

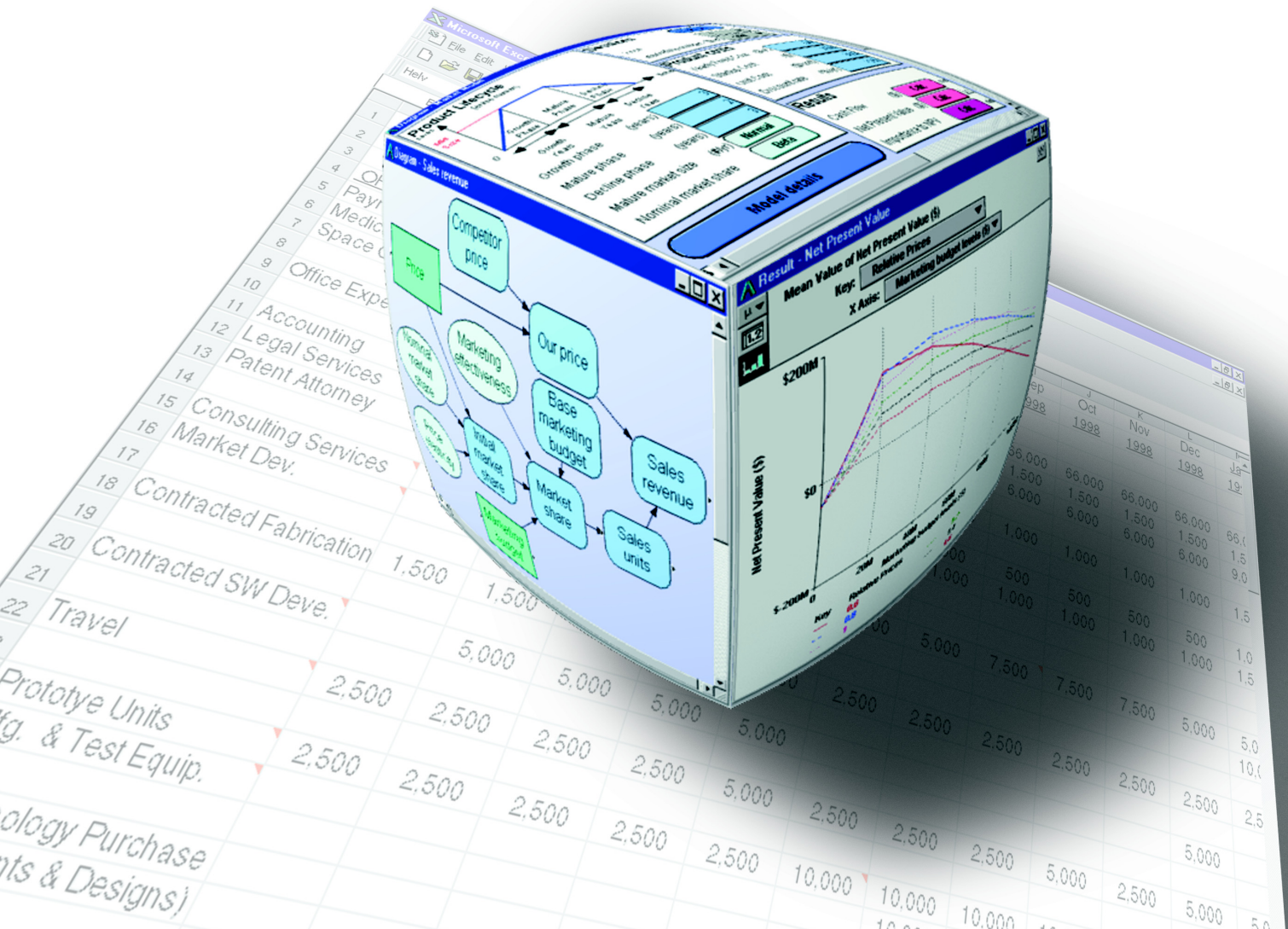


analytica[®]

Beyond the Spreadsheet

User Guide

Release 4.0



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



Copyright Notice

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

This document is © 1993-2007 Lumina Decision Systems, Inc. All rights reserved.

The software program described in this document, Analytica, includes code that is copyrighted:

© 1982-1991 Carnegie Mellon University

© 1992-2007 Lumina Decision Systems, Inc., all rights reserved.

Analytica was written using MacApp®: © 1985-1996 Apple Computer, Inc.

Analytica incorporates Mac2Win technology, © 1997 Altura Software, Inc.

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

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

Acknowledgements

This *Analytica User Guide* was written and edited by Lonnie Chrisman, Max Henrion, and Richard Morgan, with contributions from Brian Arnold, Fred Brunton, Adrienne Esztergar, Jason Harlan, Lynda Korsan, Randa Mulford, Rich Sonnenblick, Brian Sterling, and Eric Wainwright.

Contents

Acknowledgements	ii
About Analytica	1
Welcome!	2
If you don't read manuals.....	2
Hardware and software requirements	2
Installation and license codes	3
Editions of Analytica.....	5
Help menu and electronic documentation.....	7
Normally, usually, and defaults	9
Typographic conventions in this guide.....	9
User guide examples folder	10
What's new in Analytica 4.0?	10
Chapter 1: Examining a Model	15
To open or exit a model	16
Diagram window	17
Classes of variables and other objects	18
Selecting nodes	19
The toolbar	20
Browsing with input and output nodes	21
The Object window	22
The Attribute panel	23
Showing values in the Object window.....	24
Printing.....	25
Chapter 2: Result Tables and Graphs	27
The result window	28
Viewing a result as a table	30
Viewing a result as a graph.....	31
Uncertainty views.....	32
Comparing results	36
Chapter 3: Analyzing Model Behavior	39
Varying input parameters.....	40
Analyzing model behavior results	42
Chapter 4: Creating and Editing a Model	47
Creating and saving a model	48
Creating and editing nodes	48
Drawing arrows	51
How to draw arrows between different modules	53
Alias nodes	55
To edit an attribute	57
To change the class of an object	58
Preferences dialog	59

Chapter 5: Building Effective Models	63
Creating a model	64
Testing and debugging a model	67
Expanding your model	69
Chapter 6: Creating Lucid Influence Diagrams	71
Guidelines for creating lucid and elegant diagrams	73
Arranging nodes to make clear diagrams	75
Organizing a module hierarchy	79
Color in influence diagrams	80
Diagram Style dialog.....	81
Node Style dialog.....	82
Taking screenshots of diagrams	83
Chapter 7: Formatting Numbers, Tables, and Graphs	85
Number formats	86
Date formats	88
Multiple formats in one table.....	90
Graphing roles	91
Graph setup dialog box.....	94
Graph templates	102
XY comparison	105
Chapter 8: Creating and Editing Definitions	115
Creating or editing a definition	116
The Expression popup menu.....	120
Object Finder dialog.....	121
Using a function or variable from the Definition menu	123
Automatic checking for valid values.....	124
Chapter 9: Creating Inputs and Outputs	127
Using input nodes	128
Creating a choice menu	129
Using output nodes	131
Input and output nodes and their original variables	132
Using form modules.....	132
Adding icons to nodes	133
Graphics, frames, and text in a diagram.....	135
Models in XML file format	135
Hyperlinks in model documentation	137
Chapter 10: Using Expressions	139
Numbers	140
Text values	142
Boolean or logical values	142

Operators	142
IF a THEN b ELSE c.....	145
Functions	145
Math functions	146
Numbers and text	147
Datatype functions	148
Chapter 11: Arrays and Indexes	151
Introduction to arrays	153
Operations on arrays	155
IF a THEN b ELSE c with arrays	159
Creating an index.....	161
Editing a list	165
Functions that create indexes.....	165
Creating an array with an edit table	168
Editing a table	171
Choice menus in an edit table	173
Shortcuts to navigate and edit a table	174
Chapter 12: More Array Functions	177
Intelligent Arrays™	178
Functions that create arrays	181
Array-reducing functions.....	185
Transforming functions	190
Selecting, slicing, and subscripting arrays.....	193
Converting between multiD and relational tables	197
Interpolation functions.....	199
Other array functions	201
SubTable	203
Matrix functions.....	203
Chapter 13: Other Functions	209
Text functions	210
Date functions.....	212
Advanced math functions	213
Financial functions	214
Financial library functions	219
Advanced probability functions.....	222
Chapter 14: Expressing Uncertainty	225
Choosing an appropriate distribution	226
Defining a variable as a distribution.....	229
Including a distribution in a definition.....	231
Probabilistic calculation	231
Uncertainty Setup dialog box.....	232

Chapter 15: Probability Distributions	239
Probability distributions.....	240
Parametric discrete distributions.....	241
Probability density and mass graphs.....	243
The domain attribute and discrete variables.....	245
Custom discrete probabilities.....	246
Parametric continuous distributions.....	252
Custom continuous distributions.....	261
Special probabilistic functions.....	263
Multivariate distributions.....	265
Importance weighting.....	270
Chapter 16: Statistics, Sensitivity, and Uncertainty Analysis	273
Statistical functions.....	274
Weighted statistics and w parameter.....	282
Importance analysis.....	282
Sensitivity analysis functions.....	284
Tornado charts.....	287
X-Y plots.....	289
Scatter plots.....	291
Regression analysis.....	292
Uncertainty in regression results.....	294
Chapter 17: Dynamic Simulation	297
The Time index.....	298
Using the Dynamic() function.....	298
More about the Time index.....	300
Initial values for Dynamic.....	303
Using arrays in Dynamic().....	304
Dependencies with Dynamic.....	305
Uncertainty and Dynamic.....	306
Chapter 18: Importing, Exporting, and OLE Linking Data .	309
Copying and pasting.....	310
Using OLE to link results to other applications.....	311
Linking data from other applications into Analytica.....	314
Importing and exporting.....	318
Printing to a file.....	319
Edit table data import/export format.....	319
Chapter 19: Working with Large Models	323
Show module hierarchy preference.....	324
The Outline window.....	325
Finding variables.....	326
Managing attributes.....	327

Invalid variables	329
Using filed modules and libraries.....	330
Adding a module or library.....	332
Combining models into an integrated model	333
Managing windows	335
Optimization and speed-up.....	336
Chapter 20: Building Functions and Libraries	337
Example function	338
Using a function.....	339
Creating a function.....	339
Attributes of a function	340
Parameter qualifiers.....	340
Libraries	345
Chapter 21: Procedural Programming	347
An example of procedural programming.....	348
Summary of programming constructs.....	350
Begin-End, (), and ';' for grouping expressions.....	350
Declaring local variables and assigning to them.....	351
For and While loops and recursion	353
Local indexes.....	358
Ensuring array abstraction.....	359
References and data structures.....	364
Miscellaneous functions.....	368
Chapter 22: Analytica Enterprise	373
Accessing databases.....	374
Database functions	381
Reading and writing text files.....	383
Making a browse-only model and hiding definitions	384
Huge Arrays.....	387
Creating buttons and scripts	387
Performance Profiler.....	391
RunConsoleProcess(program, <i>cmdline</i> , <i>stdin</i> , <i>block</i>).....	393
Appendix A: Selecting the Sample Size	398
Appendix B: Menus	401
File menu	401
Edit menu.....	402
Object menu	403
Definition menu.....	403
System Variables submenu	405
Result menu.....	405

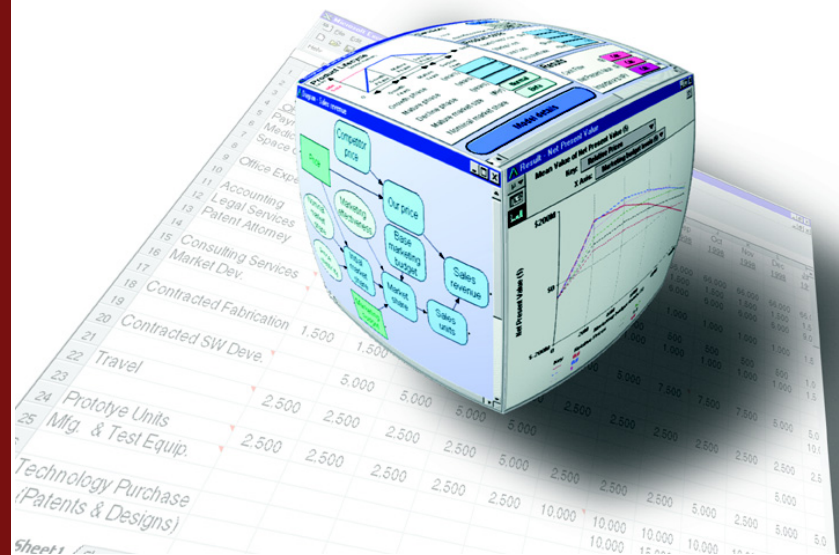
Diagram menu	406
Align submenu	407
Make Same Size submenu	407
Space evenly submenu	407
Window menu	408
Help menu	408
Right mouse button menus	409
Appendix C: Analytica Specifications	410
Memory usage	410
Appendix D: Identifiers Already Used	412
Appendix E: Error Message Types	413
Appendix F: Forward and Backward Compatibility	416
Appendix G: Bibliography	418
Appendix H: Function List	419
Analytica windows and dialogs	449

Introduction

About Analytica

This introduction explains:

- How to use this manual
- How to install Analytica
- The online help system
- How to access Analytica example models
- What's new in Analytica release 4.0.
- Typographic conventions used in this guide



Welcome!

This *User Guide* describes how to use Analytica 4.0. If you are new to Analytica, we invite you to start with the *Analytica Tutorial* to learn the essentials. Most people find they can work through the *Tutorial* quite rapidly. You may then want to read a few sections of the *User Guide* listed in the next section to learn more key concepts. You may consult the rest of this guide as a reference when you need more depth. For still more, visit Anawiki (the Analytica Wiki online at <http://www.lumina.com/wiki>), including Tips, Libraries, and Reference.

If you can't find what you want, or have comments on our documents or software, please email us at Lumina at support@lumina.com. We are always glad to hear from Analytica users.

Click cross references If you are reading this guide as a PDF document on your computer, you can click the page number in any cross reference to jump to that page. To return to the previous location, use Acrobat's **Go To Previous View** feature by pressing *Alt+left-arrow* (may vary depending on your version of Acrobat).

If you don't read manuals

Experienced modelers find most Analytica features intuitive. But, it's helpful to get a good grasp of some key concepts so you can get up to speed rapidly. Here are a few chapters that you may find especially helpful to review:

**Chapter 5:
Building effective
models**

offers 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 not specific to Analytica, but we designed Analytica to make them especially easy to follow.

**Chapter 6: Creating
lucid diagrams**

gives tips on how to create influence diagrams that are truly lucid and elegant — and how to avoid incomprehensible spaghetti.

**Chapter 11:
Arrays and Indexes**

explains Analytica's Intelligent Arrays™. After you grasp the essentials, they let you build complex multidimensional models with surprising ease. But, you may find they take a little getting used to, particularly if you have spent a lot of time with spreadsheets or programming with arrays. We recommend that even — perhaps *especially* — experienced modelers review this chapter.

**Chapter 13:
Expressing
uncertainty**

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.

**Chapter 20:
Procedural
programming**

With Analytica, you can create large and sophisticated models *without* procedural programming. But, if you really want to write complex procedural functions, read this chapter to understand Analytica as a programming language.

Hardware and software requirements

To use Analytica, you need the following quite modest minimum configuration:

- 486-66 MHz (Pentium 500 MHz+ recommended)
- 20 MB disk space
- 256 MB RAM (2 GB recommended for large models)
- 8-bit color display
- Windows 98, 2000, NT 4, ME, XP, or Vista

It helps to have a faster CPU, and, especially, more RAM for large models. Analytica will benefit from up to 3 GB RAM if you have it. It is also handy to have a large screen, or even multiple screens, when working with a large model.

Installation and license codes

After downloading the Analytica 4.0 installer from www.lumina.com, or inserting the Analytica CD-ROM into your CD or DVD drive, just double-click the installer to start installation. It will install onto your hard drive the executable software, all documentation as Adobe PDF files, plus a range of Analytica libraries and example models. If you have installed an earlier release of Analytica, such as 2.0 or 3.1, the installer will leave it there, so you can run either version.

The setup program will ask you to confirm the directory name in which to install Analytica, by default, C:\Program Files\Analytica 4.0. Most users can accept the defaults.

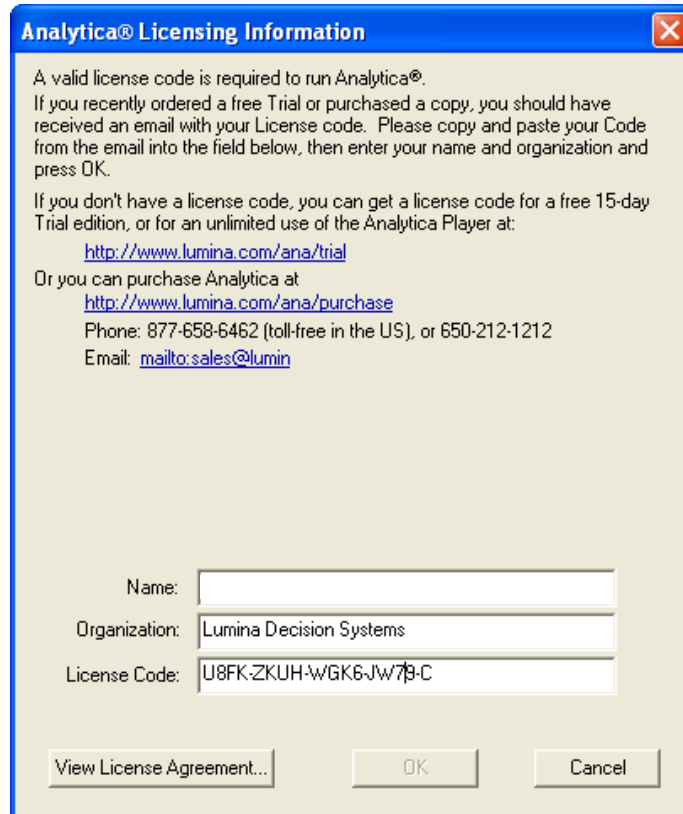
License codes You need a license code to activate the software. Lumina will email you a license code when you download a Player or Trial edition, or when you purchase a copy. If someone else purchased Analytica for you, you may need to ask that person to forward you the email with your license code.

During installation, Analytica will prompt you for a license code. You can copy and paste the code from the email into the field, or just retype it. The license code activates the specified edition of Analytica, e.g., Player, Trial, Professional, Enterprise, or Optimizer.

Stale license codes Each license code goes *stale* a few days after it is generated. If yours is stale — perhaps, because you didn't install Analytica right away, or, later, if you want to move Analytica onto another computer — fear not! Click 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. This mechanism is designed to prevent unauthorized use of old license codes. Authorized users can always get a fresh license code.

Expiration dates Some license codes — notably, for a Trial or an edition licensed per year — have a limited life, after which they *expire*. After 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. *Expiration is not the same as going stale*. To reactivate Analytica after expiration, you may need to purchase a copy.

When you purchase a license or upgrade to another edition You don't need to download and reinstall Analytica again when you purchase a license after testing the free trial, or if you want to upgrade from, say, the Professional to Enterprise edition. Just select **Update License** from the **Help** menu in Analytica and enter your new license code into the **Licensing Information** dialog:



Analytica Decision Engine (ADE) is a different application from Analytica, and requires a new installation, even if you already have another edition of Analytica installed.

To upgrade to a minor or patch release

When you upgrade a licensed copy with a patch or minor release (e.g., 4.0 to 4.0.1), simply run the installer. It will replace the older release and reuse your existing license code.

To upgrade to a major release

You can install Analytica 4.0 and retain an earlier major release, such as Analytica 3.1, on your computer. You will need a new license code for the new release.

To uninstall Analytica

After confirming that Analytica 4.0 is working, you will usually want to uninstall the earlier release. To uninstall Analytica 3.1 — or any release:

1. From the Windows **Start** menu, open the **Control Panel**.
2. Click **Add or remove programs**.
3. Find **Analytica 3.1** (or whichever release you want to remove) and click **Change/Remove** button to start the Wizard.
4. In the **InstallShield** Wizard dialog, select the **Remove** radio button, and click **Next**.

Editions of Analytica

Analytica is available in these editions. See the next page for a list of key features by edition:

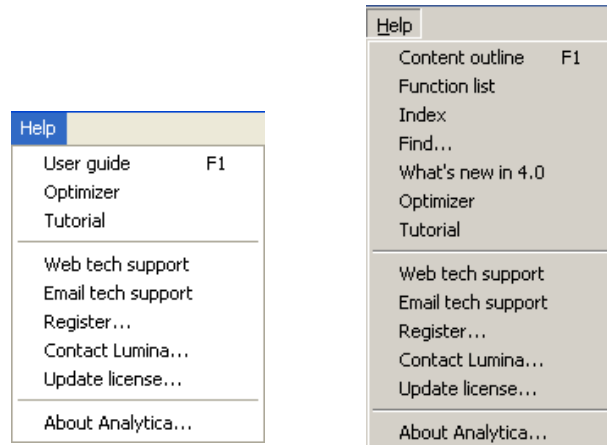
- Player** Lets you review and run Analytica models without having to purchase a license. With the Player edition, you can change designated inputs, run the model, view results, and examine selected model diagrams and variables. It does not let you create new models, make changes other than to selected inputs, or save models.
- Professional** Provides most features, including the ability to create, edit, and save models.
- Trial** A free edition of Analytica that provides the full functionality of Analytica Professional for a limited time, usually 15 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.
- Power Player** Like the Player, it lets you review models, change inputs, and view results, and does not let you create or edit models. Unlike the Player, it does let you save models with changed inputs. It also supports models that use Enterprise features, including database access, Huge Arrays, and the Profiler. See Chapter 22, “Analytica Enterprise” for details.
- Lite** Available to qualified educational users only for teaching and research. It has the features of the Professional edition, except: No OLE linking, Outline Window, Advanced function libraries, nor ability to create input and output nodes and forms.
- Enterprise** Offers all the features of Analytica Professional, plus Huge Arrays, reading and writing databases, profiling for analysis of computational effort by variable, and obfuscation (encryption) of sensitive model elements. See Chapter 22, “Analytica Enterprise” for details.
- Optimizer** Offers all the features of Analytica Enterprise, plus the Optimizer Library that provides powerful solver and optimization methods, including linear programming (LP), quadratic programming, and nonlinear programming (NLP). Optimizer is available as an extension to Analytica Enterprise, Power Player, and ADE. See the *Analytica Optimizer Guide* for details.
- The Analytica Decision Engine (ADE)** ADE runs Analytica models on a server computer. It provides an Application Programming Interface (API) 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. You will need Analytica Enterprise as the development tool to create models to run with ADE. The ADE Kit includes a license for Analytica Enterprise in addition to ADE.

