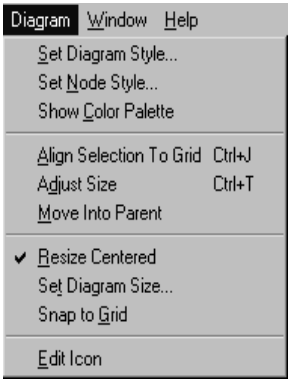


Command	Description
Show Result	Opens a Result window for the selected Object. See “2: Results” on page 45.
Mid Value	Displays the deterministic value, holding most uncertain Variables to their median value. See “Uncertainty view options” on page 52.
Mean Value	Displays the mean of the uncertain value. See “Uncertainty view options” on page 52.
Statistics	Displays the statistics of the uncertain value in a table as set in the Uncertainty Setup dialog box. See “Uncertainty view options” on page 52.
Probability Bands	Shows probability bands as set in the Uncertainty Setup dialog box. See “Uncertainty view options” on page 52.
Probability	Displays a probability density graph for an uncertain value. For a discrete probability distribution, **Probability Mass** replaces this command. See “Uncertainty view options” on page 52.
Cumulative Probability	Displays a cumulative probability graph representing the probability that a Variable's value is less than or equal to each possible (uncertain) value. See “Uncertainty view options” on page 52.
Sample	Displays a table of the values determined for each uncertainty sample iteration. See “Uncertainty view options” on page 52.

Command	Description
Graph Setup	Displays a dialog box to specify the graphing tool, graph frame, and graph style. See “Graph Setup dialog box” on page 129.
Number Format	Displays a dialog box to set the number format for displays of results. See “Number Format dialog box” on page 135.
Uncertainty Options	Displays a dialog box to specify the uncertainty sample size and sampling method and to set options for statistics, probability bands, probability density, and cumulative probability. See “Uncertainty Setup dialog box” on page 291.

Diagram menu

The **Diagram** menu contains commands for changing the display of the diagram, including nodes, arrows, and fonts.

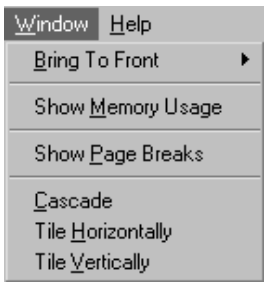


Command	Description
Set Diagram Style	Displays a dialog box to set default arrow displays, node size, and font. See “Diagram Style dialog box” on page 121.
Set Node Style	Displays a dialog box to set arrow display and font for specific nodes. See “Node Style dialog box” on page 123.
Show Color Palette	Displays a palette to set the color of the diagram background or of selected nodes. See “Changing background or node colors” on page 120.

Command	Description
Align Selection To Grid	Aligns selected node(s) to the diagram grid. See “Aligning to the grid” on page 76.
Adjust Size	Adjusts the selected node’s size to match the default node size, or to fit the title label. See “Default node size” on page 122.
Move Into Parent	Moves the selected Object from the current diagram to its parent diagram. See “Influence diagram window” on page 28.
Resize Centered	If checked, when you resize a node, the node’s center stays unmoved. If unchecked, when you resize a node by dragging a corner handle, the opposite handle stays unmoved. See “Align nodes horizontally or vertically” on page 115.
Set Diagram Size	(Obsolete) Displays a dialog box to set the number of diagram pages. See “Changing the size of the diagram” on page 124.
Snap to Grid	Turns alignment to the diagram grid on or off in edit mode. See “Aligning to the grid” on page 76.
Edit Icon	Opens a window to edit the icon for the selected node. See “Adding icons to nodes” on page 168.

Window menu

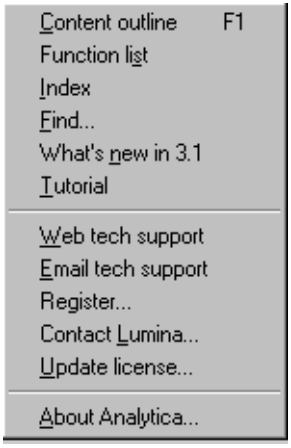
The **Window** menu contains commands for bringing windows to the front, and for opening special windows.



Command	Description
Bring to Front	Displays a list of the current windows; select one to display on top.
Show Memory Usage	Displays a window showing memory usage. See “Memory” on page 524
Show Page Breaks	Shows page breaks for the currently active diagram.
Cascade	Rearranges the open Analytica windows. All windows are resized to be a standard size. The first window is placed at the upper left corner of the main Analytica window, with each subsequent window offset slightly below and to the right.
Tile Horizontally	Rearranges the open Analytica windows. Windows are resized and repositioned to tile the main Analytica window area horizontally.
Tile Vertically	Rearranges the open Analytica windows. Windows are resized and repositioned to tile the main Analytica window area vertically.

Help menu

The **Help** menu allows you to access to Analytica’s online help system.



Command	Description
Content Outline	Displays an outline of the documentation
Function List	Displays the list of Analytica functions
Index	Opens the Analytica Users Guide Index
Find	Opens a documentation search window
What's New in 3.1	Opens the Users Guide to the "Wat's new in 3.1" section
Tutorial	Opens the Analytica Tutorial
Optimizer	Opens the Optimizer Manual (only appears in optimizer-enabled version of Analytica)
Web Tech Support	Opens your default web browser to the Analytica Tech Support page at: *http://www.lumina.com.*
Email Tech Support	Opens your email system to send an email to Analytica Tech Support
Register	Opens your default web browser to the Analytica softwareregistration page at: *http:// www.lumina.com.*
Contact Lumina	Provides contact information for Lumina
Update License	Displays your current Analytica license information and allow you to update the license code.
About Analytica	Displays useful information such as the application's edition, release number, your license code, and contact information.

Analytica Note: *The options that appear on the help menu will vary depending on your computer setup and the version of Analytica you have. If you do not have Adobe Acrobat installed on your computer, the items that appear above the line will change to only:*

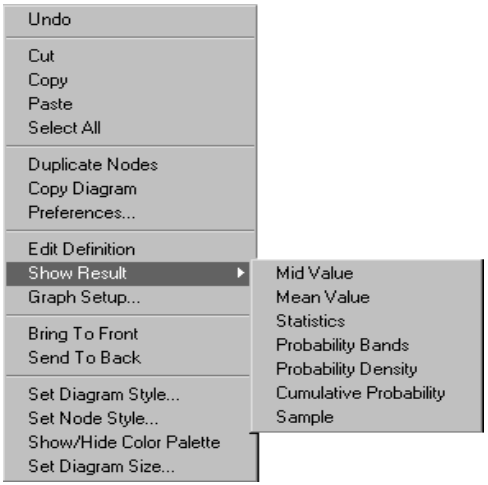
- *User guide*
- *Optimizer (if you have purchased the Optimizer)*
- *Tutorial*

Users with free Acrobat Reader version 6.0 or earlier, or users with Acrobat Standard or Acrobat Professional, see the expanded

listing shown. Users with the free Acrobat Reader version 7.0 or later will only see the truncated list.