Compare Analytica features by edition

Features	Editions of Analytica						
	Player	Power Player	Trial	Lite	Professional	Enterprise	ADE
Open models, change inputs, & view results	✓	✓	✓	✓	✓	✓	✓
Save model with changed inputs		✓	✓	✓	✓	✓	✓
Create and edit models			✓	✓	✓	✓	✓
No marking of printout		✓		✓	✓	✓	✓
Hierarchical influence diagrams	✓	✓	✓	✓	✓	✓	
Monte Carlo uncertainty analysis	✓	✓	✓	✓	✓	✓	✓
Intelligent Arrays	✓	✓	✓	✓	✓	✓	✓
Procedural programming			✓	✓	✓	✓	✓
OLE linking with Excel		✓	✓		✓	✓	
Outline Window	✓	✓	✓		✓	✓	
Create input and output controls and forms			✓		✓	✓	
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						✓	✓
Huge Arrays™ — dimension up to 100 million		✓				✓	✓
ODBC database access		✓				✓	✓
Time and Memory Profiling		✓				✓	✓
Optimizer available		✓				✓	✓
Application Programming Interface. See <i>ADE User Guide</i>							✓

Help menu and electronic documentation

Select the **Help** menu from the menu bar:



Tip Most users see the left-hand version of the menu starting with **User Guide**. The right-hand version appears if you have Adobe Acrobat Standard or Professional installed, which enable direct links into sections of a PDF document.

- Content outline F1** Opens the *User Guide* showing chapters, sections, and subsections as an expandable outline, using bookmarks. Press the function key *F1* as a shortcut.
- Function list** Opens a page listing all functions, operators, and other constructs, classified by type. Click a name to jump to an explanation of how to use it. This is a fast way to find a function if you don't know its name.
- Index** Opens the *User Guide* to its alphabetized index. Select the first letter of the term from the bookmark outline, and click an entry to jump to its explanation.
- Find** Opens the **Find** dialog box in Adobe Acrobat so you can search for a term.
- What's new in 4.0?** Opens "What's new in Analytica 4.0?" in the *User Guide*.

User Guide F1 Opens this *Analytica User Guide* as a PDF document in Adobe Reader. Press the function key *F1* as a shortcut (see "Online help and electronic documentation" on page 8).

Optimizer Opens the *Optimizer Guide* (if you have Analytica Optimizer).

Tutorial Opens the *Analytica Tutorial* as a PDF document in Adobe Reader.

-
- Web tech support** Opens Lumina's Analytica tech support web page in your default Web browser, with support information and links to frequently asked questions.
- Email tech support** Starts an email message to send to Lumina tech support using your default email application.
- Register** Opens a web page where you can register your copy of Analytica, and copies your license code into the required field.

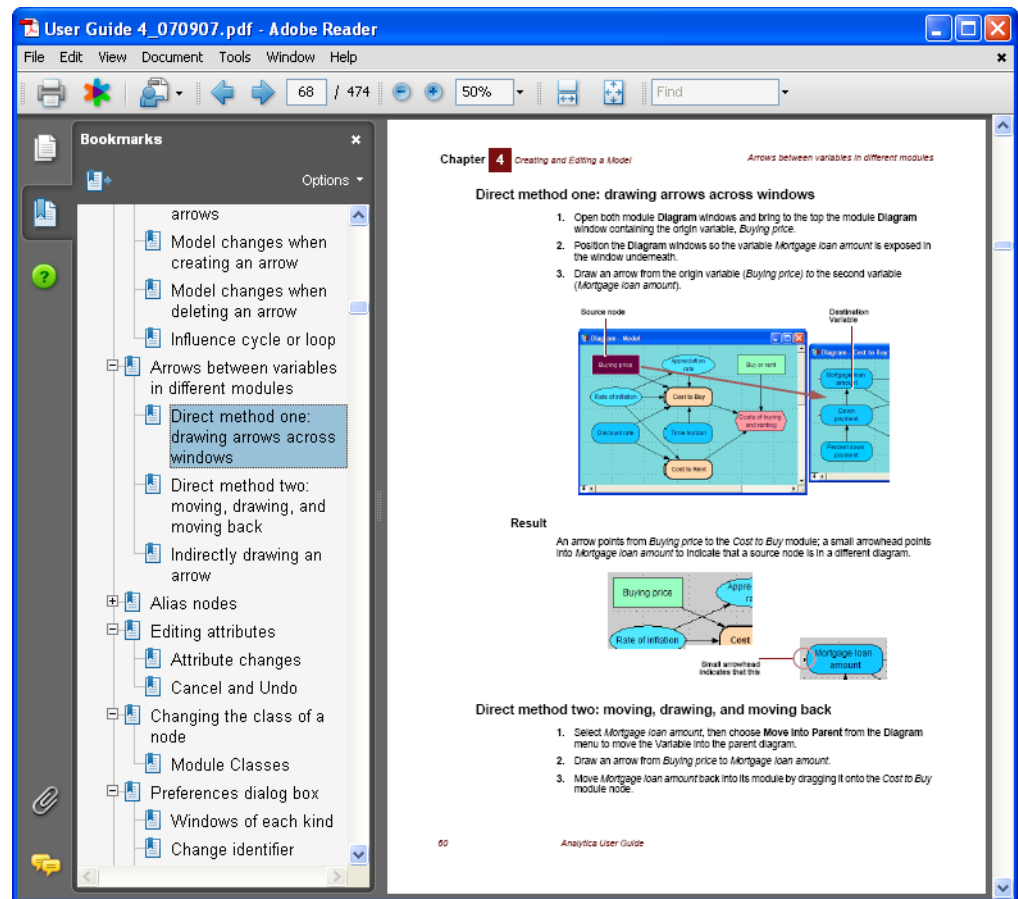
- Contact Lumina** Opens a dialog box with Lumina contact information: web links, phone numbers, email, and physical mailing address.
- Update license** Opens the **Licensing Information** dialog box so you can review your or enter a new License Code or enter a new code to upgrade your copy of Analytica.

About Analytica Opens the startup splash screen, mentioning the Analytica Edition, release number, and the name of the person to whom it is licensed.

Online help and electronic documentation

You can open the *Tutorial*, *User Guide* and *Optimizer Guide* (when available) from the **Help** menu, or press the *F1* key to open the *User Guide*.

You can read and search these PDF documents using the Adobe® Reader available free from <http://www.adobe.com>. Some additional features are available if you purchase Adobe Acrobat Standard or Professional.



The expandable outline Click a section title to view that section. Click or icons to expand and contract chapters and sections of the outline.

Appendix H: Function list If you can't remember the name of a function, go to this page. It lists functions and system variables by functional groups. From here, click a function name to jump to its full description.

Alphabetical index If the search box finds too many occurrences of a term, try the *Alphabetical Index* in the bookmarks: It usually links to the best explanation for each term.

Normally, usually, and defaults

Sometimes this guide says “normally it does this” or “usually it does that.” This isn’t because Analytica is unpredictable, or because we’re just addicted to uncertainty. It’s because Analytica has a lot of preference and style options, and it’s often simpler to say “normally” or “usually” when we mean “with the standard defaults.”

Typographic conventions in this guide

Example	Meaning
<i>behavior analysis</i>	Key terms when introduced. Most of these terms are included in the Glossary
Diagram	Menus and menu commands, window names, panel names, dialog box names, function parameters
Sequence()	Name of a variable or function in Analytica
Price - DownPmt	Expressions, definitions, example code
10^7 → 10M	In example code, this means that the variable or expression before the “→” generates the result after it
<i>Enter, Control+a</i>	A key or key-combination on the keyboard. A letter, such as “a”, may be lower- or uppercase

Code examples This guide includes snippets of code to illustrate features, for example:

```
Index N := [1, 2, 3, 4, 5]
Variable Squares := N^2
Sum(Squares, N) → 55
```

This code says that there are two objects, an Index **I** and a variable **Squares**. You would create these objects in a **Diagram** window by dragging from the node toolbar into the diagram (see “Creating and editing nodes”). You would enter the expressions, **[1, 2, 3, 4, 5]** and **N^2** into their definitions (see “Creating or editing a definition”). You would *not* enter the assignment “:=”. The last line says that the expression **Sum(Squares, I)** evaluates to the result **55** after the “→”. You might include that expression in the definition of third variable.

Array examples We use these typographic conventions to show Analytica arrays:

- An index or list and its values

N:

1	2	3	4	5
---	---	---	---	---

- A one-dimensional array, **Squares**:

N ►

	1	2	3	4	5
	1	4	9	16	25

- 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

User guide examples folder

The **Examples** folder distributed with Analytica includes **User Guide Examples** as a subfolder. It contains Analytica models used in Chapters 9, 10, 11, 12, 14, 15, 16, and 17. Open these models to see the examples in more detail.

See Chapter 8 of the *Tutorial* for a summary of the models in the **Examples** folder.

What's new in Analytica 4.0?

These are highlights of new and improved features in release 4.0.

User interface

Graphs and charts We completely rewrote the graphing and charting engine, adding a wide range of new styles and options. The **Graph Setup** dialog now has six tabs:

- **Chart type** tab includes stacked bars, filled areas with transparency, using symbol shape and size to indicate extra dimensions, 3D effects on bar charts, cylinders or boxes, and changing line width. It lets you swap horizontal (X) and vertical (Y) axes, e.g., to create horizontal bars for tornado diagrams.
- **Axis** tab offers log scales, reversed scales, and categorical scale. You can save axis settings as defaults associated with corresponding index variables. Graphing is much smarter in choosing which dates to display along an axis — by week, month, quarter, or year.
- **Style** tab lets you change colors of grid and frame — in addition to style of the grid, frame, tick marks, and key.

- **Text** tab lets you change the font type, size, style, and color for titles and labels. You can also rotate labels for axis tick marks to prevent overlaps, say for long text values.
- **Background** tab now lets you set a color or color gradient for the background of the entire chart, plot area, or key.
- **Preview** tab lets you look at the effects of the options you have selected before you decide to accept them. You can apply new graph settings to the current graph, or as defaults for all graphs.

See “Graph setup dialog box” for more information.

Graph style templates let you apply and reuse a collection of graph settings, for a consistent style for a model or your entire organization. See “Graph templates”

Graphing associates settings with the view so that it changes appropriately when you pivot or change the uncertainty view.

XY comparison now lets you plot one slice against another slice of an array variable over the Comparison index, as well as one variable against another. See “XY comparison”

Tables In graphs or tables, you can **reorder slicer** indexes (any graph indexes not shown on horizontal axis or key) simply by dragging them.

You can create smarter end-user interfaces by putting **dropdown** menus in cells of an edit table, using **Choice()** to let end users select from a list of options. See “Choice menus in an edit table”. When viewing a table, using **Find** from the **Object** menu (*Control+F*) lets you search for selected text.

The new **SubTable** function lets you define a variable as a subset of another edit table — any edit to a subtable makes the same change to its parent table, and vice versa. See “SubTable”

Smart **table splicing** controls how an edit table changes if its indexes change, e.g., editing a label or adding an item or index. You can specify default values for new cells created by expanding and index.

Number, currency, dates, and languages

Analytica is less US-centric: Number formats offer multiple currency symbols and flexible date formats, with format and language of days and months depending on Windows regional settings. You can paste text containing accents and non-English characters (Ascii>127) into **object** attributes and diagram nodes. The date functions **DateAdd**, **DatePart**, and **Today** add flexibility for computing dates. See “Number formats”

Scroll wheel and keyboard shortcuts

The **scroll wheel** on your mouse scrolls windows (diagrams, tables, and objects), vertically, or horizontally when you press *Control*. Dozens of **new keyboard shortcuts** let you navigate and select cells and regions from tables (like Microsoft Excel). When editing a diagram, shortcuts *Control+1*, *Control+2*, etc., add a new decision node, variable node, etc. *Control+e* now opens the script of a button, just like it opens the definition for a variable or function.

Influence diagrams

To make diagrams neater, use the new **Align**, **Make same size**, and **Space evenly** options from the **Diagram** menu (see “Arranging nodes to make clear diagrams”). You can now add web links to a diagram as URLs in a text node. An optional red flag in node shows which objects have descriptions (see “Preferences dialog”).

Here’s an alternative to drawing arrows: When you’re editing a definition in the **Attribute** panel below a **Diagram** window, *Alt-click* another node in the diagram to insert its identifier into the definition.

The Application

- Auto save** Analytica writes each change to an **auto save file**, so you won't lose any work after a software or hardware crash. Next time you start the model, it asks if you want to use the backup or revert to the previously saved version.
- CPU sharing** It **shares CPU** nicely with other applications, and doesn't hog the CPU when it is active.
- Multiple screens** Analytica now supports editing diagrams across **multiple screens** for a larger desktop.

Probability distributions and statistical functions

- Discrete or continuous** When graphing a probability distribution, it is smarter about displaying a probability mass function for a discrete variable or density function for a continuous variable. If needed, you can override this, by specifying **Continuous** or **Discrete** in the **Domain** attribute, or checking **Categorical** in the **Axis scale** tab in **Graph set up**. See "The domain attribute and discrete variables".
- New functions** **Random()** generates single random sample from any distribution. **Shuffle(x, i)** randomly shuffles an array. **Pdf(x)** and **Cdf(x)** return the estimated probability density or cumulative probability functions as arrays. The system variable **IsSampleMode** returns true in prob mode, false in mid mode, so you can tell the evaluation mode within a function. See "Random(expr)", "Shuffle(a, i)", and "PDF(X) and CDF(X)".
- Over parameter** You can create an array of independent probability distributions over one or more indexes by adding optional **Over** parameter to a univariate probability distribution, e.g., **Normal(0, 1, Over: I, J)**. See "Over indexes as parameters to probability distributions".
- Extended functions** **Lognormal** uses *mean* and *stddev* (standard deviation) as an alternative to *median* and *gsdev* (geometric standard deviation). **Truncate(x, min, max)** accepts min and/or max threshold parameters and preserves sample ordering, and hence rank correlations. **Uniform(min, max, integer)** adds the optional parameter **integer** to specify that values be integers in the range. **CumDist(p, r, i, smooth)** adds an optional **Smooth** parameter to control interpolation. See "Parametric continuous distributions" and "Truncate(u, min, max)".
- Uncertain parameters** Many distribution functions are much faster, especially when their parameters are uncertain (hierarchical distributions). **Gamma**, **Binomial**, **Gammallnv** are more accurate for extremely large or small parameter values. See "Gammal(x, a, b)", "Binomial(n, p)", and "Gammallnv(y, a, b)".
- Multivariate Distributions library** New distributions include **MultiUniform** and **UniformSpherical**, generalized **Dist_reshape**, functions for creating time series with serial correlations, and uncertainty about regression coefficients. See "MultiUniform(cm, i, j, lb, ub)", "Dist_reshape(x, newdist)", and "UniformSpherical(i, r)".
- Distribution Variations library** New distributions include **Smooth_fractile**, **Warp_dist**, **Erlang**, **Pareto**, **Rayleigh**, **Lorenzian**, **NegBinomial**, **InverseGaussian**, **Wald**.
- Running index for statistics** By default, the running index defining which dimension statistical functions operate over is **Run**, the index over random samples. You may specify a different running index as the last parameter to any statistical function if you want something other than **Run**, e.g., **Variance(X, I)** computes the variance over index I, even if X is not uncertain. This renders obsolete the **Data Statistics Library.ana**, previously included with Analytica.
- Importance weighting** **Importance weighting** is a powerful enhancement to Monte Carlo simulation that lets you get more information from fewer samples; it is especially valuable for risky situations with a small probability of an extremely good or bad outcome. Instead of treating

all samples as equally likely, you can set **SampleWeighting** to generate more samples in the most important areas. Graphs of probability distributions and statistical functions downweight sample values with **SampleWeighting** so that their results are unbiased. You can modify **SampleWeighting** interactively to reflect different input distributions and so rapidly see the effects the effects on results without having to rerun the simulation. In the default mode, it uses equal weights, as before, so you don't have to worry about importance sampling unless you want to use it. See "Importance weighting".

Weights for statistics By default, statistics functions use **SampleWeighting** when you are using importance sampling. You may also provide an optional parameter **W** to a statistical function to specify a nondefault set of weights. For example, **Mean(X, W: X > 0)** gives the mean of **X** conditional on **X** being positive.

New functions and language extensions

List of variables If you define a variable as a list of variables, e.g., **x := [A, B]**, it creates the list variables as the index value of **x**. This is very convenient for comparing several variables. In a table view, it usually shows the title of each variable in the index. If you double-click a variable title, it opens its **Object** window. You can add another variable **c** to the list simply by drawing an arrow from **c** to **x**, or remove it by redrawing the arrow.

IndexVals If you define **x** as a list of variables, as above, it saves the list of variables as its index in its **IndexVals** attribute. You can get these with the **IndexVals(X)** function. If you pass **x** to a function as an **Index** parameter, it uses **IndexVals**.

FOR iteration index In **FOR j := x DO e**, **x** can now be any expression that evaluates to an array. It evaluates **e** with **j** set successively to each cell (atom) of **x**. The value of the FOR expression is an array with the same index(es) as **x**.

You can now subscript an expression, as in **(A+B)[I=x]**.

Position operator @ **@J** returns the *position* (an integer from 1 to n) of each element of index **J**. **X[@J = 2]** is equivalent to **Slice(A, J, 2)**. **PositionInIndex(a, u, i)** gives the position **n** in index **i** for which **a[i=n] = u**. See "@: Index Position Operator".

Slice assignment **x[I=y] := b**, now lets you assign to a cell or slice of a local variable **x**, allowing you to write some algorithms much more efficiently. See "Assigning to a slice of a local variable".

Argmin and Argmax The new **Argmin(x, i)** and existing **ArgMax(x, i)** can both now work over multiple indexes, and return the value or position of the indexes containing the minimum or maximum value. See "Argmin(a, i)" and "Argmax(a, i)".

Trig functions We have added the inverse trigonometric functions **ArcCos**, **ArcSin**, **ArcTan**, and hyperbolic functions **Cosh**, **Sinh**, and **Tanh**. They use or return degrees, not radians. See "Math functions" and "Advanced math functions".

Rank Lets you specify mid, lower, or upper rank in the event of a tie.

RunConsoleProcess Lets you run another application from Analytica. It can pass data as function parameters or via data files. It can run a process concurrently with Analytica or wait for its result to be computed. See "RunConsoleProcess(program, cmdline, stdin, block)".

System functions **GetRegistryValue()** returns selected values from the computer registry, such as the default directory for model or data files. **ShowPdfFile()** shows an Adobe PDF file, for example, to open PDF documentation for a model. **AnalyticaLicenseInfo** returns information about the license, such as its edition, beta status, expiration date, or user ID.

- TypeOf(X)** Returns the type of each atom in X as a text value, including “Number”, “Text”, “Reference”, or “Null”. If X is a handle, it returns the class of the object pointed to by X.
- Handles** A pointer to an object, such as a variable or module. The **Inputs**, **Outputs**, or **Contains** attributes create a list of handles to objects. With handles, you can write functions that navigate around a model, e.g., to get a list of the inputs or all ancestors of a variable. The new function **Handle(X)** gives a handle to X instead of its value. **HandleFromIdentifier(T)**, as you might expect, gives a handle if T is the text identifier of an object. **IndexesOf(A)** returns a list of handles of the indexes of array A.
- Optional and repeated parameters** The qualifier **Optional** in the parameters of a function specifies that the parameter is optional. You can also supply a default for when the parameter is omitted. The repeat qualifier “...” lets you define a function that takes one or more parameters of the given type.
- Multiply by zero** **0*NaN** and **0*INF** now give a warning and return **NaN**, consistent with the IEEE 754 and SANE arithmetic standards. Earlier releases simply returned 0.

Analytica Enterprise Edition

These features are available in the Enterprise edition, and may be used from the Power Player edition:

- Database functions** You can now assign result of **DbQuery** to a local index variable, letting you create a single variable or function to return a relational table, without having to create auxiliary global indexes for rows and columns.
- MDX hypercube access** The new **MDXQuery** function supports the standard MDX language for querying and writing to multidimensional OLAP hypercube databases, such as offered by Microsoft SQL Server Analysis Services. It greatly expands ways to integrate Analytica with business intelligence and related applications.
- MDTable** Now lets you specify the first N columns of X as *coordinates* and the rest as *measures*, as used in a *fact table*, the format used to specify OLAP hypercubes. It also lets you pass it a conglomeration function. See “MDTable(t, rows, cols, vars, conglomFn, missingVal)”.
- Performance Profiler library** This library now shows a sorted table with memory and time used by each variable and function. Double-click any object title to open its **Object** window.

Analytica Optimizer

The Analytica Optimizer uses the new 7.0 release of the Premium Solver from Frontline Systems. New features include: **Grouped Integer** variable type, where a solution must assign a different integer from 1 to n to each variable in the group. The quadratic programming solver, **QpDefine**, now supports quadratic constraints in addition to linear constraints. **SolverInfo** function returns information about the current solver. The solvers offer a more flexible option for passing all parameters as a single array of parameters, labeled by parameter name in the index. You can add yet more powerful solvers, including OptQuest, Knitro NLP, Mosek SOCP and NLP, and Xpress LP, QP and MIP (priced separately). See the *Optimizer Guide* or *What's New in Optimizer 4.0* for more.

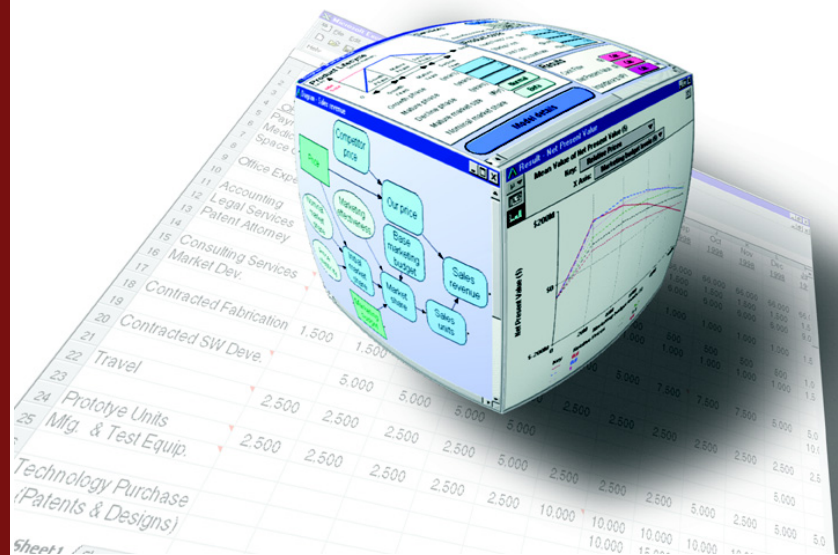
- Generalized Regression library** Offers **Logistic_Regression** and **Probit_Regression**, using the Optimizer. See “Logistic_Regression(Y, B, I, K)” and “Probit_regression(Y, B, I, K)”.

Chapter 1

Examining a Model

This chapter introduces the basics of how to open and view an Analytica model, generate results, and print them, including:

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



To open or exit a model

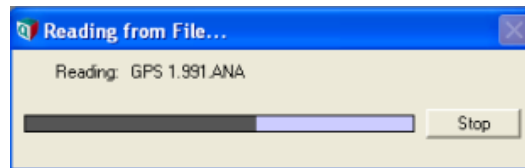
Models An Analytica **model** is a collection of variables, modules, and other objects intended to represent some real-world system you want to understand. Between sessions, a model is stored in an Analytica document file with the file type ".ana".

To open a model The simplest way to open an existing model is just to double click the icon for the model file in the Windows directory.

Another way to open a model is to:

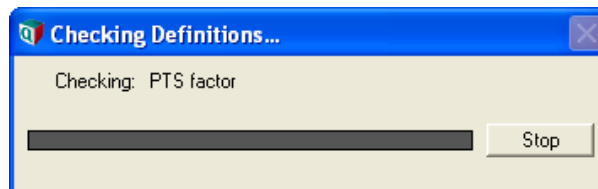
1. Start up Analytica — by double-clicking the icon of the Analytica application, or selecting **Analytica** from the Windows **Start** menu. Analytica will open a new, untitled model.
2. In the top left of the Analytica application window, press on the **File** pull-down menu, and select **Open Model**. A directory browser dialog will appear to let you find the model file you want.

Whichever way you start a model, Analytica will show this progress bar as it reads in the model file:



Tip Click the **Stop** button if you change your mind and decide not to open the model. It will stop reading, resulting in a partially loaded model.

Next, it shows a progress bar as it checks the definitions of variables and functions in the model:



Tip If you click the **Stop** button, it will stop checking. Diagrams may have missing arrows and cross-hatched nodes indicating unchecked definitions. If you later ask to show the result of a variable, it will check any variables needed. Thus, clicking **stop** here simply defers some checking, and causes no problems with the model.

If the model contains any variables whose definitions are missing or invalid, it will let you know by listing them in the **Invalid Variables** window (see "Invalid variables"). You can still compute results for variables with valid definitions, as long as they don't depend on variables whose definition is invalid.

To close 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

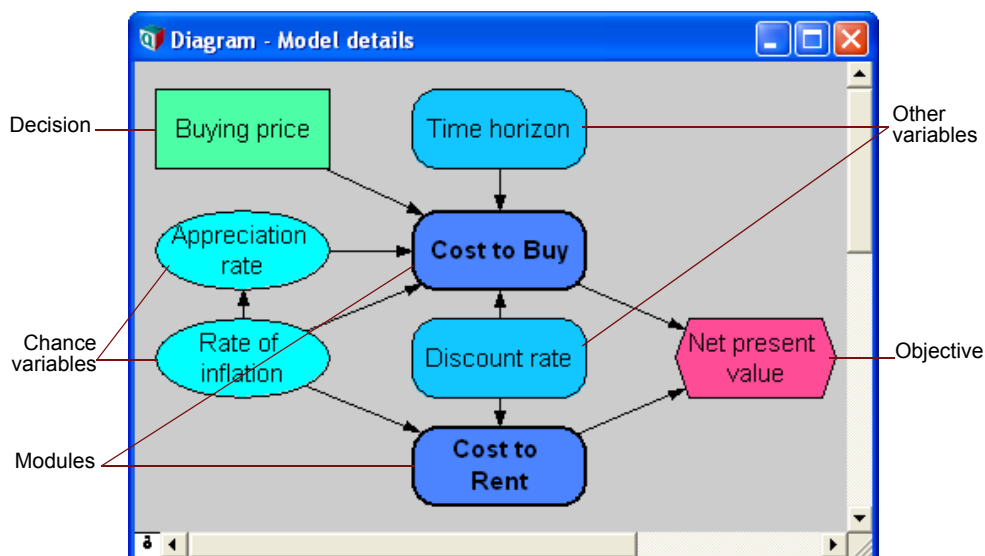
before closing — except if you are using the Player Edition, which doesn't let you save a changed model.

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

To exit Analytica To exit (or quit) Analytica, select **Exit** from the **File** menu. If you have made any changes to the model, it will prompt you to save your model first (if you are not using the Player Edition).

Diagram window

When you open a model it shows a **Diagram** window. This window usually shows an **influence diagram**, like this:




Each **node** depicts a variable (thin outline) or module (thick outline). The node shape and color tells you its class — decision, chance, objective, module, and so on. The arrows in a **Diagram** window depict the **influences** between variables. An influence arrow from variable **A** to variable **B**, means that the value of **A** influences **B**, because **A** is in the definition of **B**. So, when the value of **A** changes, it may change the value (or probability distribution) for **B**.

In the diagram above, the arrow from **Buying price** to **Cost to buy** means that the price of the house affects the overall cost of purchasing it. 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”.

To view results



To view the value of a variable, first click its node to select it. Then click this **Result** button in the navigation toolbar to open a **Result** window showing its value as a table or graph. Chapter 2, “Result Tables and Graphs,” tells you more.

Tip If it needs to calculate the value, it shows the waiting cursor  while it computes.

Opening details from a diagram

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

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

Going to the parent diagram

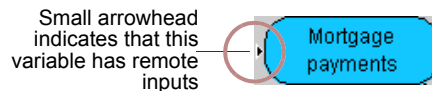


To see the diagram that contains the active module or variable, click the **Parent Diagram** button in the navigation toolbar. The module or variable will be highlighted in the parent diagram.

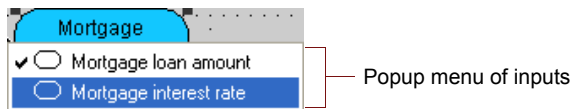
Tip If the active diagram is of the top model, it has no parent diagram, and the **Parent Diagram** button is grayed out.

Seeing remote inputs and outputs

When a variable has a Remote input — that is, it depends on a variable in another module — a small arrowhead appears to the left of its node. Similarly, if it has a remote output, a small arrowhead appears to its right. Press on the arrowhead to quickly view and navigate influences between nodes in different diagrams (modules).



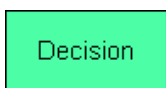
To see a list of the inputs (or outputs), remote and local, press the arrowhead on the left (or right) of the node:



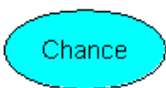
To jump to a remote input or output, select it from the list and stop pressing. It opens the **Diagram** window containing the remote variable, and highlights its node.

Classes of variables and other objects






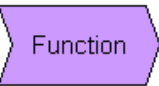

The shape of a node indicates the class of the variable or other object:



A rectangle depicts a **decision variable** — 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, it is your decision how much to bid.



An oval depicts a **chance variable** — that is an uncertain quantity whose definition contains a probability distribution. For example, whether or not it will rain tomorrow 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 hexagon depicts an objective variable — a quantity that evaluates the relative value, desirability, or utility of possible outcomes. In a decision model, you are trying to find the decision(s) that maximize (or minimize) the value of this node. Usually, a model contains only one objective.
	A rounded shape (with thin outline) depicts a general variable — a quantity that is not one of the above classes. It may be uncertain because it depends on one or more chance variables. Use this class initially if you're not sure what kind of variable you want. You can change the class later when it becomes clearer.
	A rounded node (with thick outline) depicts a module — that is, a collection of nodes organized as a diagram. Modules can themselves contain modules, creating a nested hierarchy.
	A parallelogram depicts an index variable . An index is used to define a dimension of an array. For example, <i>Year</i> is an index for an array containing the U.S. GNP for the past 20 years. Or <i>Nation name</i> is an index for an array of GNPs for a collection of nations (see “Introduction to arrays”). Indexes identify the row and column headers of a table, and the axes and key of a graph.
	A trapezoid depicts a constant — that is, a variable whose value is fixed. A constant is not dependent on other variables, so it has no inputs. Examples of numerical constants are the atomic weight of oxygen (16) or the number of feet in a kilometer. It is clearer to define a constant for each such value you need in a model, so you can refer to them by name in each definition that uses it, rather than retyping the number each time
	A shape like an arrow tail depicts a function . You can use existing functions from libraries, and define new functions to augment the functions provided in Analytica. See Chapter 20, “Building Functions and Libraries.”
	This node is a button — when you click a button (in browse mode), it executes its script to perform some useful action. You can use buttons with any edition of Analytica, but you need Analytica Enterprise or Optimizer to create a new button. See “Creating buttons and scripts” on page 387.

Selecting nodes

To view or change details of a variable or other object in a diagram, you must first select a node (or a set of nodes). You do this in much the same way as you select files or folders in the Windows File Browser, and most other applications:

To select a node Simply click a node once to select it. Selected node(s) are highlighted with reverse color in browse mode, or with handles (little corner squares) in edit mode.

You can also press the *Tab* key to select a node. Each time you press *Tab*, it selects the next node in the diagram, in the order the nodes were created. *Control+Tab* cycles through the nodes in the reverse sequence.

To select multiple nodes Click a node while pressing the *Shift* key to add it to the set of selected nodes. You can remove a node from the selection by clicking it again while pressing *Shift*.

In edit mode, you can also select a group of nodes by dragging the selection rectangle to enclose them. Press the mouse button in a corner of the diagram — say top left — and drag the cursor to the opposite corner — say bottom right. This will show the selection rectangle and select all nodes within the rectangle.

To deselect all nodes Just click the background of the diagram outside any node.

The toolbar

The toolbar appears across the top of the Analytica application window. It contains buttons to open various views of the model, and to change between browse and edit modes.



Navigation toolbar The first five buttons on the toolbar each open a window relating to the variable or the object selected in the active (frontmost) window:



Parent Diagram button: Click to open the **Diagram** window for the module or model containing the object in the current active **Diagram**, **Object**, or **Result** window. It highlights the object you were viewing in the parent diagram. If you are viewing the top-level model, which has no parent, this button is grayed out. The keyboard shortcut is *F2*.



Outline button: Click to open the **Outline** window. The outline highlights the object you were previously looking at. See “The Outline window”. The keyboard shortcut is *F3*.



Object button: Click to open the **Object** window for the selected node in a diagram or the active module. See “The Object window”. The keyboard shortcut is *F4*.



Result button: Click to open a **Result** window (table or graph) for the selected variable (see “The result window”). This button is grayed out if no variable is selected. If you have selected more than one variable, it will offer to create a compare variable that shows a result combining the values of all the variables. The keyboard shortcut is *Control+r* or *F5*.



Definition button: Click to view the definition of the selected variable. If the variable is defined as a probability distribution or sequence, it opens the function in the **Object Finder** (see “Object Finder dialog”); if the variable is an editable table (edit table, sub-table, or probability table), it opens the **Edit Table** window (see “Viewing an array as an edit table”). Otherwise, an **Attribute** panel (see “The Attribute panel”) or an **Object** window (see “The Object window”) opens, depending on the *Edit Attributes* setting in the **Preferences** dialog box (see “Preferences dialog”). This button is grayed out if no variable is selected. The keyboard shortcut is *Control+e* or *F6*.


Edit buttons These three buttons control your mode of interaction with Analytica. The shape of the cursor reflects which mode you are in:



Browse tool: Lets you navigate a model, compute and view results, and change inputs. It will not let you change other variables. See “Browse mode”.



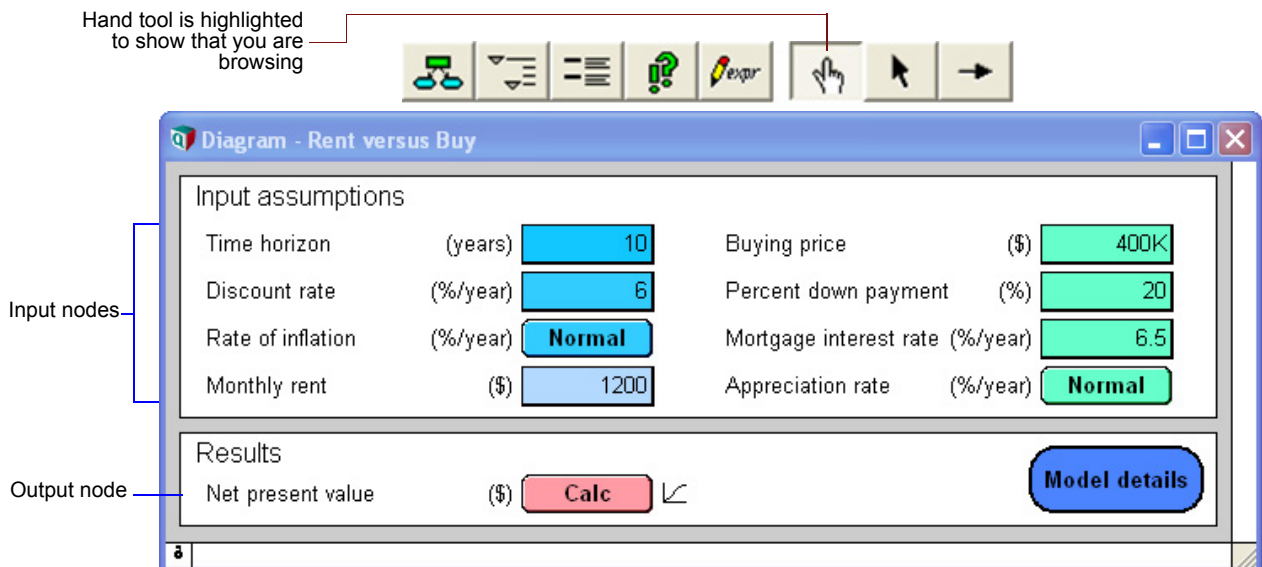
Edit tool: Lets you create new objects, and move and edit existing objects. See “Creating and editing nodes”.

 **Arrow tool:** Lets you draw arrows (influences) between nodes on a diagram. See “Drawing arrows”.

Tip If the model is locked as browse-only, or if you are using the Player or Power Player edition of Analytica, only the browse tool is available.


Browsing with input and output nodes

When you open a model with input and output nodes, the top-level **Diagram** window may look like this (instead of an influence diagram):



You can change the values in the **input nodes** directly. The **output node**, Net present value, shows a **Calc** button. Click it to compute and see its value. Double-click the **Model details** node to open up a diagram showing details of the model (the influence diagram shown above).

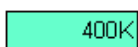
Browse mode

An existing model opens in browse mode. In browse mode, the **browse tool** button is highlighted in the navigation toolbar, and the cursor looks like this .

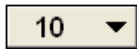


In the browse mode, you can change input node values, view output node results, and examine the model by opening windows to see more detail.

Viewing input nodes



An input field lets you see a single number or text value. Click in the box to edit the value. If it's a text value, you must put matching quotes around it (single or double).



A pulldown menu lets you choose from a menu of alternatives. Press the menu to see the alternatives.



Click the button to open a list of values, usually defining an Index. To change a value, click in its cell. For more about lists, see “Editing a list”.

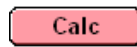


Click to open an **edit table** showing an editable array with one or more dimensions displayed as a table. For more, see “Editing a table”.

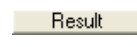


Click to view and edit a probability distribution in the **Function Finder**. For more, see “Probabilistic calculation”.

Viewing output node values



Click the **Calc** button to compute and display the value of this output variable. When computing is complete, it will show a number in this node, or, if it’s an array, it changes to the **Result** button and opens a **Result** window showing a table or graph. See Chapter 2, “Result Tables and Graphs” for more.



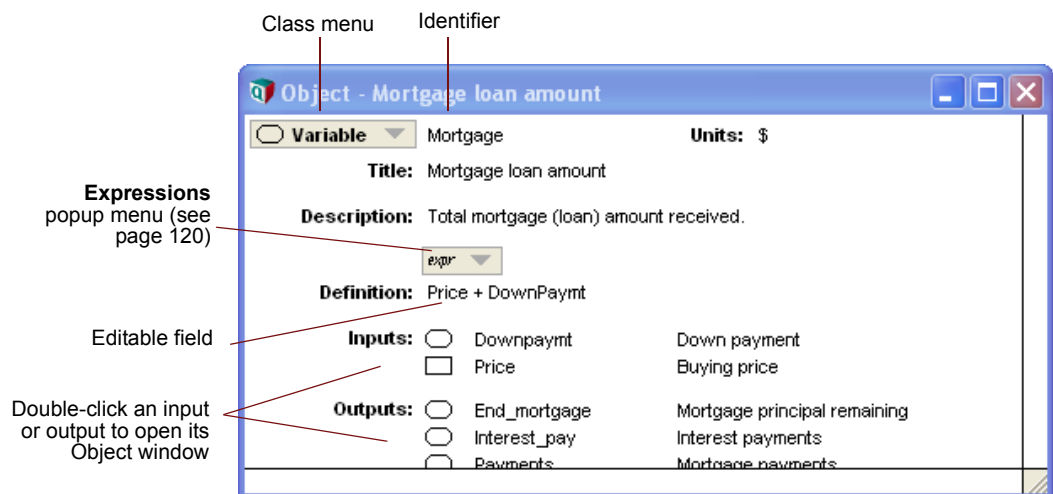
The **Result** button shows that an array has been calculated. Click it to open a **Result** window showing a table or graph. See Chapter 2, “Result Tables and Graphs” for more.

Opening module details

To see the structure of the model, double-click the module **Model details**, to display its diagram window (see “The Object window”).


The Object window

The **Object window** shows the attributes of an object. All objects have a class and identifier — a unique name of up to 20 characters. A variable also has a title, units, description, definition, inputs, and outputs:



To open an Object window

Here are some ways to open the **Object** window for an object **x**:

- Double-click **x** in a **Diagram** window (see page 22).
- Select **x** in its **Diagram** window and click the **Object** button  in the navigation toolbar.
- Double-click the entry for **x** in the **Outline** window (see “The Outline window”).
- If a **Result** window for **x** is displayed, click the **Object** button in the navigation toolbar.
- Double-click **x** in the **Inputs** or **Outputs** list of a variable in an **Object** window.

Returning to the parent diagram



Click the **Parent Diagram** button in the navigation toolbar to see the diagram that contains this node, with the node highlighted.

The Attribute panel

The **Attribute panel** offers a handy way to rapidly explore the definitions, descriptions, or other attributes of the variables and other nodes in a **Diagram** window. You can open the panel below the diagram, and use it to view or edit any attribute of the node you select. It shows the same attributes that you can see in the **Object** window, and often several other attributes.

Select node to see its attribute below

Key icon is open

Title of the selected object

Value of the attribute

Attribute menu from which you select the attribute to show

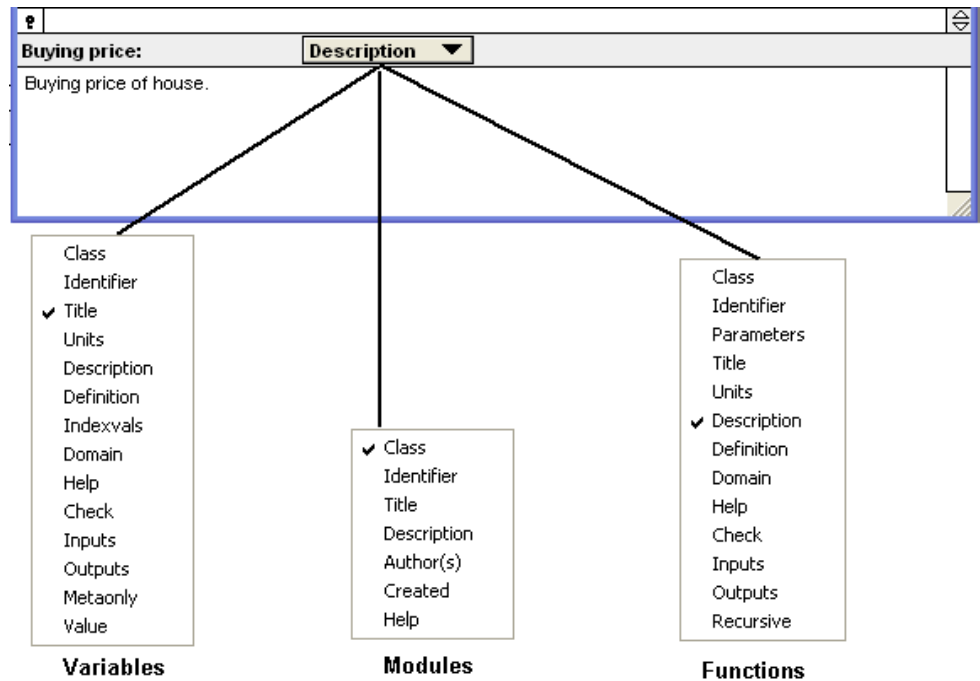
Drag partition to change panel height

Drag box to change panel height and diagram width

Click the key icon  to open the **Attribute** panel. Here are things you can do in it:


- Select another node in the diagram to see the selected attribute of a different object.
- Click the background of the diagram to see the attributes of the parent module.
- Select another option from the **Attribute** menu to see a different attribute.
- To enter or edit the attribute value, make sure you are in edit mode, and click in the attribute panel, and start typing. (Not all attributes are user-editable.)