Right mouse button menus

Many of common commands found on the Analytica application menu are also available from a right mouse button pop-up menu.



The above pop-up appears when you depress the right mouse button with the mouse above a node in a diagram window while in edit mode. A slightly different set of options appear if you are in browse mode, or if you depress the right mouse button while the mouse is over the diagram background.

There are only two menu items that are appear only on the right mouse button menu:

Command	Description
Bring to Front	Brings the selected Object(s) to the front of the drawing order so that if the Object(s) overlap any other elements, the Object will be visible.
Send to Back	Sends the selected Object(s) to the back of the drawing order so that the selected Object(s) are drawn behind any overlapping elements.

Appendix C

C. Analytica specifications

Hardware and software

CPUs supported	486 and higher (Pentium recommended)
System Software	Windows 98, 2000, NT 4, ME, or XP
Memory requirements	16 MB (24 MB+ recommended)
Application size	Approximately 6 MB
Typical model file size	20K (small)–200K (large)

Objects

Number of system objects	619
Maximum user-defined objects	14730
Maximun number of local-variables	31

Uncertainty

Probability methods	Random Latin HyperCube Median Latin HyperCube Monte Carlo
Maximum sample size	30,000 (Analytica Professional) 99,999,999 (Analytica Enterprise) *also limited by available memory*
Random sampling methods	SANE Minimal Standard L'Ecuyer Knuth

Numbers and arrays

Number precision	15 significant digits for floating-point numbers 9 digits for integers

Maximum elements in a dimension	30,000 (Analytica Professional) 99,999,999 (Analytica Enterprise and Power Player)
Maximum dimensions in an array	15

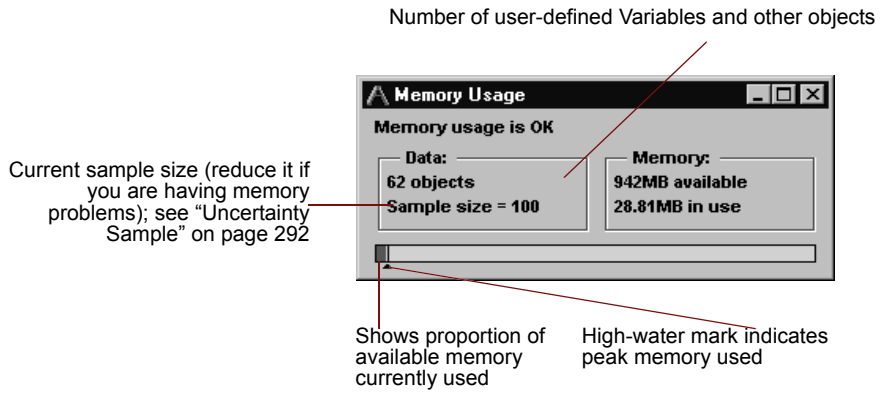
Appendix D

D. Memory

Memory usage

The Memory Usage window displays the amount of memory available on your system, as well as the memory currently in use by all applications, including Windows itself. The memory available on your system is the sum of all physical memory installed on your system and the swapfile on your hard disk, which is used to complement the physical memory.

To display the Memory Usage window, select **Show Memory Usage** from the **Window** menu.



Analytica Note: *This window appears automatically when Analytica runs low on memory.*

If you require additional memory to run your model at a given sample size, you can take several steps to increase the amount of memory available to Analytica:

1. Close other open applications.

   All applications require a segment of memory to operate, and this reduces the memory available to Analytica.

2. Increase the size of your computer’s swapfile.

   Under Windows 95, the swapfile size is dynamically handled by the system by default, and is limited only by the free space available on the hard disk where the swapfile resides. You

can manually change swapfile settings in the **Memory** control panel.

Under Windows NT, the minimum swapfile size is set through the **Hardware** control panel. If your hard disk is full or nearly full, you will need to free space on your hard disk, or select a different hard disk to hold your swapfile, in order to provide more memory for Analytica computations.

3. Finally, consider adding more physical memory to your computer.

Memory message on opening a model

When you save a model, the number of megabytes of peak memory used during the session is also saved. When you open the model, the saved peak memory is compared to the amount of memory allocated to Analytica. If the saved peak memory exceeds 95% of Analytica's memory allocation, a message will recommend either reducing the sample size (see "Uncertainty Setup dialog box" on page 291) or changing the application memory size (see next section).

Appendix E

E. Reserved Words

Identifiers used for Analytica's builtin functions, attributes, classes, attributes, and other objects may not be reused for new objects. Analytica will warn you if you try to do so. You can list all the reserved words thus:

1. Press **CTRL-'**, to open the Typescript Window
2. Type in 'List', followed by **Enter**.

Appendix F

F. Error message types

There are several types of error messages in Analytica. Many messages are designed to inform you that something in the model needs to be corrected; some messages indicate that Analytica cannot continue or complete your request. Each error message begins with its message type, one of: warning, lexical, syntax, evaluation, system, and fatal errors.

In general, Analytica allows you to continue working on your model unless it cannot proceed until a problem has been corrected. When you are editing a Variable definition, you can request an error message by pressing *Alt-Enter* or by clicking on the definition Warning icon().

Warning

A warning indicates that there is a possible problem. For example:

> **Warning:**
>
> **Log of non-positive number.**

A warning is reported during result evaluation to inform you that continuing may yield unexpected results.

You can suppress evaluation warnings for all Variables by disabling the **Show result warnings** preference (see “Preferences dialog box” on page 88). When **Show result warnings** is unchecked, any warning conditions encountered during result evaluation will be ignored. You can also suppress warnings during evaluation of a single expression with the `IgnoreWarnings(`*`expr`*`)` function. See “IgnoreWarnings(expr)” on page 466 for details.

If an identifier in a module you are adding to a model has a name conflict with an identifier in the model, you will see a warning similar to the following:

Warning:

Can't declare Variable Location because the Identifier is already in use as Attribute Location.

Declare using the Identifier Location1?

Lexical error

A lexical error occurs when a component of an expression was expected and is missing or is invalid. For example, if you enter a number with an invalid number suffix, you may get a message similar to the following:

Lexical error while checking:

2sdf
^

Invalid exponent code.

Syntax error

A syntax error occurs when an expression contains a syntax mistake. Analytica often reports the mistake together with the fragment of the expression that contained the error. For example:

Syntax error while checking:

2 + + 3
^

Expression expected.