Different classes of objects have different sets of attributes:



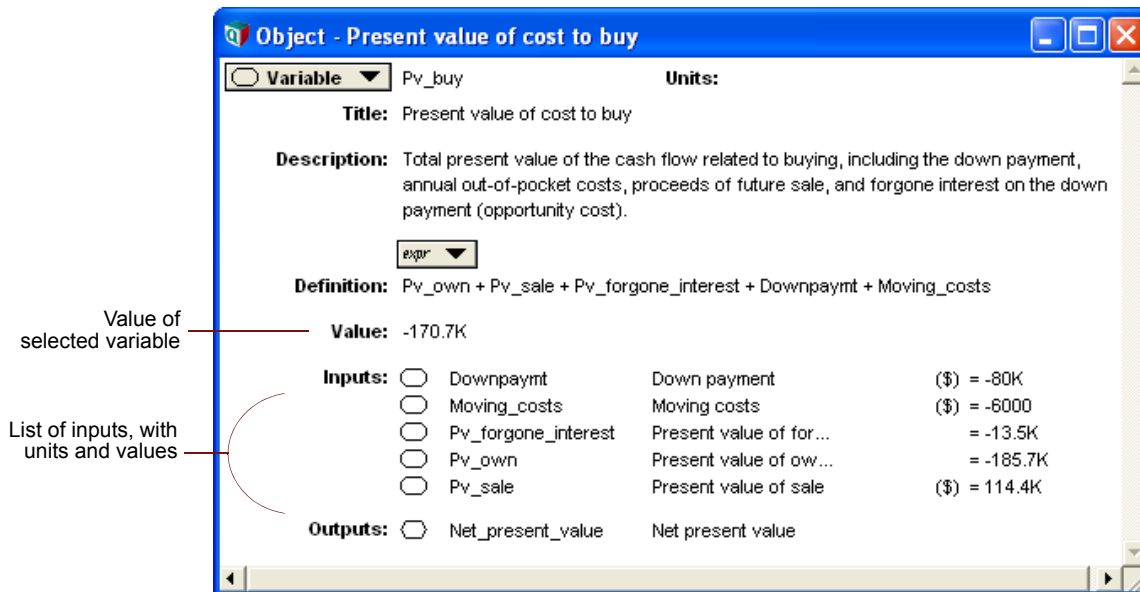
If you try to see an attribute not defined for an object, it will show its description.

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

To close the **Attribute** panel, click the key icon  again,.

Showing values in the Object window

When reviewing a model and trying to understand how it works, it is useful to show the value of a variable and its inputs in the object window. To switch on this option, select **Show with Values** from the **Object** menu. The **Object** window for a variable then shows the mid (deterministic) value of the variable and each of its inputs:



Value of selected variable

List of inputs, with units and values

Atom and array values

If a value has not yet been calculated, it shows a **Calc** button. Click to compute it. If the resulting value is an **atom** — a single number or text value, not an array — it shows the value in the **Object** window, as above. If the value is an array, it shows instead a **Result** button **Result**, which you can click to compute and display the array in a separate **Result** window.

For more about the **Result** window, see Chapter 2, “Result Tables and Graphs.”

Printing

To print the contents of an active window — **Diagram**, **Outline**, **Object**, **Result Table** or **Graph** — select **Print** from the **File** menu. Selecting **Print Setup** on the **File** menu can then set printing options such as page orientation, paper size, or scaling. Any print settings that you specify are associated only with the window that was active when you selected **Print Setup**.

Previewing page breaks before printing

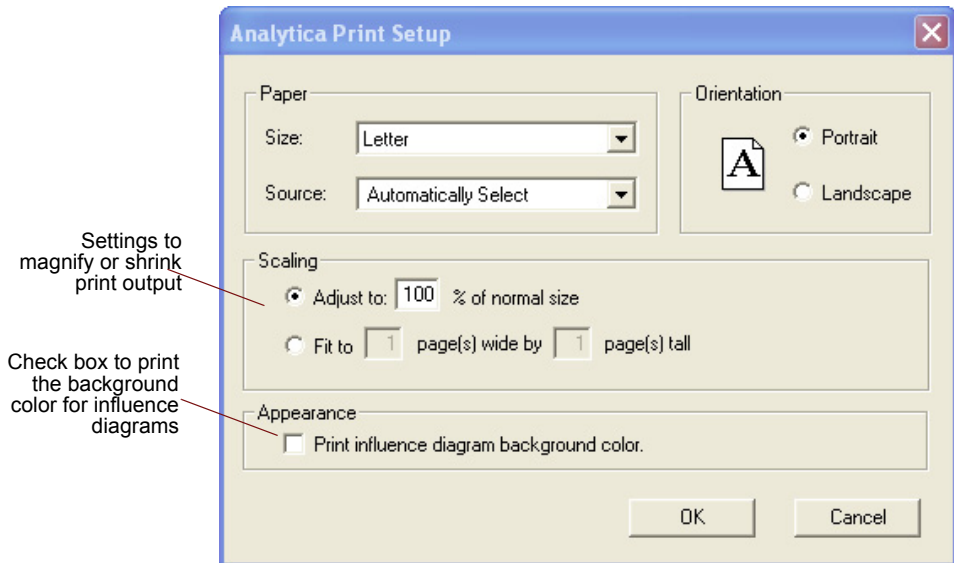
When you select the **Print preview** command on the **File** menu, it displays a **Preview** window to show what will be printed and where page breaks will occur. You can adjust print settings such as scaling until you get the desired page breaks. When previewing a result table or graph, you can toggle the option for showing or hiding the index variable titles.

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, in two ways:

- **Adjust to p % of normal size:** $p < 100\%$ to shrink output. $p > 100\%$ to enlarge it.
- **Fit to n page(s) wide by m page(s) tall:** Shrinks the output to fit on the specified pages. It preserves aspect ratio. It does not enlarge, so the actual number of pages printed may be less than $n \times m$.

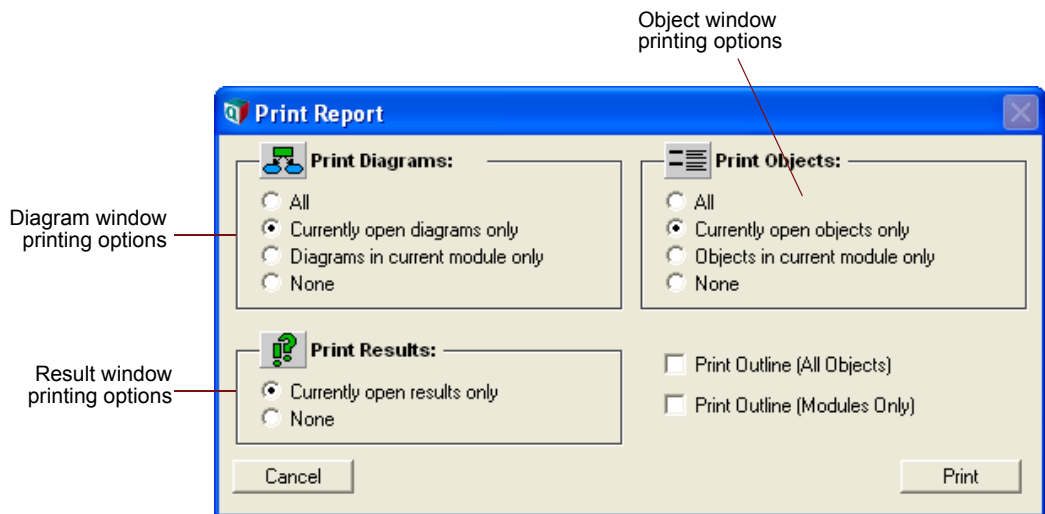


Printing the background

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 into a single document, use the **Print Report** command in the **File** menu. It uses the print settings set in **Print settings** for each window.



Check **Print outline (all objects)** to print a list of all objects in the model, each in its parent module, indented to show the module hierarchy.

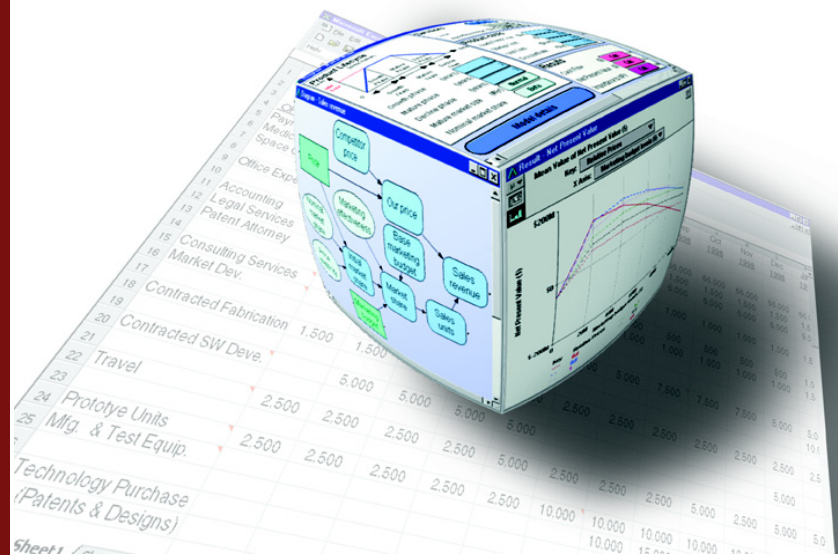
Check **Print outline (modules only)** to print a list of all modules (including libraries and form nodes), indented to show the module hierarchy.

Chapter 2

Result Tables and Graphs

This chapter shows you how to:

- View **Result** windows as graphs or tables
- Rearrange or pivot results, exchanging rows and columns, or graph axes and keys, and slicer dimensions.
- Select an uncertainty view to display probabilistic results.
- Compare two or more variables in the same table or graph.



The result window

When you open the **Result** window for a variable, it computes its value if it hasn't previously cached it, and displays it. If the value is an array or a probability distribution, you can display it as a table or graph. Here is a **Result** window with a table and equivalent graph:

Result controls

Index selection area

Uncertainty View popup menu

Table view

Graph view

	1		2		3		4	
	X	Y	X	Y	X	Y	X	Y
Present value of cost to buy	-314.6K	0	-284.2K	5m	-281.6K	0.01	-271.9K	

Result - Net present value

Cumulative Probability of Net present value (\$)

Key: Net present value (\$)


Cumulative Probability

Net present value (\$)

— Present value of cost to buy — Present value of cost to rent

To open a result window

Click the variable node in its influence diagram to select it, and do one of these:

- Click the **Result** button  in the toolbar, or press key *Control+r*.
- Select **Show Result** from the **Result** menu.
- Select an **uncertainty view** option, such as **Mid Value**, **Mean Value**, or **Cumulative probability**, from the **Result** menu.
- In the **Attribute** panel below a diagram, select **Value** or **Probvalue** from the **Attribute** menu, and click the **Calc** or **Result** button.

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

Result controls

The **Result controls**, in the upper left corner of the **Result** window include:



Press the **Uncertainty View** popup menu, to select how to display an uncertain quantity. See "Uncertainty views".



Click this button to display the result as a **table**.

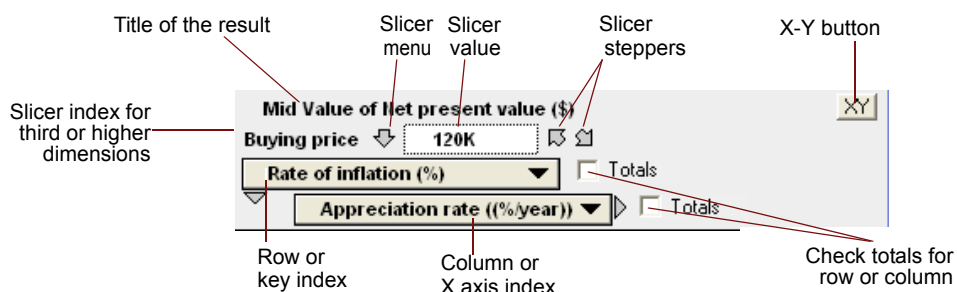


Click this button to display the result as a **graph**.

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

Index selection

The **Index selection** area is the top part of a **Result** window. For a table, it shows which index goes down the rows, and which across the columns. For a graph, it shows which index is on the X axis (and sometimes Y axis) and which is in the key. For either view, if the array has too many dimensions to display directly, it also shows **slicers** that select the values of the extra indexes. Each control has a popup menu to let you exchange indexes and rearrange (**pivot**) the view:



The index selection area of a graph or table contains these items (example variables and indexes in the following text refer to the figure above):

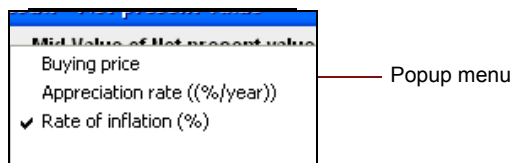
Title Shows the uncertainty view (Mid, Mean, etc.), the title of the variable, and its units, e.g., **Mid Value of Costs of buying and renting (\$)**.

Slicer index The title, units, and value of any index(es) showing dimensions not currently displayed in the table or graph.

Slicer menu Press for a popup menu from which you can change the slicer value for the results displayed.

Slicer stepper arrows Click or to cycle up or down through the slicer values.

Row or key index Shows the title of the index displayed down rows for a table, or in the color key for a graph. Press to open a menu from which you can select another index:



Column or X axis index Shows the title of the index displayed across the columns for a table, or along the X (horizontal) axis for a graph. Press to open a menu from which you can select another index.

XY button Click to be able to plot this variable against one or more other variables, or to plot one slice of this variable against another slice. See "XY comparison" on page 105.

Totals check boxes Check a box to show row or column totals the table view. If you check Totals for an index and then pivot it to be a slicer index, "Totals" will be its default slicer value. This lets you show total values over the slicer index in the graph or table.

The default view

When you first display a result for a variable, by default, it displays it as graph, if possible, and otherwise as a table. You can change this default in the **Default result view** in the **Preferences** dialog box (see “Preferences dialog”).

When you display the **Result** window again, it uses all the options you last selected when you viewed this variable, including table versus graph, uncertainty view, index pivoting and slicer values, and any graph settings.


Recomputing results

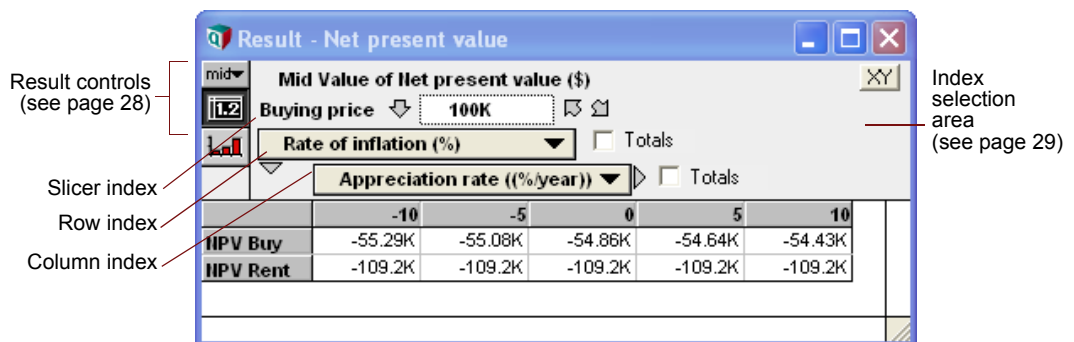
If you change a predecessor of a variable shown in a **Result** window, the table or graph disappears from the window and is replaced by a **Calculate** button.



Click **Calculate** to compute and display the new value.

Viewing a result as a table

Toggle to table view If a result window shows a graph, click  on the top left to toggle to table view.






Three-dimensional table

The index display options depend on the number of dimensions in the variable.

Row index (down) Use this menu to select which index to display down the rows of the table. Select blank to display a single row.


Column index Use this menu to select which index to display across the columns of the table. Select blank to display a single column.

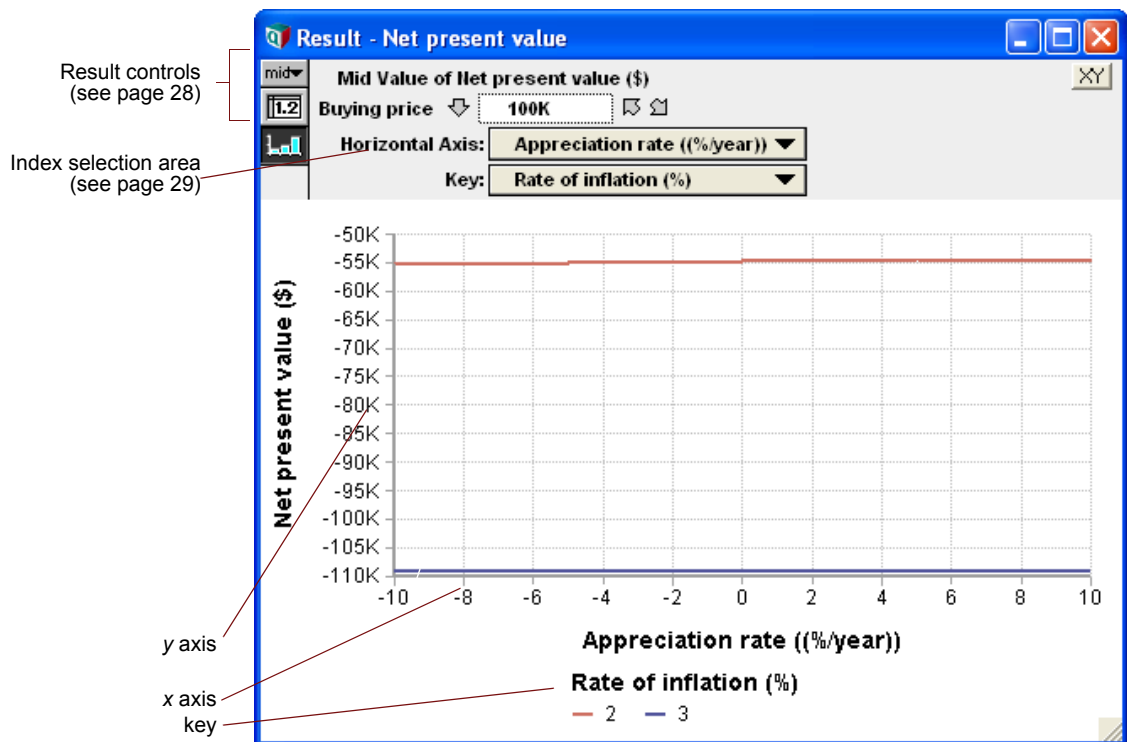
Slicer index(es) If the array has more than two indexes, the extra index(es) are shown as **Slicer** menus. The table shows values only for the slice (subarray) setting the slice index to the shown slicer value. Open the slicer menu  select a different slicer value, or click  or  to step through the slicer values.

Formatting numbers To specify the format for the numbers in a table or along the Y (usually vertical) axis of a graph, show the graph and select **Number Format** from the **Result** menu, or press

Control-b. The **Number format** dialog offers many options, including currency signs, dates and Booleans. See “Number formats” for details.

Viewing a result as a graph




Toggle to graph view If a result window shows a table, click  on the top left to toggle to graph view.



The **y** axis, usually vertical, plots the values of the variable. The **x** axis, usually horizontal, shows the value of a selected index. The index display options depend on the number of dimensions in the variable:

X axis If the array has more than one index, use this menu to select which index to display along the **x** axis (usually horizontally).

Key index If the array has more than one index, use this menu to select which index to display in the key, usually showing each value by color.

Slicer index(es) If the array has more indexes than you can assign graphing roles (such as **x** axis or key), the extra indexes are shown as **Slicer** menus, as in a table view. The graph shows values only for the slice (subarray) setting the slice index to the shown slicer value. Open the slicer menu  select a different slicer value, or click  or  to step through the slicer values.

To reorder slicers If the graph has more than one slicer index, you can reorder the slicer indexes simply by dragging one up or down.

Graph setup options There are a rich variety of ways to customize the graph, including line style (lines, data points, symbols, barcharts, stacked bars, thickness, transparency), axis ranges, log or

inverted axes, grid and tickmarks, background colors, and font color and size. To change these settings, open the **Graph Setup** dialog:

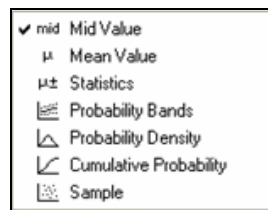
- Select **Graph Setup** from the **Result** menu, or
- Double-click anywhere on a graph in the **Result** window.

See “Graph setup dialog box” for details.

Uncertainty views

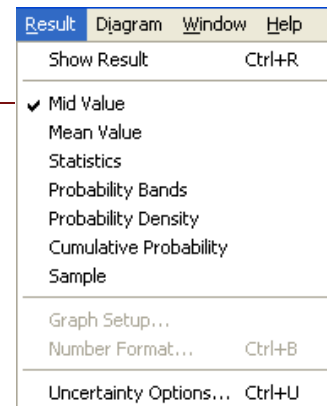
Every variable has a certain or deterministic value, which we term its *mid* value. Some variables, notably *chance* variables and variables that depend on chance variables, may also have an *uncertain* (probabilistic) value, which we term a *prob* value. A mid value is computed using the mid value of each variable it depends on or the median of any probability distribution. The mid value of a result is not necessarily the median of its probability distribution, but usually close.

The **Result** window offers a Mid value and six *uncertainty views* as a way to visualize a prob value. You can select the uncertainty views from a menu in the top-left corner of a **Result** window. Or you may select a variable, and select an uncertainty view option from the **Result** menu:



Uncertainty View popup menu

Currently selected uncertainty view option



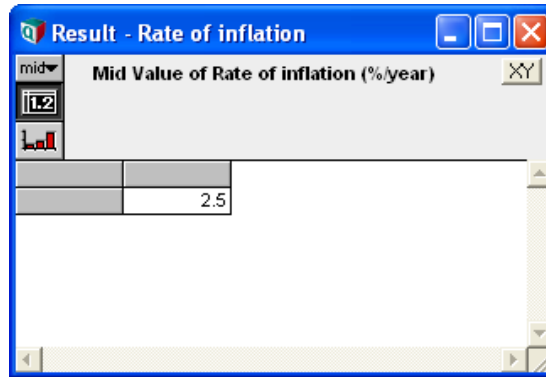
Result menu uncertainty view options

The check mark indicates the currently selected view.