The following are two common syntax errors:

Expecting ","

Indicates a comma is missing, or there are too few parameters to a function.

Expecting ")"

Indicates there are too many parameters to a function.

If you attempt to change the identifier for a Variable, and the new identifier is assigned to another node, you will see a message similar to the following:

Syntax error:

The Identifier "Location" is already in use.

Evaluation error

An evaluation error occurs when there is a problem while evaluating a Variable, user-defined function, or system function. You are asked if you want to edit the definition of the Variable currently being evaluated:

Error during evaluation of Ch1.

Do you want to edit the Definition of Ch1?

If a system function expects a specific kind of argument, an error message similar to the following is displayed:

Evaluation error:

First argument of Sysfunction Argmax must be a table.

This message indicates that an argument passed to the function is of a different type or cannot be handled by that function. You may need to redefine a Variable being used as an argument to the function, or change an expression being passed as an argument.

Invalid number

If a calculation such as division by zero is performed, a warning is displayed with an option to continue calculating. Three possible error codes may be returned as a result of an invalid calculation:

Code	Meaning
`INF`	Infinity.
`NAN`	Invalid argument, such as `Sqrt(-1)`, or Invalid division, such as `0/0`.
Undefined (blank)	Displays as a blank cell if the result is a table, or shows the **Compute** button otherwise. Results from certain functions, such as `SubIndex()`, when a result is not available.

These can be detected in expressions using `"X=INF"`, `Isnan(X)`, or `Isundef(X)`.

System error

If you see this message type, please contact Lumina Decision System's technical support department (see "How to contact us" on page 17) to report the error.

Out of memory error

Indicates that Analytica has used up all available memory and cannot complete the current command. If this occurs, first save your model. Before attempting to evaluate again, close some windows, use a smaller sample size, or expand the memory available to Analytica (see "Memory" on page 524).

Appendix G

G. Forward and backward compatibility

Superseded functions and constructs

Analytica 3.1 includes several syntax constructs and functions that have been superseded, so it will run models from earlier releases that used them. We strongly encourage you to use the new constructs and avoid the superseded ones—they may not be supported in future releases. Use the old ones only if you absolutely require backward compatibility with colleagues using earlier releases of Analytica, such as the Macintosh edition.

Analytica Note: *there are some caveats to the general ability to run Analytica 3.0 models in Analytica 3.1.*

- *Two pixels were added to the left-hand margin of text nodes for improved athetics & readability, so text nodes in 3.0 models may wrap their text and need adjustment.*
- *Any expression making use of a binary operation on the value* `Null` *will evaluate differently in 3.1. For example,* `5*Null` *evaluates to 5 in Analytica 3.0 but evaluates to* `Null` *in Analytica 3.1. In most instances, this change would expose errors in Analytica 3.0 models that went undetected and is unlikely to impact intended behavior, but it may cause error messages to occur that did not appear when the model was run in Analytica 3.0.*
- *Analytica3.1 contains numerous bug fixes, so any model that (perhaps inadvertently) utilized a bug to its advantage could be impacted.*

It is also possible to run models created or edited in Release 3.1 in Analytica 3.0, provided you don't use any functions or new syntax options, such as calling function parameters by name, added in 3.1. When opening a 3.1 model in Analytica 3.0, you may get a warning about an unrecognized attribute `Att__discretenessinf`, which you can safely ignore.

Encouraged	Superseded	Meaning	For more see:
Var ***x*** := ***e ; f***	Using ***x*** := ***e*** Do ***f***	Define local Variable ***x***, assign it initial value ***e***, and evaluate expression ***f*** which may refer to ***x***.	page 439
For ***x***[***j, k..***] := ***a*** Do ***e***	Using ***x*** := ***a In i*** Do ***e***	Assigns to local Variable ***x***, successive subarray values from array indexed by all indexes of ***a*** other than ***i*** and repeats evaluation of expression ***e*** for each value of ***i***. Returns an array of values of ***e*** with the same indexes as ***a***.	page 443
For ***x***[] := ***i*** Do ***e***	Using ***x*** := ***i*** Do ***e***	Assigns to local Variable ***x***, successive scalar values from index ***i*** and repeats evaluation expression ***e*** for each value. Returns an array indexed by ***i***.	page 439
While ***Test*** Do ***Body***	Iterate()	**Iterate** provides an iterative convergence algorithm. Most such algorithms can now be more clearly and reliably implemented with the **While** construct.	page 446 & page 447
/****comments*** */	{ ***comments*** }	Brackets enclose comments within definitions.	page 147
text & ***text***	***text*** + ***text***	Joining text values into a single text value	page 195
JoinText(*t*, *i*)	Join(*t*, *i*, ‘’) Sum(*t*, *i*)	To join elements of an array of text and/or numbers ***t*** over index ***i*** into one text value	page 195

Renamed text functions

We have renamed previous text functions to be consistent with Analytica's terminology, adding the word *Text*, or using it instead of *String* (the term used in some other computer languages). The old function names still work in 3.1 for compatibility with old models, but we encourage you use the new *Text* ones for greater consistency. The following functions have been renamed.

Old Function Name	New Name
`Join()`	`JoinText()`
`Split()`	`SplitText()`
`StringLength()`	`TextLength()`

Appendix G

Old Function Name	New Name
`StringLowerCase()`	`TextLowerSCase()`
`StringMixedCase()`	`TextSentenceCase()`
`StringReplace()`	`TextReplace()`
`StringUperCase()`	`TextUpperCase()`
`SubFindString()`	`FindInText()`
`SubString()`	`SelectText()`

Appendix H

H. Bibliography

Morgan, M. Granger and Henrion, Max. *Uncertainty: A Guide to Dealing with Uncertainty in Quantitative Risk and Policy Analysis,* Cambridge University Press (1990,1998).

Written by the original authors of Analytica, this text provides extensive background on how to represent and analyze uncertainty in quantitative models. It includes chapters on:

- Building good policy models
- Categorizing types and sources of uncertainty
- How people make judgments under uncertainty
- Encoding expert judgment in the form of probability distributions
- Choosing a computational method for propagating uncertainty in a model
- Analyzing uncertainty in very large models
- Displaying and communicating uncertainty
- How to tell if representing uncertainty could make a significant difference to your conclusions, or "the value of knowing how little you know"

We recommend the second edition, published 1998, which contains a full chapter on Analytica (Chapter 10). If you have the first edition (1990), we recommend that you ignore Chapter 10, which describes the precursor of Analytica and is quite out of date!

Clemen, Robert T. *Making Hard Decisions: An Introduction to Decision Analysis*. Duxbury Press (1991).