Here we illustrate each uncertainty view using the chance variable, **Rate_of_inflation**, defined as a normal distribution with a mean of 2.5 and a standard deviation of 1:

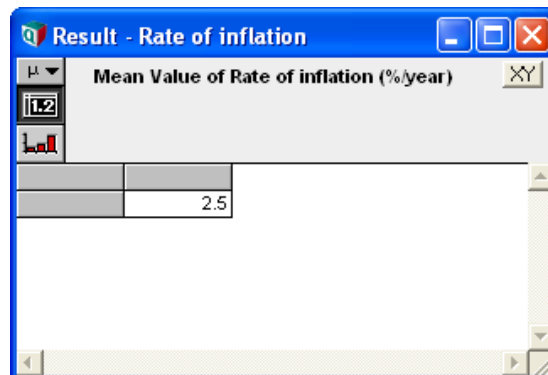
```
Chance Rate_of_inflation := Normal(2.5, 1)
```

Mid value The deterministic value, computed by using the median instead of any input probability distribution. It is computed very quickly compared to uncertain values. It is the only option available for a variable that is not probabilistic:

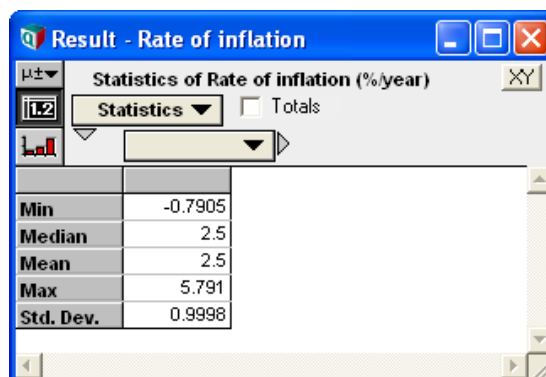


Tip A mid value is much faster to compute than a prob(abilistic) value, since it doesn't use Monte Carlo simulation to compute a probabilistic sample. It is often useful to look first at the mid value of a variable as a quick sanity check. Then you might select an uncertainty view, which will cause its prob value to be computed if it has not already been cached.

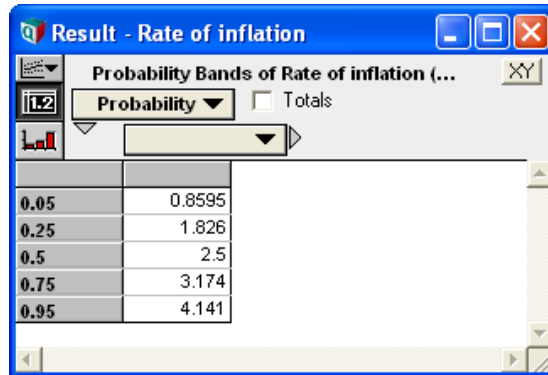
Mean value An estimate of the mean (or expected value) of the uncertain value, based on the random (Monte Carlo) sample.



Statistics A table of statistics of the uncertain value, usually, the minimum, median, mean, maximum, and standard deviation, estimated from the random sample. You can select which statistics to show in the **Statistics** tab of the **Uncertainty Setup** dialog from the **Result** menu (see "Statistics option").

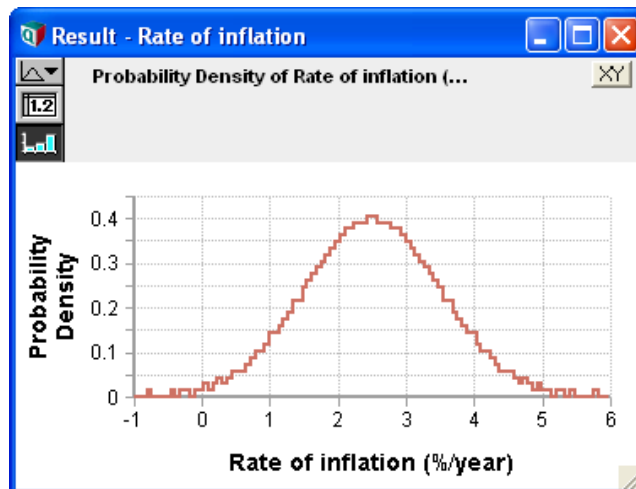


Probability bands An array of percentiles (fractiles) estimated from the random sample, by default the 5%, 25%, 50%, 75%, and 95%iles. You can select which percentiles to show in the **Probability bands** tab of the **Uncertainty Setup** dialog from the **Result** menu (see “Statistics option”).

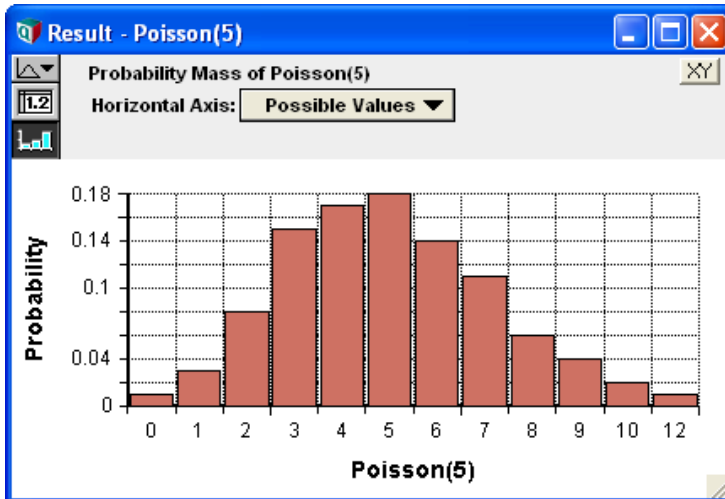


Probability density Select **probability density** to display the uncertain distribution as a probability density function (PDF).

For a probability density function, it plots values of the quantity over the X (usually horizontal) axis, and probability density on the Y (vertical axis). Probability density shows the relative probability of different values. High values show probable regions; low values show less probable regions. The peak is the mode, the most probable value. If the density is zero, it is certain that the quantity will not have values in that range.



Probability mass function If you select **Probability density** for a discrete variable, it actually displays the variable as a **probability mass function** (PMF) in a bar graph with the height of each bar indicating the probability of that value:

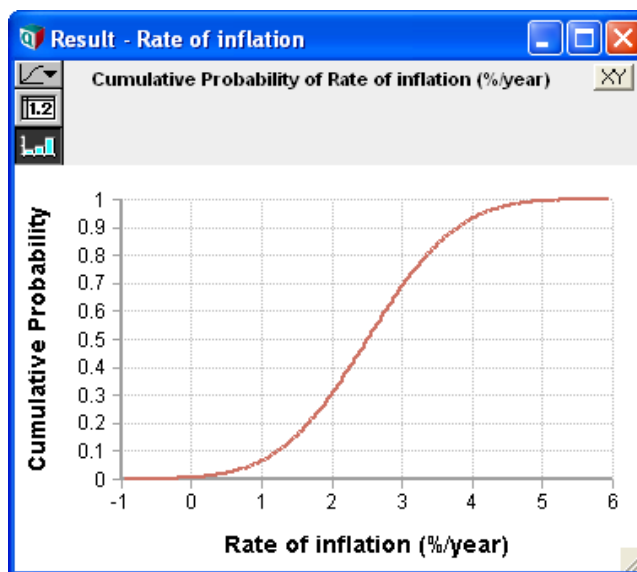


Usually, it figures out whether to use a probability density or mass function. Very rarely, you may need to tell it the domain is discrete. See “The domain attribute and discrete variables”. See “Is the quantity discrete or continuous?” and “Probability density and mass graphs” for more about the distinction between discrete and continuous distributions.

Cumulative probability

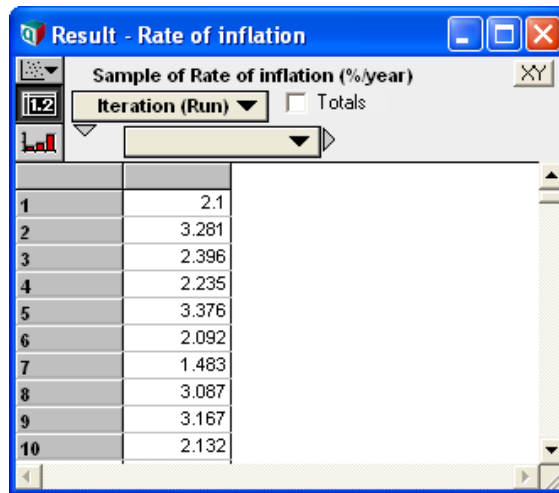
The cumulative probability distribution (CDF) plots the possible values of the uncertain quantity along the X (usually horizontal) axis. The Y value (usually height) of the graph at each value of X shows the probability that the quantity will be less than or equal to that x value. The CDF must start at a probability of 0 on the extreme left and increase to a probability of 1 on the extreme right, never decreasing.


The steeper the curve, the more likely the quantity will have a value in that region. The PDF is the slope (first derivative) of the CDF. Conversely, the CDF is the cumulative integral of the PDF.

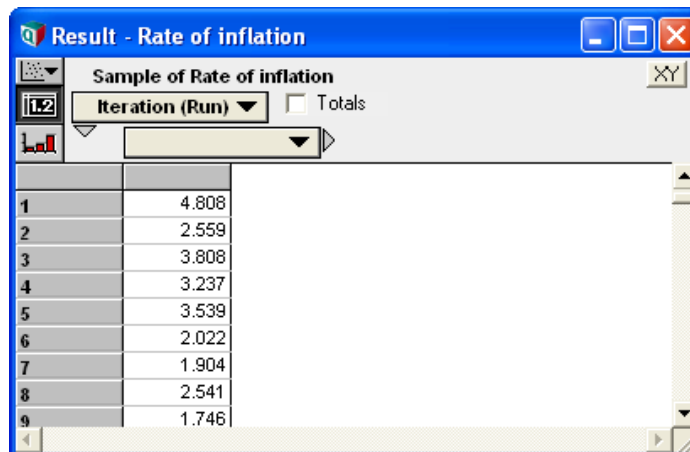


Sample A sample is an array of the random values from the distribution generated by the Monte Carlo sampling process. The sample is the underlying form used to represent each

uncertain quantity. All the other uncertainty views use statistics estimated from the sample. The sample view gives more detail than you usually want. You will likely want to view it mainly when verifying or debugging a model.



Like any other graph, you can display a sample as a table by clicking  to see the underlying numerical values:

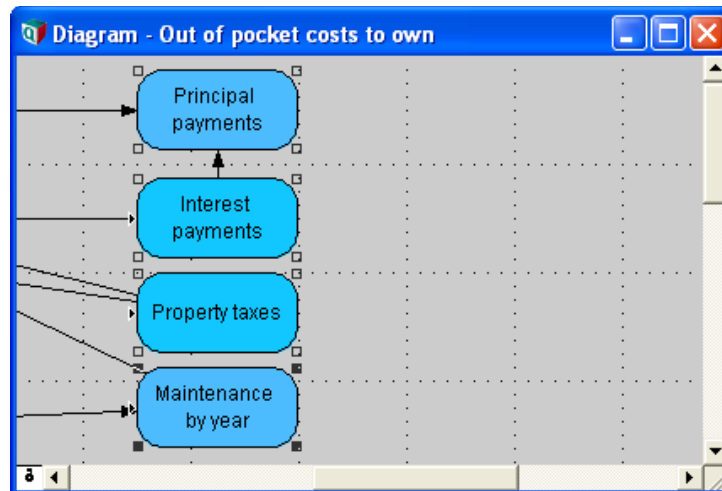



The precision of these estimates depends on the sample size and the sampling method. A larger sample size gives higher precision and takes more time and memory to compute. You can modify the sample size and sampling method in the **Uncertainty setup** dialog from the **Result** menu. See “Uncertainty Setup dialog box” “Selecting the Sample Size”.

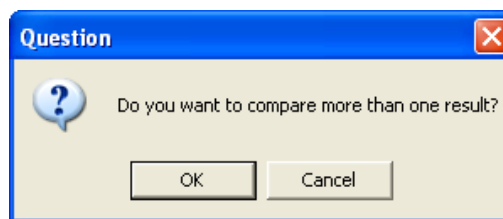
Comparing results

It's easy to compare directly two or more variables in one table or graph:

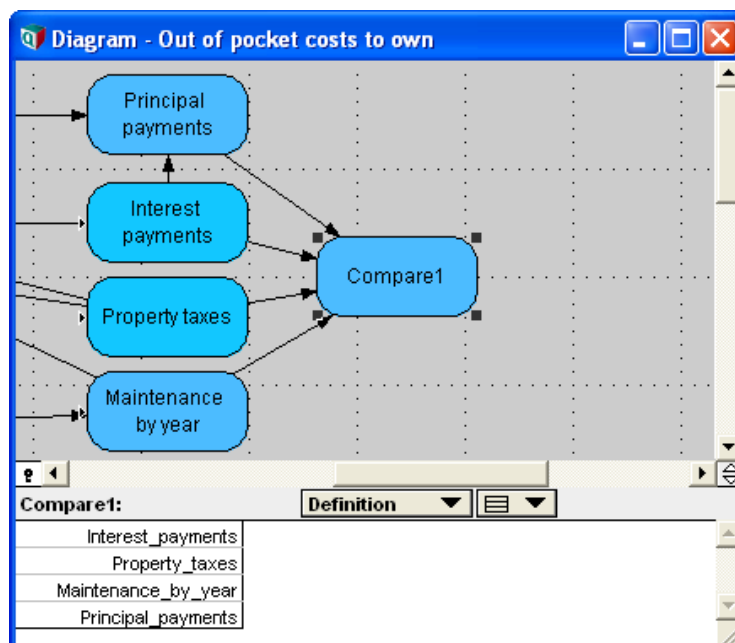
1. Select the variables together in the diagram, using *Shift+click* to add each to the selection, or dragging a selection rectangle around them:



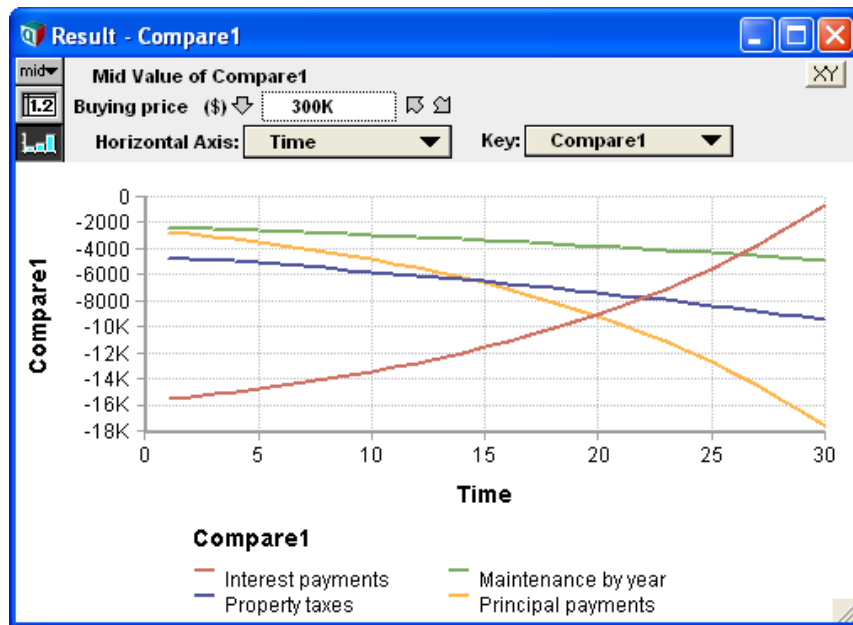
2. Click  in the navigation toolbar, or press *Control+r*.
3. Click **OK** in the confirmation dialog.



This creates a new variable with a default identifier, **Compare1**, with a list of the selected variables:



The result of `compare1` is a graph containing an index containing the titles of the variables being compared. This index is Self index of the `compare1`. It also includes all the indexes of the array variables being compared — in this case, `Time` and `Buying Price`:



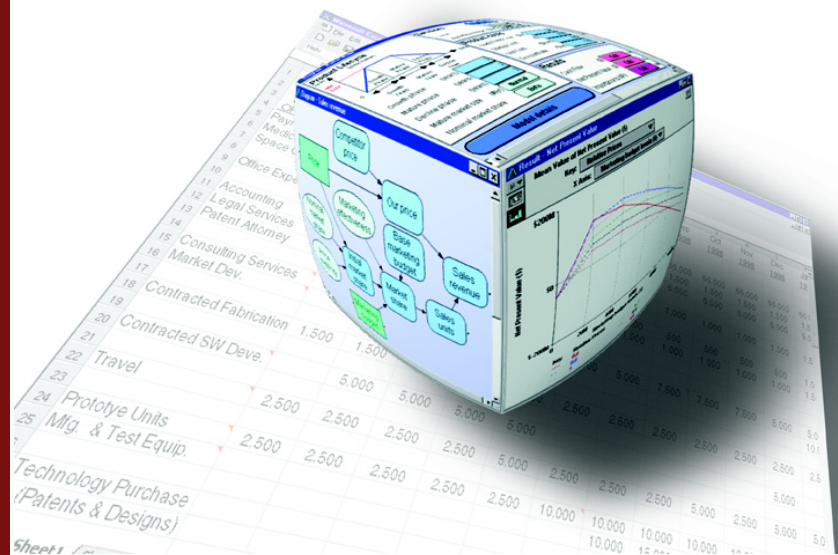
This helps clarify how the interest payments reduce (become less negative) as the principal payments on the mortgage increase (become more negative).

Chapter 3

Analyzing Model Behavior

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



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**. 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 selected 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 input values.

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 decision 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 16, “Statistics, Sensitivity, and Uncertainty Analysis”.) You can then select the most important variables as the parameters to vary to analyze model behavior.

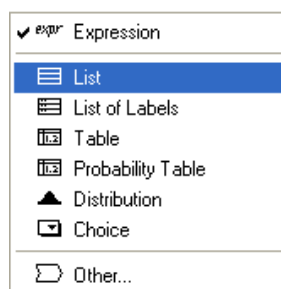
How many values to assign? Usually it is best to assign a list of three alternative values to each parameter — a low, medium, and high value. In some cases, two values may be sufficient. If you have a special interest in a particular parameter (for example, if you suspect it may have a strongly nonlinear 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 Change the definition of each parameter to a list, thus:

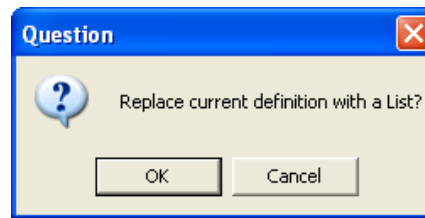
1. Select the variable by clicking its node in the influence diagram.
2. Display the variable’s definition by clicking the **Definition** button in the tools palette, or press *Control+e*.
3. Click the **Expressions** popup menu above the definition and select the **List** option. (Do *not* select the **List of Labels** option.)



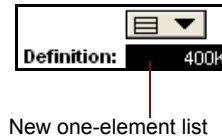
tax deductions.



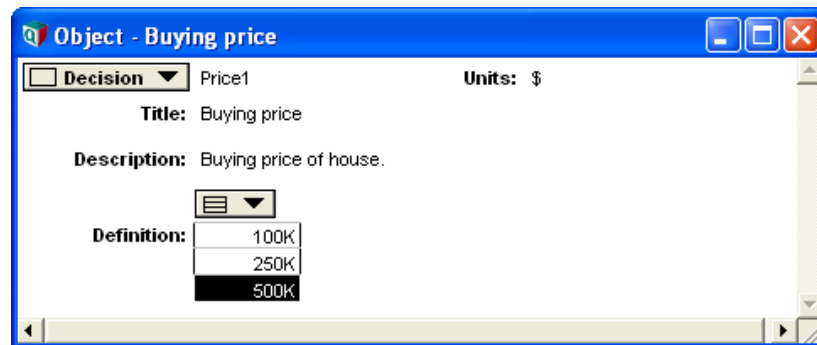
- A dialog box asks for confirmation. Click **OK**.



It displays a list with one item, containing the old definition of the variable.



- Click the item to select it.
- Type in the lowest value for the variable.
- Press *Enter* and type in the next value.
- Repeat step 7 until you have all the values you want.



Tip When you add an item to a list of two or more numbers, it uses the increment between the last two numbers to generate the next. If the last two values are 10 and 20, it offers 30 as the next.

For details on how to edit a list, see “Editing a list” on page 165.

If you want to create a list of successive integers, use the “..” operator, for example:

```
Decision Year := 2000 .. 2010
```

If you want to create a list of evenly spaced numbers, use the **Sequence(x1, x2, dx)** function, for example:

```
Decision Quarters := Sequence(2000, 2010, 025)
```

See “Sequence (start, end, stepSize)” on page 166).

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

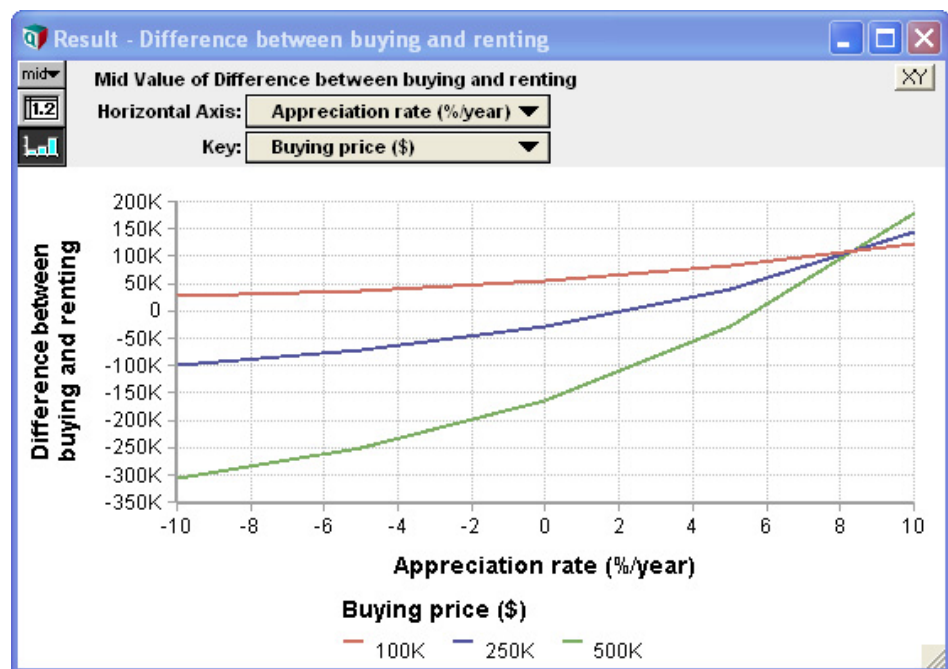
effects on model behavior you would like to explore, vary just a few at a time, rather than trying to vary them all simultaneously.

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

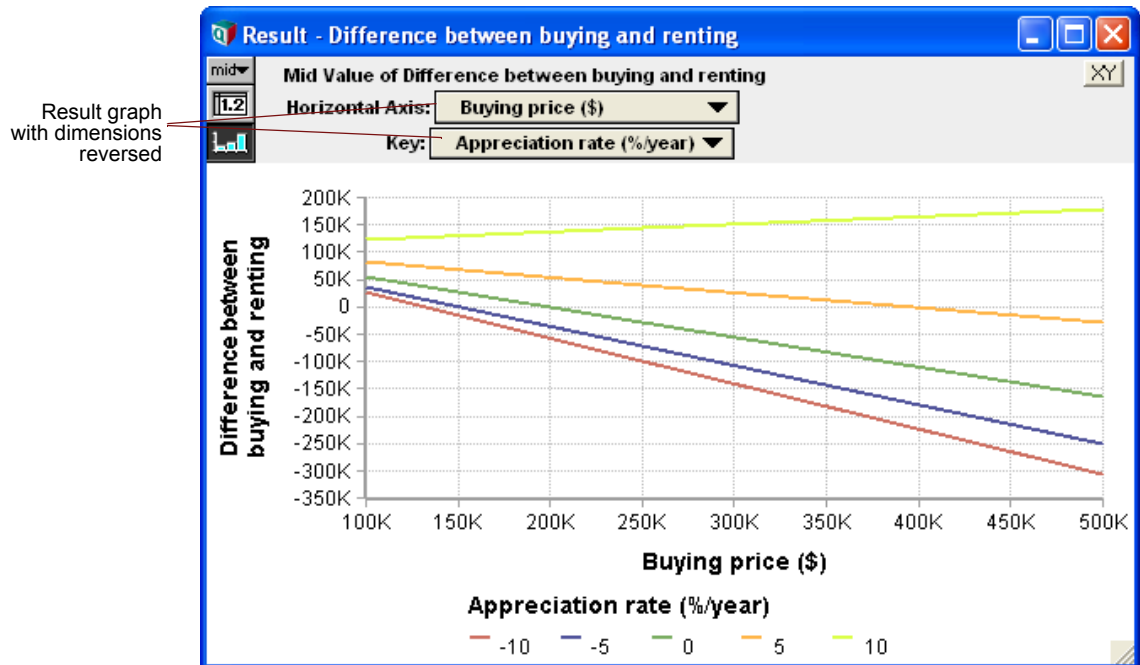
Once you have assigned a list to one or more inputs, you can examine their effect by viewing the result on an output variable. If your model has an objective, you might start by looking at that variable.

1. Select the variable you wish to view by clicking its node in the diagram.
2. View the result by clicking the **Result** button  in the navigation toolbar. 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") to get additional insights into model behavior.



Understanding unexpected behavior

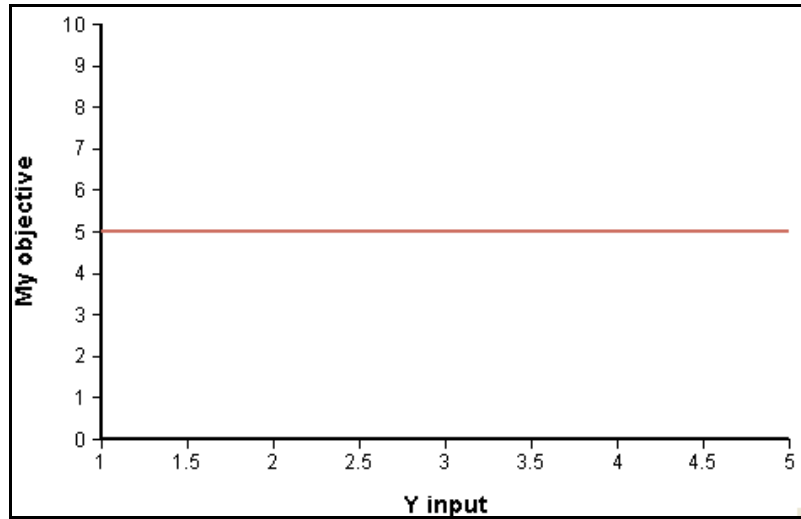
If you find the model's behavior unexpected or inexplicable, you may want to look more deeply into how the behavior arises. An easy way to do this is simply to look at the results for other variables between the input(s) and the output(s) in which you're interested. You can work forwards from an input 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.

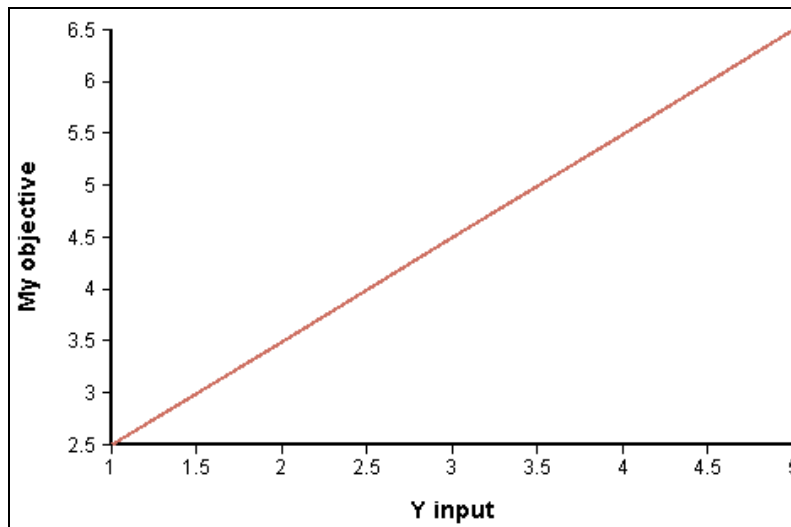
Understanding model behavior

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.

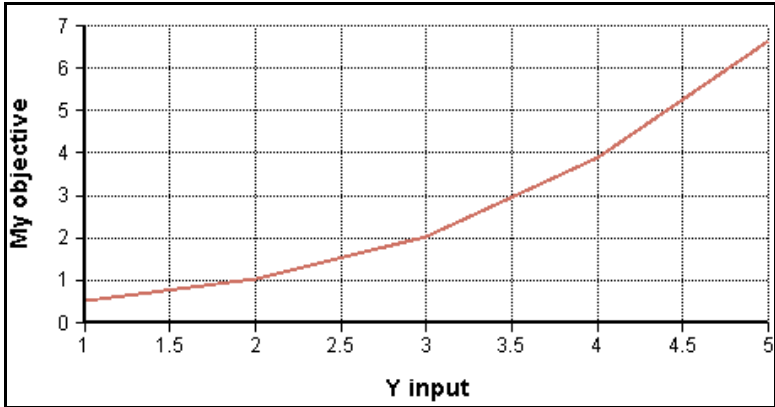
- A horizontal line shows that changes in the input over the specified range have no effect on the output.



- 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.)

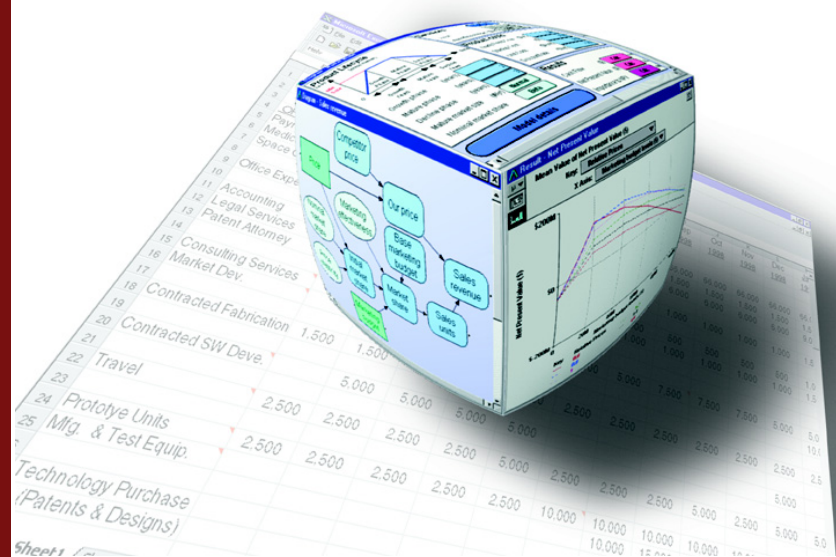


Chapter 4

Creating and Editing a Model

This chapter shows you how to:

- Create a new model
- Save changes
- Create and edit nodes
- Draw arrow connections between nodes
- Create aliases



Creating and saving a model

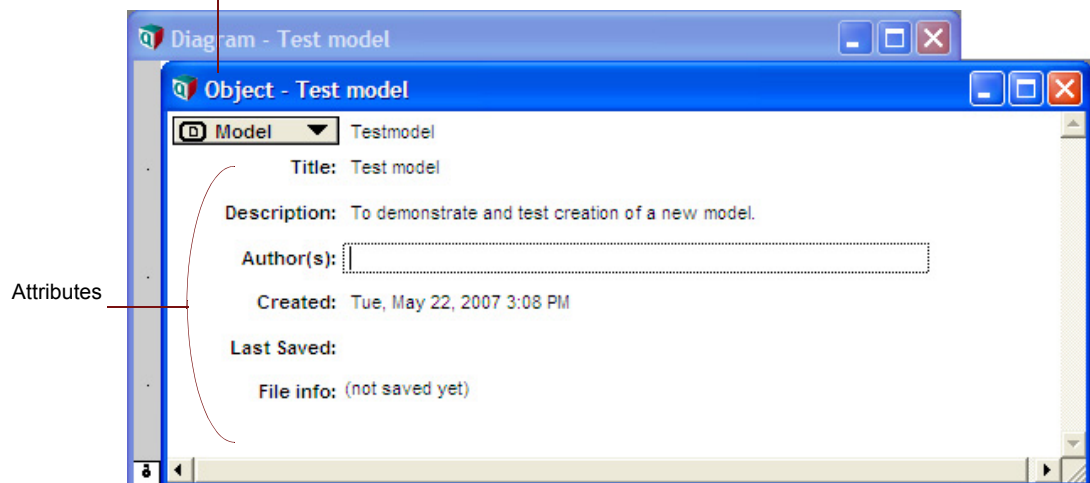
To start a new model Start Analytica like any Windows application: Select **Analytica** from the Windows **Start** menu or double-click the Analytica application file. It opens a new, untitled model.

If you are already running an Analytica model, you can also select **New Model** from the **File** menu. Since once instance of Analytica can't run two models at once, it first prompts you to save the existing model, if you have changed it.

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 start a new model, it displays the **Object** window for the new model, initially *untitled*. First, enter a title, description, and possible name yourself in the Author attribute:

Blank Diagram window




When you have entered a title and other attributes into the **Model object** window, bring the **Diagram** window to the top:



- Click the **Parent Diagram** button, or
- Click anywhere in the **Diagram** window behind the **Object** window.

You are now ready to draw an influence diagram for the new model.

Creating and editing nodes

To begin editing a diagram, if you are not already in edit mode, click the edit tool . This displays the **node toolbar** as an extension of the navigation toolbar:

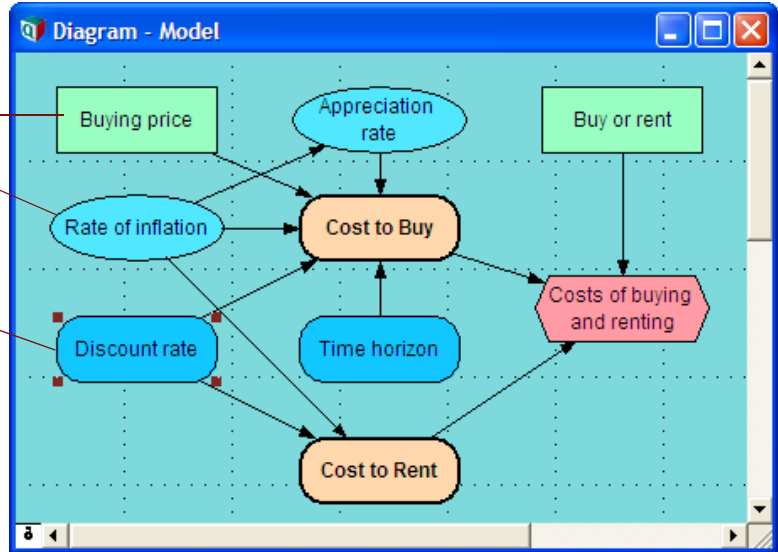


The edit tool is highlighted to show that it is selected

Node toolbar

Nodes

Selected node



For a description of each node shape (or class), see “Classes of variables and other objects”.



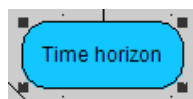
Decision
Variable
Chance
Objective
Module
Index
Constant
Function
Text
Button

The node toolbar is displayed when either the edit tool or arrow tool is selected.

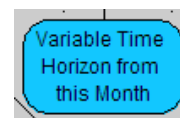
Create a node To create a new node, press the node class you want in the node toolbar and drag it to the location you want in the diagram. You can then type a title into the new node.

Edit a node title To edit the title of a node (in edit mode) click it once to select the node; then click its title. (Pause momentarily between mouse clicks to prevent them being interpreted as a double-click, which would open the node’s **Object** window.) Then type in a new title to replace the old one. Or click a third time to put a cursor into the existing title where you can add text.

Click a node once to select it, showing its handles — small black squares at its corners:



You can edit the title when the node looks like this



The node is resized to fit the text

After editing the title, click outside the node (or press *Tab* or *Alt-Enter*) to accept the new title. If the node is too small for the title text, it expands the node vertically to fit. It can accept a title of up to 128 characters, but it's usually best not to have titles longer than about 40 characters.

Identifiers and titles Every node has a unique *identifier* of up to 20 characters. Formulas in the definition of a variable or function refer to other variables or functions by their identifier. The title is usually longer. It is used for node labels in the diagram and other places for your convenience as the model builder or end user.

By default, when you create a node, it forms an *identifier* for the node from its title, consisting of the first 20 characters of the title, using underscore '_' to replace any character that is not a letter or number. If the first character is not a letter, it substitutes 'A', because identifiers must start with a letter.

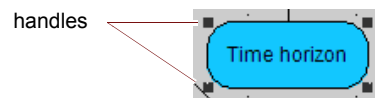
Identifiers must be unique, unlike titles. So, if by chance there already exists an object with the same identifier, it will append a number to the new identifier to keep it unique.

If you want, you can edit an identifier directly in the **Object** window or attribute panel, like any other user-editable attribute.

If you edit the title again, by default, it will ask if you want to change the identifier to match the new title. Usually, it's simplest to have them match. But, sometimes you may want to retain the original identifier. You can switch off this default behavior by unchecking **Change identifier when title changes** in the **Preferences** dialog from the **Edit** menu. (See "Change identifier").

When you, or Analytica, change an identifier, it automatically updates any definitions that refer to it to use the new version, to keep the model consistent.

Select a node To select a node, single-click 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, press and hold the *Shift* key while selecting the nodes. 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.

Move a node To move a node, click the mouse in the node (not on a handle), and drag the node to where you want it.

You can also adjust node positions using the arrow keys (up, down, left, right). By default, each *arrow* press moves the node by 8 pixels. If you uncheck **Snap-to-grid** in the **Diagram** menu, each *arrow* press moves it by a single pixel.

Move a node to another module Simply drag the node onto the module until the module becomes highlighted. When you release the mouse button, the node moves into the module. It will have the same location in the diagram of the new module that it previously had in the old one.

Alternatively, double-click the module to open its diagram window. Move the diagram windows so both you can see both the node and the new diagram. Then drag the node to the desired location in the new diagram.

- Change the size of a node** Click the node to show its handles. Then drag a handle until the node is the size you desire. By default, it fixes the center of the node at the same location, and expands or contracts its four corners. This keeps node centers aligned with the grid. If you want to move one corner, leaving the opposite corner fixed, uncheck **Resize Centered** in the **Diagram** menu.
- Delete a node** Select the node(s) and choose **Clear** from the **Edit** menu, or press the *Delete* key. It will ask you to confirm your intention because deleting cannot be undone. Sometimes it is better to create a module and title it Trash. (There is a Trash library with a suitable icon.) Then you can drag nodes into it — and still retrieve them just in case.
- Cut, copy and paste nodes** You can use the standard **Cut** (*Control+x*), **Copy** (*Control+c*), and **Paste** (*Control+v*) commands from the Edit menu on one or a collection of nodes. Initially, the copies are identical except it appends a number to the identifiers to make them unique.

If you cut a node, you can paste it just once. If you copy a node you can paste it as many times as you wish.

- Duplicate nodes** Select the node(s) and choose **Duplicate Nodes** from the **Edit** menu (or press *Control+d*). This is equivalent to using **Copy** and **Paste**, without writing to the clipboard. Duplicating a node creates a new object identical to the original, adding a number to its identifier to make it unique and located below and to the right.

Duplicating a set of nodes retains the same relationships among the duplicated nodes as exists among the origin nodes. For example, suppose you have three variables:

```
Variable X := 100
Variable Y := X^2
Variable Z := X + Y
```


If you duplicate Y and Z, but not X, you will get two new variables:

```
Variable Y1 := X^2
Variable Z1 := X + Y1
```

Note that it appends "1" to the identifiers to make them distinct from their original nodes. And that the definition of **z1** refers to the unduplicated **x** and the duplicated variable **y1**.

Drawing arrows

Use the arrow tool to draw or remove arrows (influences) between variable nodes. Drawing an arrow from variable or function **a** to **b** puts **a** in the list of **inputs** of **b**. This makes it conveniently available to select from the inputs menu when creating or editing the definition of **b** (see "Creating and Editing Definitions").

- Draw an arrow** To draw an arrow, first click the arrow icon  in the toolbar to select the arrow tool. In arrow mode, the cursor changes to this arrow icon when over a diagram window.

1. Drag from the origin node (which highlights) to the destination node (which also highlights).
2. Release the mouse button, and it draws the arrow.

To speed up drawing arrows from multiple nodes to a single destination, select all the origin nodes. Then drag from any origin node to the destination node. When you release the mouse, it draws arrows from all the origin nodes.

Tip Some arrows are hidden. They do not appear even when you try to draw them. For example, by default arrows to and from indexes and functions are not shown. See “Diagram Style dialog” on page 81 and “Node Style dialog” on page 82 to change these settings.

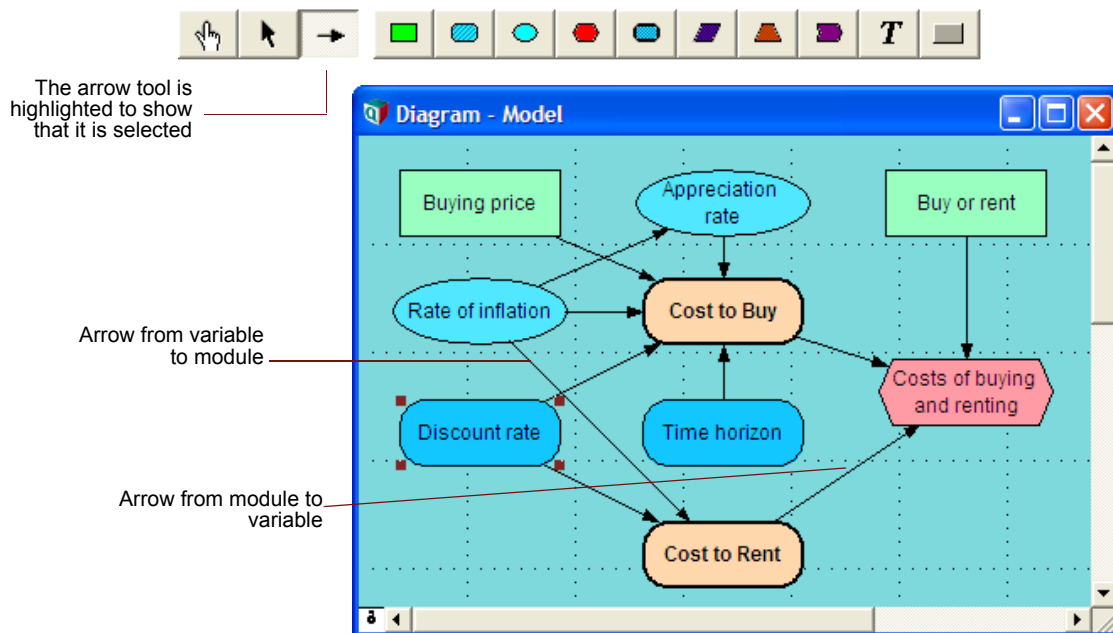
- To remove an arrow**
- Click the arrow to select it, then press the *Backspace* or *Delete* key, or
 - Just redraw the arrow from the origin node to the destination node. If the origin variable is used in the definition of the destination, it will ask if you really want to remove it.

Tip When you enter or edit a definition, Analytica automatically updates the arrows into the variable to reflect those other variables that it mentions (or does not mention). See “Creating or editing a definition”.

Influence cycle or loop An *influence cycle* occurs when a variable A depends on itself directly, where $A \rightarrow A$, or indirectly so that the arrows form a directed circular path, e.g., $A \rightarrow B \rightarrow C \rightarrow A$.