Howard, R., and Matheson, J. Influence Diagrams. In *Readings on the Principles and Applications of Decision Analysis*, eds. R. Howard and J. Matheson. pp. 721-762. Menlo Park, Calif.: Strategic Decisions Group (1981).

Keeney, R. *Value–Focused Thinking: A Path to Creative Decision Making,* Cambridge, MA: Harvard University Press (1992).

Knuth, D.E. *Seminumerical Algorithms, 2nd ed., vol. 2 of The Art of Computer Programming,* Reading, MA: Addison-Wesley (1981).

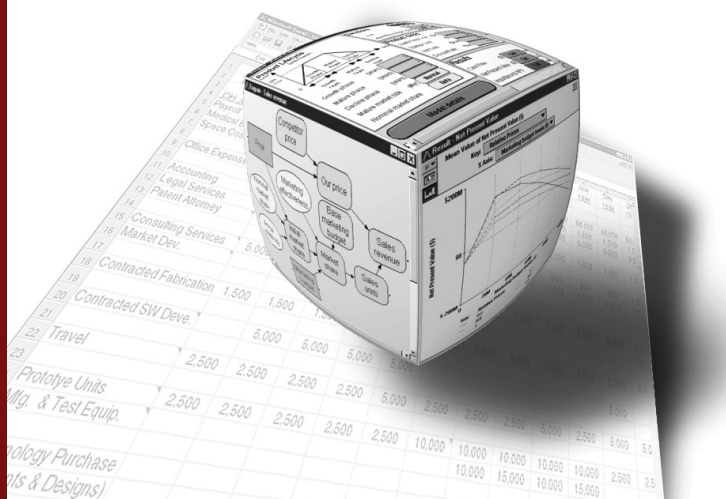
L'Ecuyer, P. *Communications of the ACM,* **31**, 742-774 (1988).

Park, S.K., and K.W. Miller. *Communications of the ACM,* **31**, 1192-1201 (1988).

Pearl, J. *Probabilistic Reasoning in Intelligent Systems*, San Mateo, Calif.: Morgan Kaufmann (1988).

Appendix H

Function List



In this Chapter

This appendix lists all the built-in functions in Analytica, organized by category. It also contains information on obsolete functions and constructs, explaining what functions have taken their place

Function list

When viewing this list online, click on the category or function name to see details.

Basic Math
Abs, Arctan, Ceil, Cos, Degrees, Exp, Factorial, Floor, Ln, Logten, Mod, Radians, Round, Sin, Sqr, Sqrt, Tan

Advanced Math
Arccos, Arcsin, Arctan2, BetaFn, BetaI, Combinations, Cosh, CumNormal, CumNormalInv, Erf, ErfInv, GammaFn, GammaI, GammaIInv, Lgamma, Permutations, Regression, Sinh, Tanh

Creating Arrays
[...], m..n, Array, CopyIndex, Sequence,Table

Array-Reducing
Area, Argmax, Average, Max, Min, Product, Subindex, Sum

Transforming arrays
Cumproduct, Cumulate, Integrate, Normalize, Rank, Sortindex, Uncumulate

Selecting from Arrays
v[I=v], Choice, Slice, Subscript

Interpolating
Cubicinterp, Linearinterp, Stepinterp

Other Array functions
Concat, IndexNames, Size, Sortindex, Subindex, Subset, Unique

Tables and arrays
MDArrayToTable, MDTable

Matrix functions
Decompose, Determinant, Determtable, DotProduct, Invert, Transpose, SingularValueDecomp

Continuous distributions
Beta, Chisquared, Cumdist, Exponential, Fractiles, Gamma, Logistic, Lognormal, Normal, Probdist, StudentT, Triangular, Truncate, Uniform, Weibull

Discrete distributions
Bernoulli, Binomial, Certain, Chancedist, Geometric,Hypergeometric, Poisson, Probtable

Statistical functions
Frequency, Getfract, Kurtosis, Mean, Mid, Probability, Probbands, Rankcorrel, Regression, Sample, Sdeviation, Skewness, Statistics, Variance

Text functions
&, Asc, Chr, FindinText, JoinText, SelectText, SplitText, TextUpperCase, TextLength, TextLowerCase, TextSentenceCase, TextReplace, ReadTextFile, WriteTextFile

Sensitivity analysis
Correlation, Dydx, Elasticity, RankCorrel, Regression, Whatif, WhatIfAll

Special functions
Dynamic, Error, Evaluate, IgnoreWarnings, Iterate, Subindex, Time, Today, MsgBox, Whatif, WhatIfAll

Financial functions
Cumipmt, Cumprinc, Fv, Ipmt, Irr, Nper, Npv, Pmt, Ppmt, Pv, Rate, Xirr, Xnpv

Operators
+ - * / ^ < <= = <> >= > := & \ # NOT OR AND OF

Database access
DBLabels, DBQuery, DBTable, DBTableNames, DBWrite, SqlDriverInfo
(Enterprise edition Only)

Datatypes
Isnan, Isnumber, IsReference, Istext, Isundef

Control constructs
(s1;s2;...), Begin … End, Error, For, FunctionOf, Index, If, IfAll, IfOnly, IgnoreWarnings, Iterate, MemoryInUseBy, Var, While

System Variables
AnalyticaPlatform, AnalyticaVersion, CurrentDataDirectory, CurrentModelDirectory, Run, Samplesize, Time

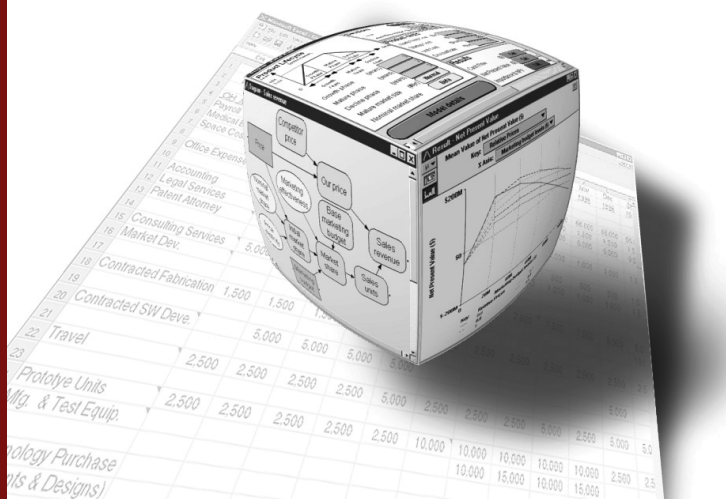
System constants
False, Null, Pi, True, Inf

Object classes
Chance, Constant, Decision, Determ, Form, Index, Library, Model, Module, Objective, Variable

Parameter qualifiers
All, Atomic, ArrayType, Ascending, Coerce, Context, Descending, DetermType, IndexType, IsNotSpecified Numeric, Optional, Positive, ProbType, ReferenceType, Sample, Scalar, TextType,Vartype, Vector

Optimizer functions
LpDefine, LpFindIIS, LpObjSA, LpOpt, LpRead, LpSolution, LpStatusNum, LpStatusText, LpWrite, LpWriteIIS, NlpDefine, QpDefine

Glossary



In this Chapter

A compilation of terms specific to Analytica as well as statistical terms used in this manual.

Glossary

ADE — See “Analytica Decision Engine.”

Alias — A node in a diagram that refers to a Variable or other node located somewhere else, usually in another module. An alias permits you to display a Variable in more than one module. An alias node is distinguished by having its title in italics.

Analytica Browser — A free edition of Analytica that allows a user to evaluate and view results, and change input fields; however, from Analytica Browser a user cannot enter edit mode or otherwise change the content of a model. Copies of Analytica without a valid registration number run as the Analytica Browser.

Analytica Decision Engine — A product sold by Lumina Decision Systems, Inc., separate from Analytica. With the Analytica Decision Engine (ADE), you embed the Analytica computation engine in your web-server backend or in your custom applications built in Visual Basic, C++, Microsoft Office, or any language supporting ActiveX Automation or COM.

Analytica Enterprise — A edition of Analytica for users who intend to share data or models with others in their organization. Analytica Enterprise contains all features of Analytica Pro as well as functions for accessing ODBC databases and features for protecting your intellectual property.

AnalyticaProfessional edition — The standard fully-functional edition of Analytica. Analytica Pro provides all the features and functionality required to create, edit, and evaluate models.

Analytica Trial — A fully-functional, but expiring, edition of Analytica. Analytica Trial can be downloaded from the Lumina web site (www.lumina.com) for those wishing to "test drive" the product. Analytica Trial contains the complete functionality of Analytica Pro. After expiration, Analytica Trial converts to Analytica Browser.

Array — A collection of values that can be viewed as one or more tables. An array has one or more dimensions; each dimension is identified by an index.

Array abstraction — See “Intelligent Array Abstraction™.”

Arrow — An arrow or influence from one Variable node to another indicates that the origin node affects (influences) the destination node. If the nodes depict Variables, the origin Variable usually appears in the definition of the destination Variable.

Arrow tool The Arrow tool, or Influence Arrow tool, is in the shape of a left-to-right pointing arrow cursor. The Arrow tool is used to draw arrows connecting Variables to create relations between them.

Attribute A property or descriptor of an Object, such as its title, description, definition, value, or inputs.

Attribute panel An auxiliary window pane that you can open below a diagram or outline window. Use the Attribute panel to rapidly examine one Attribute at a time of any Variable in the model, by selecting the Variable and then the Attribute from a popup menu.

Author An Attribute recording the names of the person or people who created the model, or other Object.

Behavior analysis Model behavior analysis is a type of sensitivity analysis in which you specify a set of alternative values for one or more inputs and examine the effect on selected model output Variables. It is also known as parametric analysis.

Browse-only models Analytica Enterprise users can save a copy of their model in a browse-only form. When a browse-only model is loaded into any edition of Analytica, the user cannot enter edit mode, and therefore can only make changes to Variables with input nodes. Browse-only models are also obfuscated.

Browse tool The Browse tool is in the shape of a hand. With the Browse tool, you can examine the diagram but cannot make any changes, except to change the values in input nodes.

Chance Variable A Chance Variable is uncertain and cannot be directly controlled by the decision maker. Usually, it is defined by a probability distribution. A Chance Variable is depicted as an oval node.

Check The check Attribute contains an expression that checks the validity of the value of a Variable. It displays a message when the Variable's value is out of specified bounds.

Class The type of Analytica Object: decision, chance, objective, or index Variable; function; module; library; form; model.

Cloaking See “Definition Hiding.”

Conditional dependency A Chance Variable *a* is conditionally dependent on another Variable *b* if the probability of a value of *a* depends on the value of *b*. If *a* is defined by a probability table, *b* may be an index of its probability table.

Constant A Variable whose value is not probabilistic, and does not depend on other Variables, such as the number of minutes in an hour.

Continuous distribution

A probability distribution defined for a continuous Variable—that is, for a real-valued Variable. Example continuous distributions are beta, normal, and uniform. Compare to “Discrete distribution.”

Continuous Variable

A Variable whose value is a real number—that is, one of an infinite number of possible values. Its range can be bounded (for example, between 0 and 1) or unbounded. Compare to “Discrete Variable.”

Created

The date and time at which the model was first created. This model Attribute is entered automatically, and is not user-modifiable.

Cumulative probability distribution

A representation of a probability distribution that plots the cumulative probability that the actual value of the uncertain Variable *x* will be less than or equal to each possible value of *x*. The cumulative probability distribution is a display option in the Uncertainty View popup menu.

Cyclic dependency

A cyclic dependency occurs when a Variable depends on itself directly or indirectly so that the arrows form a directed circular path. The only cyclic dependencies allowed in Analytica are in Variables using the `Dynamic()` function that contain a time lag on the cycle.

Decision Variable

A Variable that the decision maker can control directly. Decision Variables are represented by rectangular nodes.

Definition

A formula that defines how to compute a Variable’s value. It can be a simple number, a mathematical expression, a list of values, a table, or a probability distribution. In text format, it is limited in length to 32,000 characters.

Definition Hiding

A feature in Analytica Enterprise for protecting your intellectual property when distributing models you have created to others. Definition hiding controls whether the end-user of your model can view the definitions of selected nodes.

Description

Text explaining what the node represents in the real system being modeled. It is limited in length to 32,000 characters.

Deterministic table

A deterministic function that gives the value of a Variable *x* conditional on the values of its input Variables. The input must all be discrete Variables. The table is indexed by each of its inputs, and gives the value of *x* that corresponds to each combination of values of its inputs.

Deterministic value

A Variable's deterministic value, or mid value, is a calculation of the Variable's value assuming all uncertain inputs are fixed at their median values.

Deterministic (determ) Variable
A Variable that is a deterministic function of its inputs. Its definition does not contain a probability distribution. The value of a deterministic Variable can be probabilistic if one or more of its inputs are uncertain. A deterministic Variable is displayed as a double oval. You can also use a general Variable (rounded rectangle) to depict a deterministic Variable.

Determtable
See “Deterministic table.”

Diagram
See “Influence diagram.”

Dimension
An array has one or more dimensions. Each dimension is identified by an index Variable. When an array is shown as a table, the row header (vertical) and column headers(horizontal) give the two dimensions of the table.