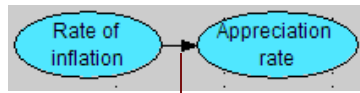
If you try to draw arrows that would make a cycle, it will warn and prevent you. The exception is if at least one of the variables in the cycle is defined with the Dynamic function, and contains a time-lagged dependence on another variable in the cycle, shown as a gray arrow (see Chapter 17, “Dynamic Simulation”).

Arrows linking to module nodes When there are arrow between variables in different modules, they are reflected by arrows to and from the module nodes:



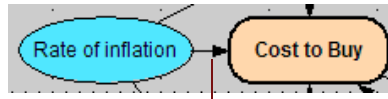
Arrows between variable and module nodes:

Arrow from variable node to variable node



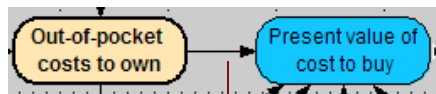
Indicates that the target variable depends on the origin variable.

Arrow from variable node to module node



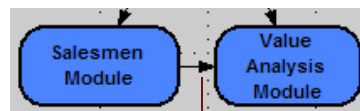
Indicates that at least one variable in the target module depends on the origin variable.

Arrow from module node to variable node



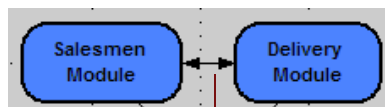
Indicates that the target variable depends on at least one variable in the origin module.

Arrow from module node to module node



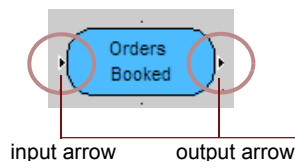
Indicates that the target module contains at least one variable that depends on at least one variable in the origin module.

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.

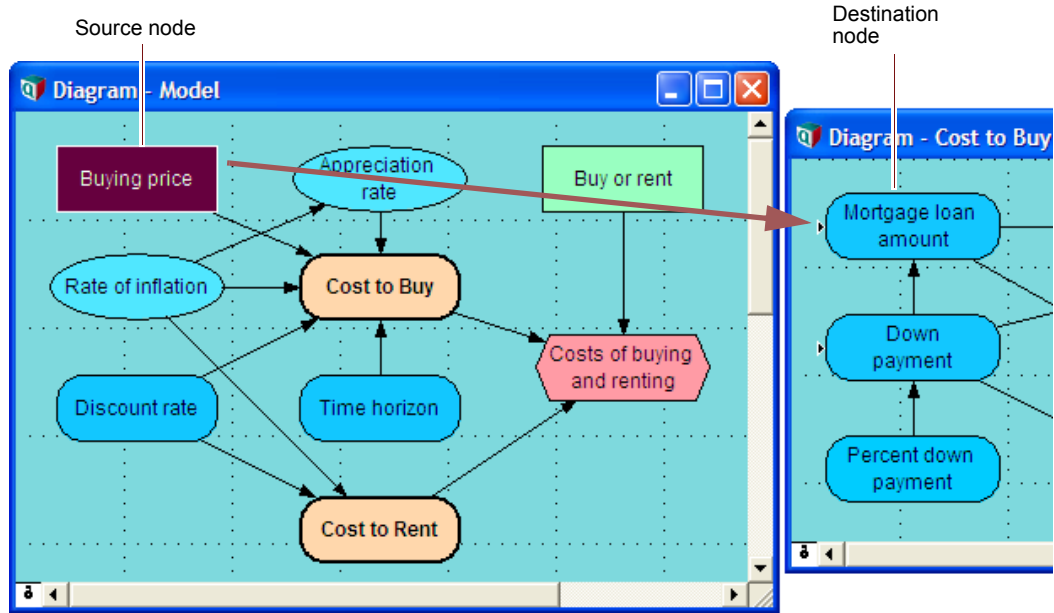
Small arrowhead to the right or left of a variable node




Indicates that the variable has a remote input or output — a variable that is not inside the displayed variable's module (see "Seeing remote inputs and outputs").

How to draw arrows between different modules

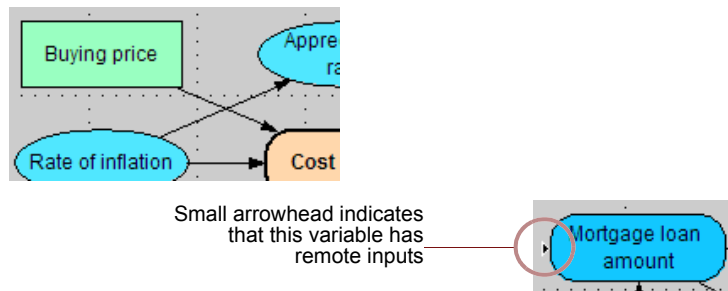
You may draw arrows between variables in different modules, either directly (if both diagram windows are visible on screen, or indirectly. Here a several methods. Suppose you want to draw an arrow from the variable **Buying price** to the variable **Mortgage loan amount** in another module:



Draw arrow across windows

1. Open the two **Diagram** windows and arrange them so that both origin and destination nodes are visible on screen at the same time.
2. In arrow mode , press on the origin node, **Buying price**, so that it highlights. Drag an arrow to the destination node, **Mortgage loan amount**, which also highlights. Release the button.

If, as in this case, the destination module appears in the origin diagram, the arrow points from the origin node **Buying price** to the destination module **Cost to Buy**; a small arrowhead appears on the left edge of destination node **Mortgage loan amount** showing that it has an input node from another diagram:



Move nodes to same diagram to link them

A second method is to move one of the nodes into the diagram containing the other. Then you simply draw an arrow between them. Finally, you move the node back to the diagram it came from. This may be convenient if you have large diagrams and a small screen so that they are hard to arrange so that both nodes are visible in different diagrams.

Copy or type the origin into destination definition

Copy the identifier of the origin variable, open the definition of the destination variable, and paste it in. (See “Creating or editing a definition”). When the definition is complete and accepted, it will automatically draw the arrows to reflect the relationships.

Make an alias node in the other diagram

If the origin node and destination module are in the same diagram, you can draw an arrow directly between them. This will make an alias node of the origin in the destination

diagram. Then you can simply draw an arrow from the alias to the destination node. You can use a corresponding method when the origin module and destination node are in the same diagram. See next section for more.

Alias nodes

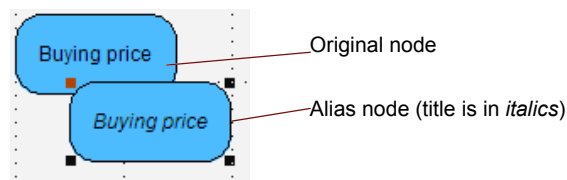
An *alias* is a copy of a node, referring to the same variable, module, or other object as the original node. It's often useful to display an alias node in a different module than its original node. For example, if module **m1** contains variable **x** and its inputs, and **x** has outputs in another module **m2**, it is often useful to add an alias of **x** in **m2** to display the influence of **x** on its outputs explicitly. This makes it easy to draw arrows from **x** to or from other variables in **m2**.

A variable or other object may have only one original node, but an unlimited number of alias nodes.

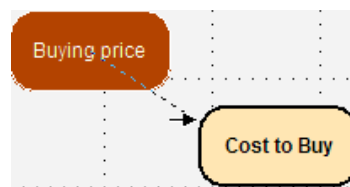
Tip An alias node is identified by its title being shown in *italics*.

You can create an alias directly with the Make alias command, or indirectly by drawing an arrow to or from a module node:

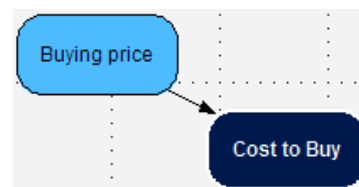
Make Alias command Select the original node. Then choose the **Make Alias** option from the **Object** menu (or press *Control+m*). The alias node appears next to the original node. You can then drag it into another module.



Draw arrow between variable and module Draw an arrow from the original node to a module node, or from a module node to the original node. This creates an alias in the module. For example, draw an arrow from the variable **Buying price** to the module **Cost to Buy**:



It displays an arrow between the nodes:



Open up the module `cost to Buy` to see the new alias:

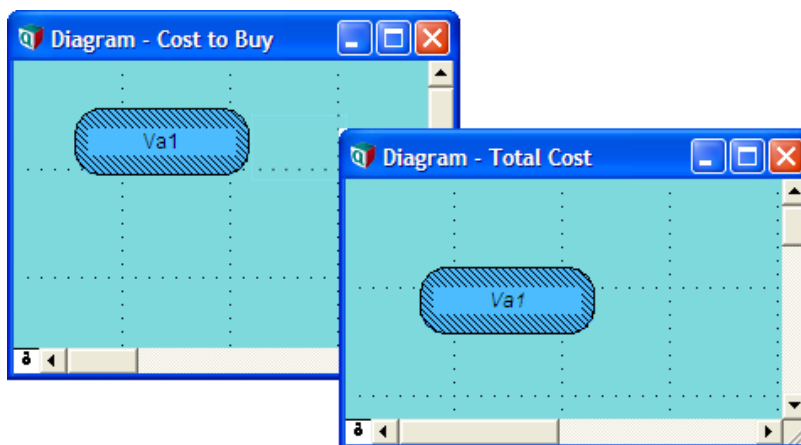


Draw arrow between two modules

If you draw an arrow from one module node `Cost to Buy` to another module `Total Cost`,



it creates a new variable node with a default name, such as `va1`, in the first module `Cost to Buy`, with an alias of `va1` in the second module `Total Cost`.



An alias is like its original

An alias looks and behaves like its original node, except the fact that its label is in *italics*: You can select it, double-click it to open its **Object** window, move, resize, edit its label, and draw arrows to or from it, just like any other node. The alias and original show the same title: If you edit the title in one of them, it automatically changes in the other.

How alias and original can differ

On the other hand, the properties of the *node* — rather than the *object* that it depicts — may differ between the original and its alias: You can modify one node's location (obviously) and size, its color (using the Color palette), and its styles (see “Node Style dialog”).

Tip

If an alias and its original node are in the same diagram, it displays any arrows to or from only the *original* node, not the alias. If the alias is in a different module, it displays arrows connecting it to other nodes in that module, as they would be displayed if it were the original node.

Input and output nodes are aliases

Input and output nodes are kinds of alias nodes that have special node style properties. See “Using input nodes”page 128 and “Using output nodes”.


To edit an attribute

You can edit most attributes of an object directly in the **Attribute** panel (see “The Attribute panel”) or in the **Object** window (see “The Object window”). User-editable attributes include identifier, title, description, units, and definition. See next section on how to change class. Some attributes you cannot edit because they are computed, including inputs, outputs, and value.

To edit an attribute, first display it in the Attribute panel or **Object** window for the object, and make sure you are in edit mode. Then:

1. Click in the *Attribute* field. A blinking text cursor and dotted outline around the attribute indicate that the attribute is editable.
2. Use standard text-editing methods to edit it: type, copy and paste, and use the mouse to select text or move the cursor.
3. To save the changes, click anywhere outside the *Attribute* field, press *Enter*, or display another attribute.

Cancel and undo edits

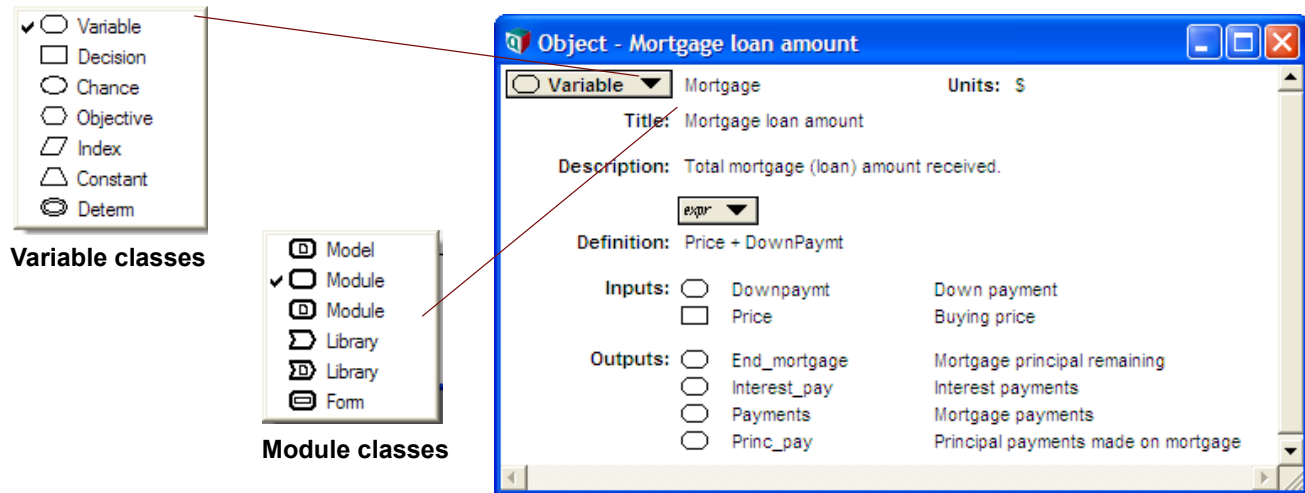
To cancel changes while editing an attribute, press the *Esc* (escape) key to revert to the previous version. Except when editing a definition, click  to cancel changes. To cancel changes *after* you have just made and accepted them, select **Undo** from the **Edit** menu (or press *Control+z*).

Attribute changes

All displays of an object use its same attributes: So any change to an attribute will affect all views that display that attribute. For example, any change to a title appears in other diagram nodes, object windows, or result views referring to that object by title. Any change to a definition will cause redrawing arrows to reflect any changes in dependencies.

To change the class of an object

You may press on the **class** of a variable or module in an object window or attribute panel to open a popup menu. The options depend on whether the node is a variable or a module:







To change class, just select another option from the menu. The shape of the node and other class-dependent properties will change automatically.

Tip You cannot change the class of a function, and you cannot change a variable into a module, or vice versa.



For more, see “Classes of variables and other objects”.

Module Subclasses

All modules contain other objects, including sometimes other modules. There are several different subclasses of module:

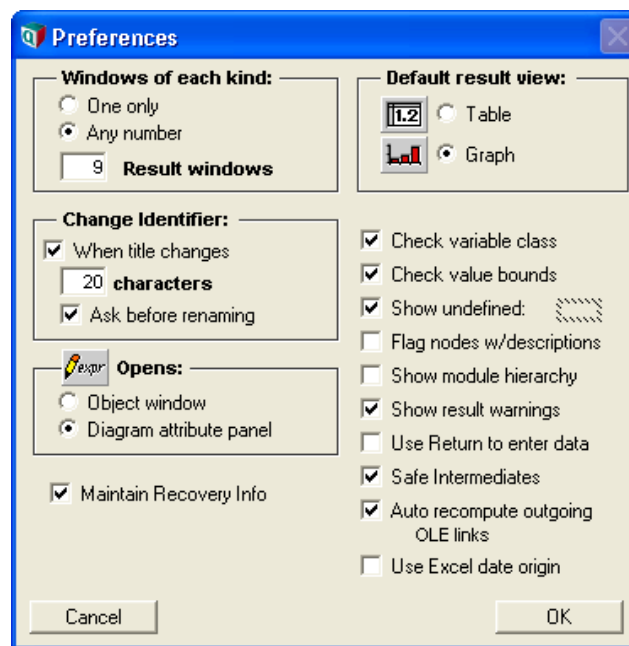
-  **Model:** is usually the top module in a module hierarchy, and is saved as a separate file (document with extension .ANA). Any nondefault preferences (see “Preferences dialog”), uncertainty options (see “Uncertainty Setup dialog box”), and graph style templates are saved with the model, but not other module types.
-  **Module:** A collection of nodes that are displayed in a single diagram. A standard module contains a set of other nodes, and is usually part of the module hierarchy within a model or other module type.
-  **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”.
-  **Library:** A module that contains functions and sometimes variables. Read-in libraries are listed in the **Definition** menu below the built-in libraries, with a hierarchical sub-

menu listing the functions they contain, giving easy access. See “Libraries” on page 345.

-  **Filed library:** A library 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”
-  **Form:** A module containing input and output nodes. You can easily create input and output nodes in a form node by drawing arrows from their original node to the form (for inputs) or from the form to the variable for outputs. See Chapter 9, “Creating Inputs and Outputs”.

Preferences dialog

Use the **Preferences** dialog to inspect and set a variety of preferences for the operation of Analytica. All preference settings are saved with the model. To open the **Preferences** dialog, 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”).

- 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” 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**

These radio buttons control where you view the definition of a selected object, when you click  in the toolbar, press *Control+e*, or when you choose to edit a variable from a warning message:

- Object window* Open the **Object** window and select the definition text (see “The Object window”).
- Diagram Attribute panel* Open the **Attribute** panel on the appropriate **Diagram** window and select the definition text (see “The Attribute panel”).

Maintain Recovery Info

When this checkbox is checked (the default), Analytica saves each change to a recovery file, starting from the last point at which the model was saved. If the application terminates unexpectedly due to a software or hardware problem, the next time you start Analytica, it detects the recovery file and displays a dialog offering to resume the model where you left off, including all changes.

The only reason to switch off this option is when you are editing huge edit tables, in which case, this feature may slow down editing and consume significant disk space for the recovery file.

Unlike the other preference settings, this is stored as a user setting, and is not stored with the model.

Tip

Even when *Maintain Recovery Info* is checked, we recommend you save your model at frequent intervals.

Default result view

Select the radio button to specify which view you prefer as the default when you first display the **Result** window for a variable (see “The result window”).



Display result as a table.



Display result as a graph.

If you change the view in a result window, it will use that view next time you open that result.

Checkboxes

Check Variable class

Display a warning if:

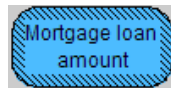
- A variable whose class is not Chance contains a probability distribution.
- A constant depends on another variable (other than indexes to an edit table).
- An index has a value that is not a one-dimensional array, or is an array with another index.

Check value bounds

Evaluate check attributes for variables that have them. See “Automatic checking for valid values”.

Show undefined

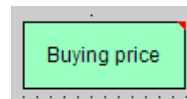
Nodes without a valid definition display with cross-hatching:



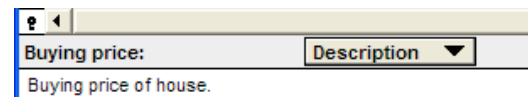
Node is filled with diagonal pattern: the definition is missing or is syntactically incorrect

Flag nodes w/descriptions

Show a red triangle in their upper right corner of nodes that have text in their description attribute:



Node is flagged with a red triangle to indicate that it has a description.

*Show module hierarchy*Show a hierarchy bar at the top of each **Diagram** window showing its nesting level. See “Show module hierarchy preference”*Show result warnings*

If checked, it stops evaluation and shows a warning message, when it encounters a warning condition. If unchecked, it continues without displaying a warning.

*Use Return to enter data*A standard MS 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 checked, these are reversed, with *Enter* or *Alt-Return* starting a new line and *Return* completing the entry of data.*Safe Intermediates*

Analytica ensures that all intermediate arrays generated during evaluation are fully rectangular. By default this is checked. If unchecked, some large models — especially those using dynamic simulation — run faster, sometimes dramatically so. Very occasionally, unchecking can cause incorrect results. If speed is an issue, compare results with this box checked and unchecked. If the values are the same, uncheck this checkbox to improve performance.

Auto recompute outgoing OLE links

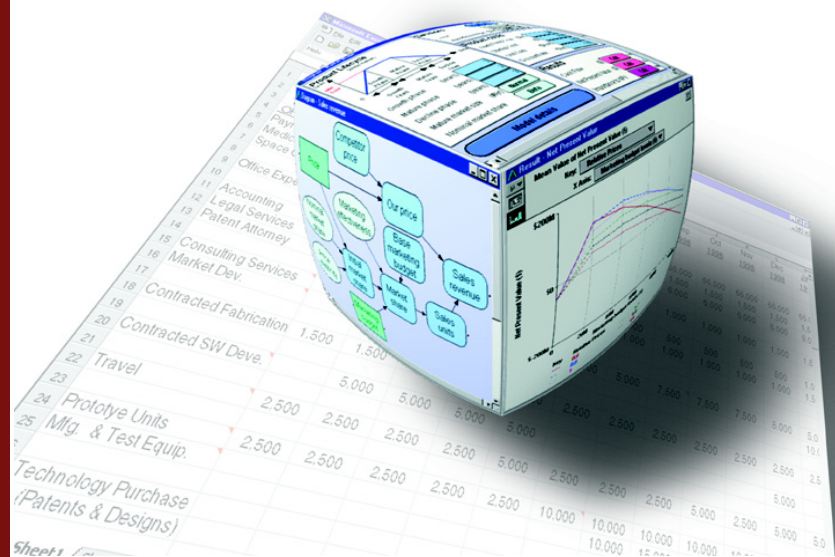
Analytica automatically recomputes and updates OLE-linked tables whenever model changes affect them. With large models, it is sometimes best to uncheck this box to avoid immediate time-consuming recomputation after each small change. See OLE linking in “Using OLE to link results to other applications”.

Chapter 5

Building Effective Models

This chapter shows you how to build models that are:

- Focused
- Simple
- Clear
- Comprehensible
- Correct



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 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.