Discrete distribution
A probability distribution over a finite number of possible values. Example discrete distributions are Bernoulli and the Probtable function. Compare to “Continuous distribution.”

Discrete Variable
A Variable whose value is one of a finite number of possible values. Examples are the number of days in a month (28, 29, 30, or 31), or a Boolean Variable with possible values **`True`** and **`False`**. A Variable that is defined as a list or list of labels is discrete. Compare to “Continuous Variable.”

Domain
The possible outcomes of a Variable. The Domain has a type as well as value. The possible types are List of labels, List of numbers, or Continuous; the default type is Continuous, except for Variables defined with the **`Choice()`**, **`Probtable()`**, and **`Determtable()`** functions.

Dynamic Variable
A Variable that depends on the system Variable *Time* and is defined by the **`Dynamic()`** function. A dynamic Variable can depend on itself at a previous time period, directly or indirectly, through other dynamic Variables.

Edit Table
A definition that is a table is also called an Edit Table because it can be edited.

Edit tool
The Edit tool is in the shape of the normal mouse pointer cursor. The Edit tool is used to create a new model or to change an existing model. It allows you to move, resize, and edit nodes, and exposes the Arrow tool and node palette.

Excel Graph
The graphing engine of Microsoft Excel®. Users who have Excel installed on their computers can take advantage of Excel Graph to graph results.

Expression — A formula that can contain numbers, Variables, functions, distributions, and operators, such as `0.5`, `a-b`, or `Min(x)`, combined according to the Analytica language syntax. The definition of a Variable must contain an expression.

Expression type — The Expression popup menu, which appears above the definition field, allows you to change the definition of a Variable to one of several different kinds of expressions. Expression types include expression, list (of expressions or numbers), list of labels (text values), table, probability table, and distribution. Any definition, regardless of expression type, can be viewed as an expression.

File Info — The name of the file and folders in which the model was last saved.

Filed library — A library whose contents are saved in a file separate from the model that contains it. A filed library can be shared among several models without making a copy for each model.

Filed module — A module whose contents are saved in a file separate from the model that contains it. A filed module can be shared among several models without making a copy for each model.

Fractile — The median is the 0.5 fractile. More generally, there is probability *p* that the value is less than or equal to the *p* fractile. Quantile is a synonym for fractile. (Fractal is something different!) Compare to “Percentile.”

General Variable — A Variable that can be certain or probabilistic. It is often convenient to define a Variable as a general Variable without worrying about what particular kind of Variable it is. A general Variable is depicted by a rounded rectangle node.

Identifier — A short name for a Variable used in mathematical expressions in definitions. An identifier must start with a letter, have no more than 20 characters, and contain only letters, numbers, and ‘_’ (underscore, used instead of a space). Each identifier in a model must be unique. Compare to “Title.”

Importance analysis — Importance analysis lets you determine how much effect the uncertainty of one or more input Variables has on the uncertainty of an output Variable. Analytica defines importance as the rank order correlation between the sample of output values and the sample for each uncertain input. It is a robust measure of the uncertain contribution because it is insensitive to extreme values and skewed distributions.

Unlike commonly used deterministic measures of sensitivity, this rank order correlation averages over the entire joint probability

distribution. Therefore, it works well even for models where the sensitivity to one input depends strongly on the value of another.

Index — An index of an array identifies a dimension of that array. An index is usually a Variable defined as a list, list of labels, or sequence. An index is often, but not always, a Variable with a node class of Index.

Indexes — Plural of index. Indicates a set of index Variables that define the dimensions of a table (in an Edit Table or value).

Index selection area — The top portion of a Result window, containing a description of the result and other information about the dimensions of the result.

Index Variable — A class of Variable, defined as a list, list of labels, or sequence, that identifies the dimensions of an array—for example, in an Edit Table. An Index Variable is depicted as a parallelogram node. Variables of other classes whose definition or domain consist of list, list of labels, or sequence can also be used to identify the dimensions of an array, and are sometimes referred to as index Variables.

Influence arrow — See “Arrow.”

Influence diagram — An intuitive graphical view of the structure of a model, consisting of nodes and arrows. Influence diagrams provide a clear visual way to express uncertain knowledge about the state of the world, decisions, objectives, and their interrelationships.

Innermost dimension — The dimension of an array that varies most rapidly in the `Table()` function. The innermost dimension is the last index listed in a `Table()` or `Array()` function. Compare to “Outermost dimension.”

Input — A Variable that appears in the definition of the selected Variable. See also “Output.”

Input arrowhead — An arrowhead pointing into a node, indicating that the node has one or more inputs from outside its module. Click on the arrowhead for a popup menu of the input Variables.

Inputs — A list of the Variables or functions on which this Variable or function depends. The inputs are determined by the arrows drawn to and the Variables or functions referred to in this Variable’s or function’s definition or check Attribute. See also “Outputs.”

Intelligent Array Abstraction™ — A powerful key feature of the Analytica Engine that automatically propagates and manages the dimensionality of multidimensional arrays within models.

Key In a results graph, the key shows the value of the key index Variable that corresponds to each curve, indicated by pattern or color.

Kurtosis A measure of the peakedness of a distribution. A distribution with long thin tails has a positive kurtosis. A distribution with short tails and high shoulders, such as the uniform distribution, has a negative kurtosis. A normal distribution has zero kurtosis.

Last Saved The date and time at which the model was last saved. This model Attribute is entered automatically, and is not user-modifiable. If the model is new, this field remains empty until the model is first saved.

Library A model component that typically contains a collection of user-defined functions and/or Variables to be shared.

List A type of expression available in the Expression popup menu consisting of an ordered set of numbers or expressions. A list is often used to define Index and Decision Variables.

List of labels A type of expression available in the Expression popup menu consisting of an ordered set of text items. A list of labels is often used to define Index and Decision Variables.

Matrix A two-dimensional array of numbers with indexes of equal length.

Mean The average of the population, weighted by the probability mass or density for each value. The mean is also called the ***expected value***. The mean is the center of gravity of the probability density function.

Median The value that divides the range of possible values of a quantity into two equally probable parts. Thus, there is 0.5 probability that the uncertain quantity is less than or equal to the median, and 0.5 probability that it is greater than the median.

Mid value The result of evaluating a Variable deterministically, holding probability distributions at their median value. Analytica calculates the mid value of a Variable by using the mid value of each input. The mid value is a measure of central value, computed very quickly compared to uncertainty values. Compare “Probvalue.”

Mode The most probable value of the quantity. The mode is at the highest peak of the probability density function. On the cumulative probability distribution, the mode is at the steepest slope, at the point of inflection.

Model A module, or a hierarchy of linked and/or embedded modules and libraries, on which you work during an Analytica session; the

	main, or root, module at the top of the module hierarchy. Between sessions, a model is stored in an Analytica document file.
Module	A collection of related nodes, typically including Variables, functions, and other modules, organized as a separate Influence Diagram. A module is depicted in a diagram as a node with a thick outline.
Module hierarchy	A model can contain several modules, each one containing details of the model. Each module can contain further modules, containing still more detail. This module hierarchy is organized as a tree with the model at the top. You can view the hierarchical structure in the Outline window.
Multimodal distribution	A probability distribution that has more than one mode.
Node	A shape, such as a rectangle, oval, or hexagon, that represents an Object in an Influence Diagram. Different node shapes are used to represent different types of Variables.
Obfuscated	Saved in a non-human-readable (i.e, encrypted) form. Obfuscation provides a mechanism for protecting intellectual property. Analytica Enterprise users can distribute obfuscated copies of their models to their end-users. In Analytica, obfuscation also has the effect of making settings for definition hiding and/or browse-only mode permanent.
Object	A Variable, function, or module in an Analytica model. Each Object is depicted as a node in an Influence Diagram and is described by a set of attributes. See also “Class,” “Node,” “Attribute,” and “Influence diagram.”
Object Finder	A dialog box used to browse and edit the functions and Variables available in a model.
Object window	A view of the detailed information about a node. The Object window shows the visible attributes, such as a node’s type, identifier, and description.
Objective Variable	A Variable that evaluates the overall desirability of possible outcomes. The objective can be measured as cost, value, or utility. A purpose of most decision models is to find the decision or decisions that optimize the objective—for example, minimizing cost or maximizing expected utility. An objective Variable is represented by a hexagonal node.
OLE Linking	A standard in the Windows operating system for sharing data between applications.

Operator — A symbol, such as a plus sign (+), that represents a computational process or action such as addition or comparison.

Outermost dimension — The dimension of an array that varies least rapidly in the `Table()` function. The outermost dimension is the first index listed in a `Table()` or `Array()` function. Compare to “Innermost dimension.”

Outline window — A view of a model that lists the objects it contains as a hierarchical outline.

Output — A Variable whose definition refers to the selected Variable. See also “Input.”

Output Arrowhead — An arrowhead pointing out of a node, indicating that the node has one or more outputs outside its module. Click on the arrowhead for a popup menu of the output Variables.

Outputs — A list of the Variables or functions that depend on this Variable or function. The outputs are determined by the arrows drawn from this Variable or function and the Variables or functions in whose definition or check Attribute this Variable or function appears. See also “Inputs.”

Parameters — The arguments of a function.

Parametric analysis — See “Behavior analysis.”

Parent diagram — The diagram for the module that contains this Object.

Percentile — The median is the fiftieth percentile (also written as 50%ile). More generally, there is probability *p* that the value is less than or equal to the *pth* percentile. Compare to “Fractile.”

Probabilistic Variable — A Variable that is uncertain, and is described by a probability distribution. A probabilistic Variable is evaluated using simulation; its result is an array of sample values indexed by *Run*.

Probability bands — Probability bands are a way to display the uncertainty in a value by showing percentiles from its distribution—for example, the 5%, 25%, 50%, 75%, and 95% percentiles. On a graph, these often appear as bands around the median (50%) line. Probability bands are also referred to as credible intervals.

Probability density function — A representation of a probability distribution that plots the probability density against the value of the Variable. The probability density at each value of *X* is the relative probability that *X* will be at or near that value. The probability density function can be displayed for continuous, but not discrete Variables. It is a display option in the Uncertainty View popup menu. Compare to “Probability mass function,” which is used with discrete Variables.

Probability distribution — A probability distribution describes the relative likelihood of a Variable having different possible values.

Probability mass function — A probability mass function is a representation of a probability distribution for a discrete Variable as a bar graph, showing the probability that the Variable will take each possible value. The probability mass function can be displayed for discrete, but not continuous Variables. It is a display option in the Uncertainty mode View menu. Compare to “Probability density function,” which is used with continuous Variables.

Probability table — A table for specifying a discrete probability distribution for a Chance Variable. In a probability table, you specify the numerical probability for each value in the domain of *the Variable*. If *the Variable* depends on (that is, is conditioned by) other discrete Variables, each of these conditioning Variables gives an additional dimension to the table, so you can specify the probability distribution conditional on the value of each conditioning Variable.

Probtable — See “Probability table.”

Probvalue — The probabilistic value of a Variable, represented as a random sample of values from the probability distribution for the Variable. The probvalue for a Variable is based on the probvalue for the inputs to the Variable. See also “Probabilistic Variable” and compare to “Mid value.”

Reducing function — A function that operates on an array over one of its indexes. The result of a reducing function has that dimension removed, and hence has one fewer dimension.

Remote Variable — A Variable in another module, not shown in the active diagram. Typically a remote Variable is an input or output of a node in the active diagram.

Result view — A window that shows the value of a Variable as a table or graph.

Sample — An array of values selected at random from the underlying probability distribution for a quantity. Analytica represents uncertainty about a quantity as a sample, and estimates statistics, probability density function, and other representations of a probability distribution from the sample.

Sampling method — A method used to generate a random sample from the probability distributions in a model (for example, Monte Carlo and Latin hypercube).

Scalar — A value that is a single number.

Scatter plot — A graph that plots the samples of two probabilistic Variables against each other.

Self — A keyword used in two different ways:

- Refers to the index of a table that is indexed by itself. `Self` refers to the alternative values of the Variable defined by the table.
- Refers to the Variable itself, as a substitute for the Variable's identifier, in a Check Attribute expression or a `Dynamic` expression.

Sensitivity analysis — A method to identify and compare the effects of various input Variables to a model on a selected output. Example methods for sensitivity analysis are importance analysis and model behavior analysis.

Skewness — A measure of the asymmetry of the distribution. A positively skewed distribution has a thicker upper tail than lower tail, while a negatively skewed distribution has a thicker lower tail than upper tail. A normal distribution has a skewness of zero.

Slice — A slice of an array is an element or subarray selected along a specified dimension. A slice has one less dimension than the array from which it is sliced.

Standard deviation — The square root of the variance. The standard deviation of an uncertainty distribution reflects the amount of spread or dispersion in the distribution.

Suffix — Numbers such as 10K, 123M, or 1.23u are in suffix notation. The suffix letter denotes a power of ten; for example, K, M, and u denote 10^3, 10^6, and 10^{-6}, respectively.

Symmetrical distribution — A distribution, such as a normal distribution, that is symmetrical about its mean.

System function — A function available in the Analytica modeling language. See also "User-defined function."

System Variable — A Variable that is part of the Analytica modeling language, such as *Samplesize* or *Time*.

Table — A two-dimensional view of an array. The array can have more than two dimensions, but only two can be seen at one time. In the Result window, click on the **Table** button to select the table view of an array-valued result.

Tail — The upper and lower tails of a probability distribution contain the extreme high and low quantity, respectively. Typically, the lower

	and upper tails include the lower and upper ten percent of the probability, respectively.
Title	The full name of an Analytica Object. A Variable's or module's title is displayed in its node, in window titles, and in Object lists. It is limited to 255 characters. It can contain any characters, including spaces and punctuation. Compare to “Identifier.”
Uncertain value	See “Probvalue.”
Units	The units of measurement for a Variable. Units are used to annotate tables and graphs; they are not used in any calculation.
User-defined function	A function that the user defines to augment the functions provided as part of the Analytica modeling language.
Value	See “Mid value.”
Variable	An Object that has a value, which may be text, a number, or an array.
Variance	A measure of the uncertainty or dispersion of a distribution. The wider the distribution, the greater its variance.

Alphabetical Index

Symbols

A

Index

B

Index

Index

E

F

Index

G

H

I

Index

M

N

Index

O

P

Index

Q

R

S

Index

T

Index

U

V

W

X

Index

Z

Index

Analytica windows and dialogs

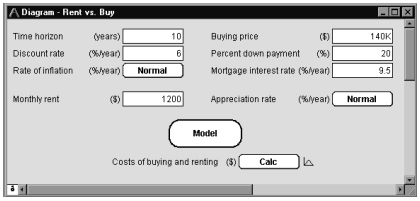


Diagram Window: Inputs and Outputs

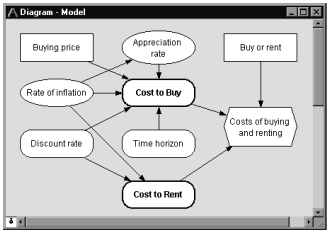
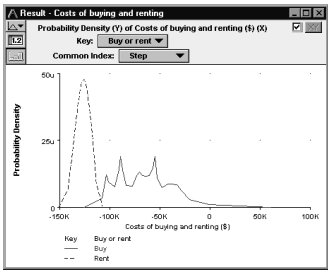
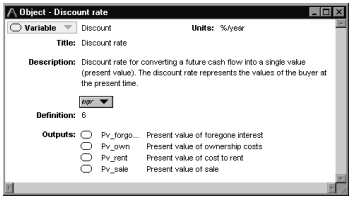


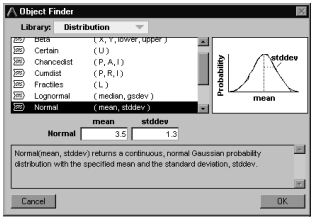
Diagram Window: Influence Diagram



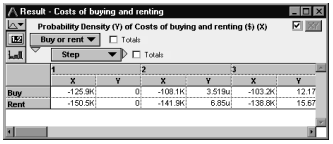
Result Window—Graph View



Object Window



Object Finder



Result Window—Table View

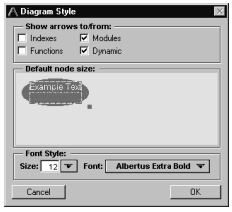
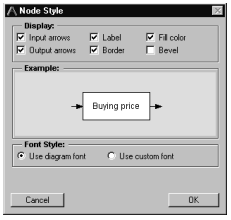
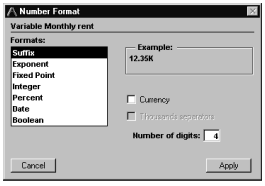


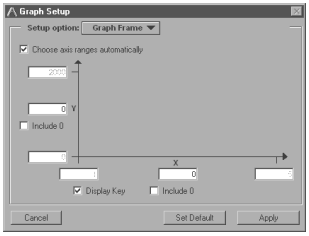
Diagram Style Dialog



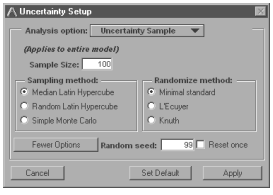
Node Style Dialog



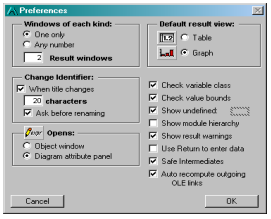
Number Format Dialog



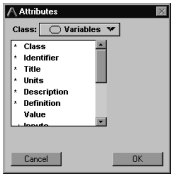
Graph Setup Dialog



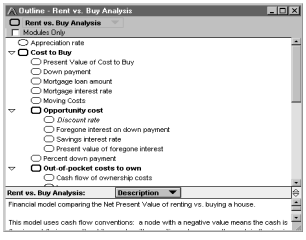
Uncertainty Setup Dialog



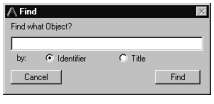
Preferences Dialog



Attributes Dialog



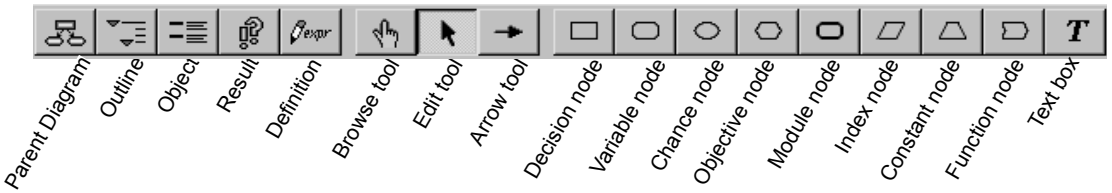
Outline Window



Find Dialog

Analytica Quick Reference

The Tool Bar



The node palette is displayed when either the Edit tool or Arrow tool is selected.

Numerical Formats (Output)

Format	Description	Example
Suffix	the default (see the following table)	12.35K
Exponent	scientific exponential	1.235e04
Fixed Point	fixed decimal point	12345.68
Integer	fixed point with no decimals	12346
Percent	percentage	1234568%
Date	text date	12 Jan 93
Boolean	true or false	True

Numerical Prefixes and Suffixes (Input)

Power of 10	Suffix	Prefix	Power of 10	Suffix	Prefix
3	K	Kilo	-2	%	percent
6	M	Mega or Million	-3	m	milli
9	G	Giga	-6	μ	micro (mu)
12	T	Tera or Trillion	-9	n	nano
15	Q	Quad	-12	p	pico
			-15	f	femto

Analytica Note: *If integer or fixed point is selected, numbers larger than 10^9 display in exponent format.*