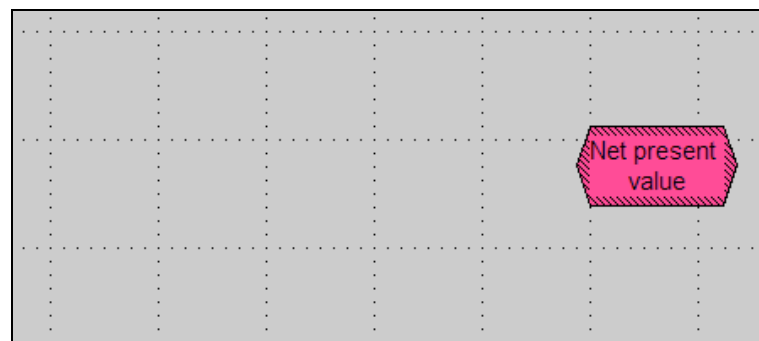
These guidelines have been distilled from many years of experience by master modelers, using Analytica and a variety of other modeling software. However, they are general guidelines, not rules to be adhered to absolutely. We suggest you read this chapter early in your work with Analytica and revisit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

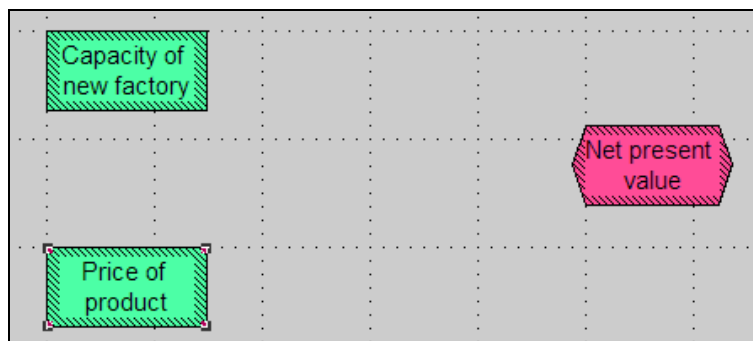
For an excellent guide on how to identify and structure objectives, see *Value-Focused Thinking* by Ralph Keeney (Appendix G, “Bibliography”).

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.

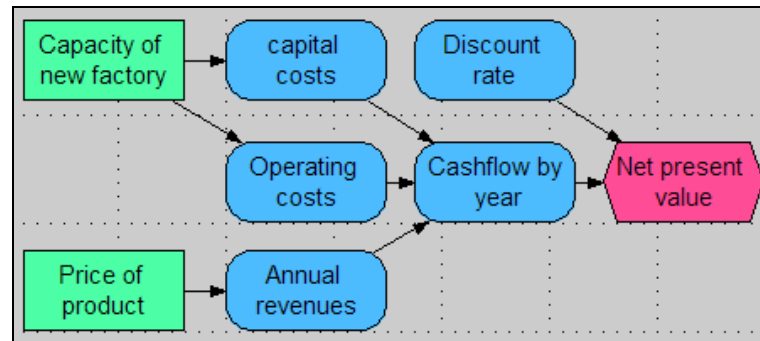
A **decision variable** is one that the decision maker can affect directly — which computer to buy, how much to bid on the contract, which medical treatment to choose, when to start construction, and so on. Occasionally, people want to build a model just for the sake of furthering understanding, without 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 create 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

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 be more accurate.

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

Reuse and adapt existing models

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. You can also check the Lumina wiki for Analytica libraries, templates, and example models (<http://lumina.com/wiki>).

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

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, or may not apply to all conditions. It is important to test and evaluate your model to make sure it 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 descriptions, 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 reports 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 16, “Statistics, 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 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 as you build

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 `NR123`.

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).
- Start with a deterministic model, then add probabilistic inputs to make the model probabilistic.

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), and Analytica will propagate the needed changes automatically throughout the model.

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") 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 negligible.

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

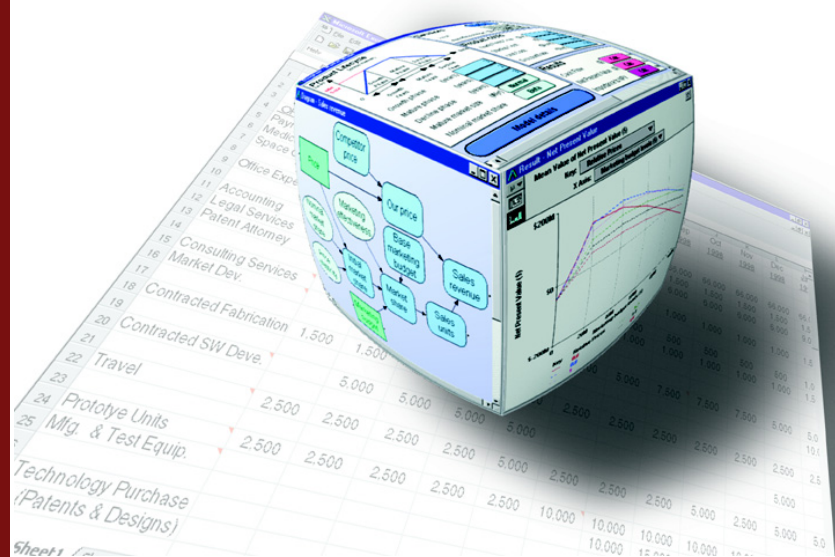
There's no reason that a model must grow successively more complex as you develop it. Sensitivity analysis may reveal that an uncertainty or submodel is just not very important to the results. In this case, consider eliminating it. You may find that some dimensions of a table are unimportant — for example,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

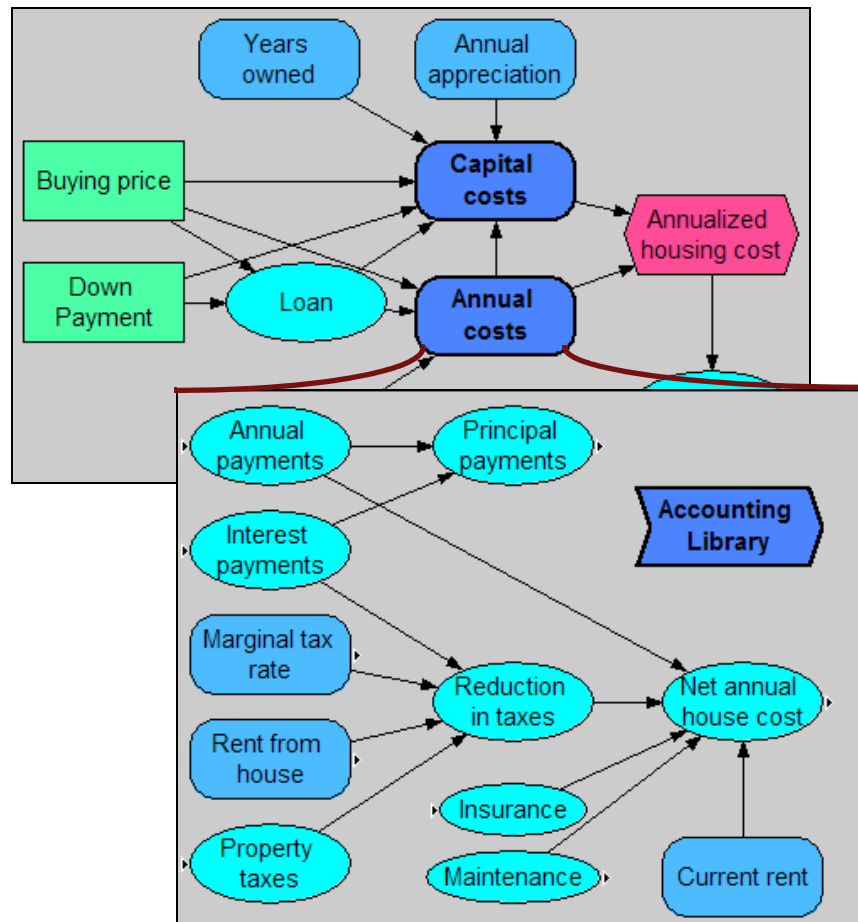
Chapter 6

Creating Lucid Influence Diagrams

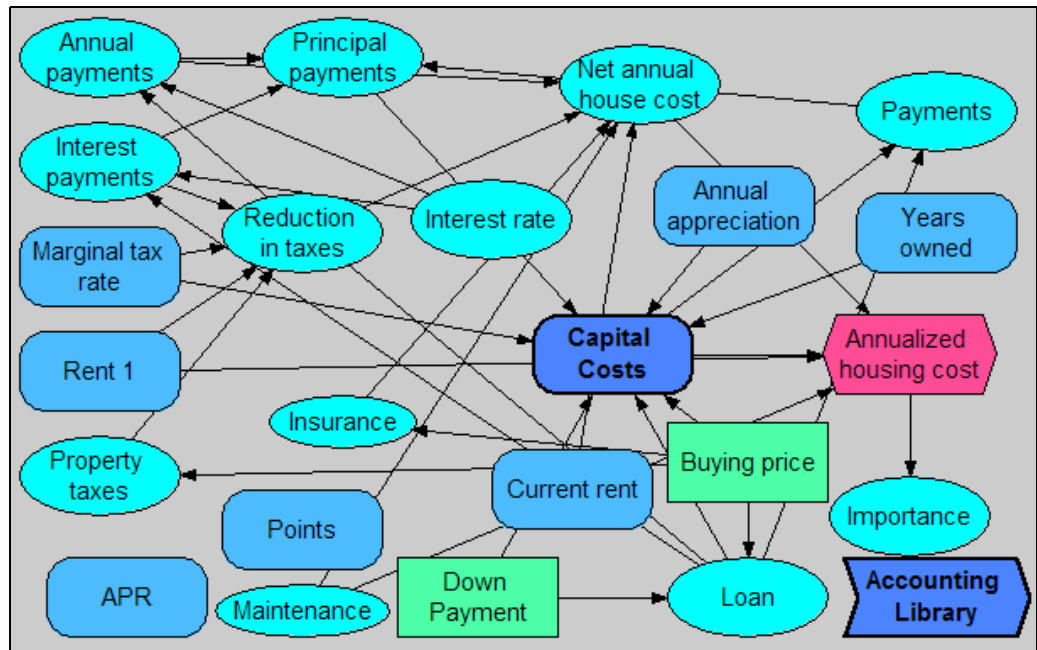
This chapter offers guidelines for creating influence diagrams that are clear and comprehensible by careful arrangement of nodes, well-designed module hierarchies, and judicious use of color. It also describes how to adjust and align nodes, and customize styles for nodes and diagrams. Options include which arrows to show, node sizes, colors, text size, and font family.



Hierarchical influence diagrams can provide a lucid display of the essential qualitative structure of a model, uncluttered by quantitative details.



It is also possible to create impenetrable spaghetti!



Guidelines for creating lucid and elegant diagrams

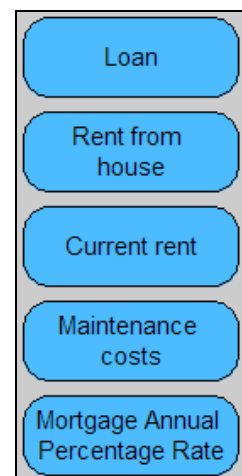
Where aesthetics are involved, rules cannot be hard and fast. You may adapt and modify these guidelines to suit your particular applications and preferences.

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 lower-case) is clearer than all letters uppercase.



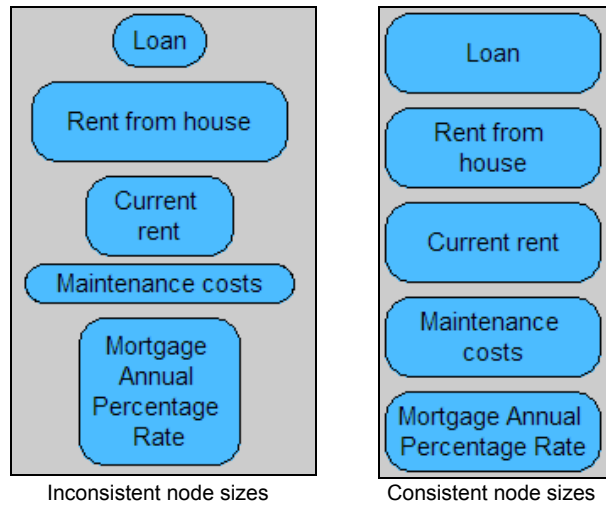
Poor object titles



Good object titles

Use consistent node sizes

Diagrams usually look best if most of the variable nodes are of the same size:



Node sizes will be uniform if you set the default minimum node size in the **Diagram Style** dialog (see “Diagram Style dialog”) large enough so that it fits the title for nodes. When creating nodes, it uses this default size unless the text is too lengthy, in which case it expands the node vertically to fit the text. For more information on how to adjust node sizes see “Adjust node size” on page 75.

To make nodes the same size, select the nodes (*Control+a* selects all in the diagram), and select **Make same size > Both** from the **Diagram** menu (or press = key twice)

Use small and large nodes sparingly

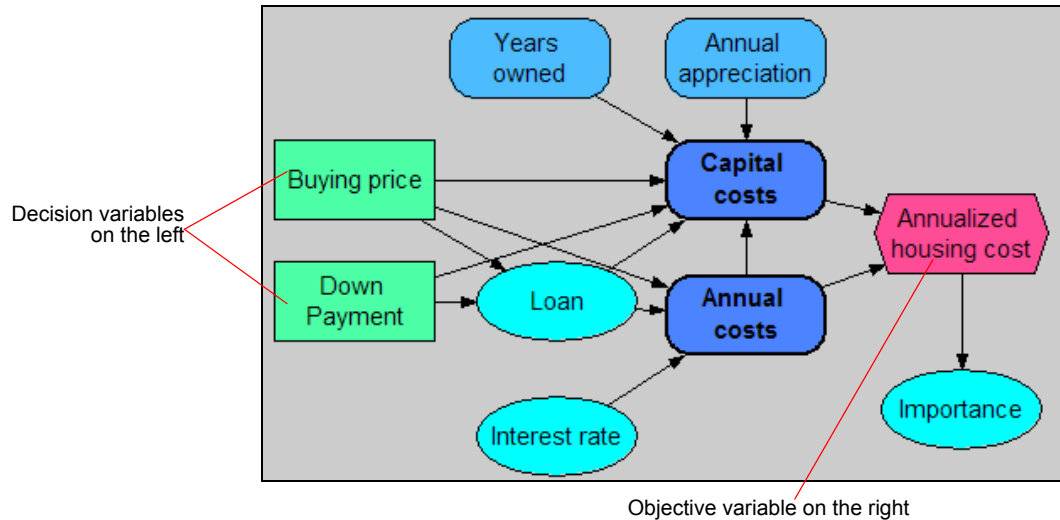
Sometimes, it is effective to make a few special nodes extra large or small. For example, start and end nodes, which may link to other models, often look best when they are very small. Or you may make a few nodes containing large input tables or modules containing the “guts” of a model larger to convey their importance.

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

People find it natural 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 need to let a few arrows go counter to the general flow to reduce crossings or overlaps. In dynamic models, time-lagged feedback loops (shown as gray arrows) may appropriately go counter to the general flow.

1. Or right to left for models in Arabic or Hebrew.

**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: 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.

Arranging nodes to make clear diagrams

Adjust node size

If you have nodes of different sizes, you can make them more consistent by selecting **Adjust Size** (*Control+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.

Selected nodes can also be set to be the same width, height, or size. To set the size of selected nodes to be the same size use the **Make Same Size** submenu in the **Diagram** menu. The options are:

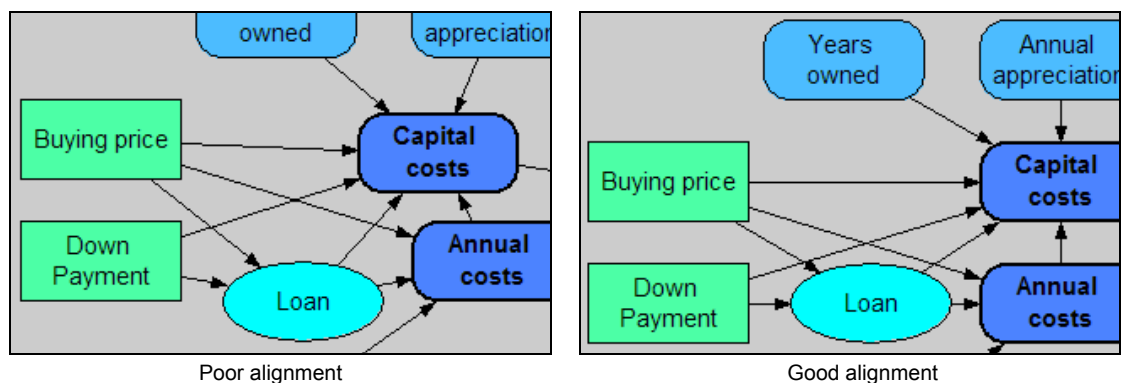
- **Make Same Size Width** — Sets all the selected nodes to the width of the widest node
- **Make Same Size Height** — Sets all the selected nodes to the height of the tallest node

- **Make Same Size Both** — Sets all the selected nodes to the width of the widest node and the height of the tallest node

Align to the grid 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 pixel blocks underlying each diagram makes this easy. When the grid is on (the default), each node that you create or move is centered on a grid intersection. 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 (*Control+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, and you can move the nodes pixel by pixel.

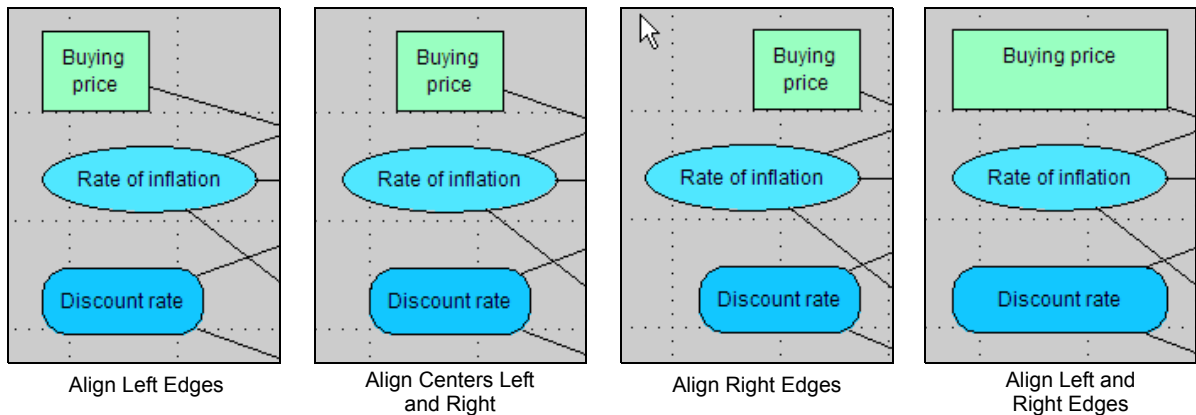


If nodes are not centered on a grid point, re-center them by following these steps:

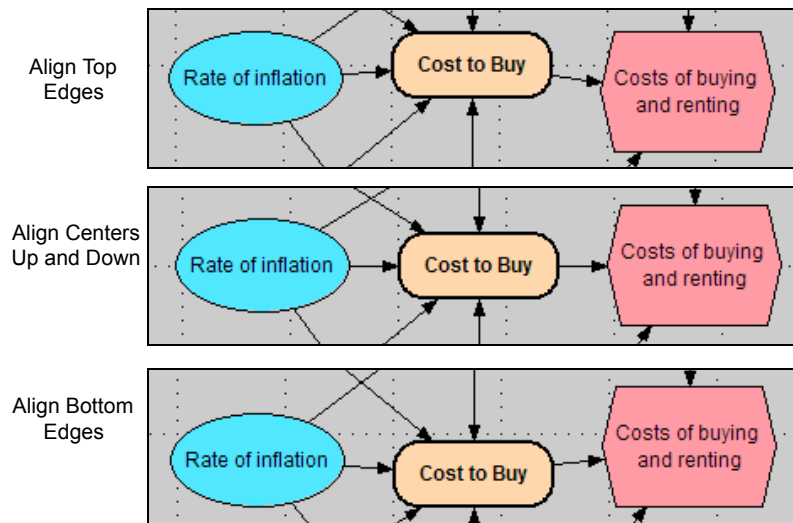
1. Select all nodes in the diagram with the **Select All** (*Control+a*) command from the **Edit** menu.
2. Select **Align Selection To Grid** from the **Diagram** menu.

Align selected nodes To line up selected nodes with each other, use the **Align** submenu in the **Diagram** menu. You can align selected nodes with one another in the following ways:

- Align their left edges
- Align their centers left and right — this aligns their centers horizontally
- Align their right edges
- Align their left and right edges — this makes all the selected nodes the same width and aligns, so that their left and right edges match up. All nodes will be set to the width of the widest node.



- Align their top edges
- Align their centers up and down — this aligns the nodes so that their centers are at the same vertical height
- Align their bottom edges



Distributing nodes To distribute selected nodes evenly, use the **Space Evenly** submenu in the **Diagram** menu. You can distribute selected nodes so that their centers are evenly spaced vertically (**Space Evenly Across**) or horizontally (**Space Evenly Down**).

Choosing which node is in front By defaults, text and picture nodes are behind arrows, and arrows are behind all other types of nodes (decision, chance, variable, etc.). If nodes overlap, the more recently created node will be on top of the older node. You can change this order by selecting a node(s) and using the **Send to Back** and **Bring to Front** options from the right-click menu.

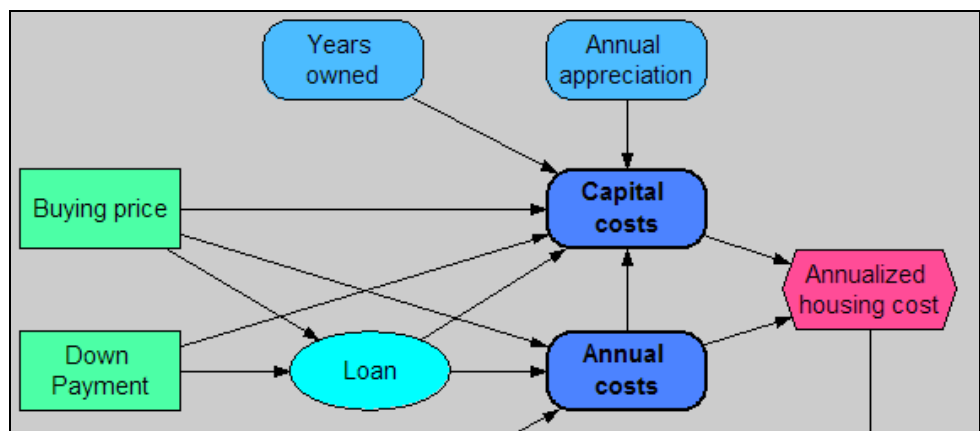
Hide less important arrows Sometimes so many nodes are interrelated that it is hard or impossible to arrange a diagram to avoid arrows crossing each other or crossing nodes. It may be helpful to hide some arrows that show less important linkages. For example, indexes and functions are often connected to many other variables; that's why arrows to and from them are switched off by default.

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

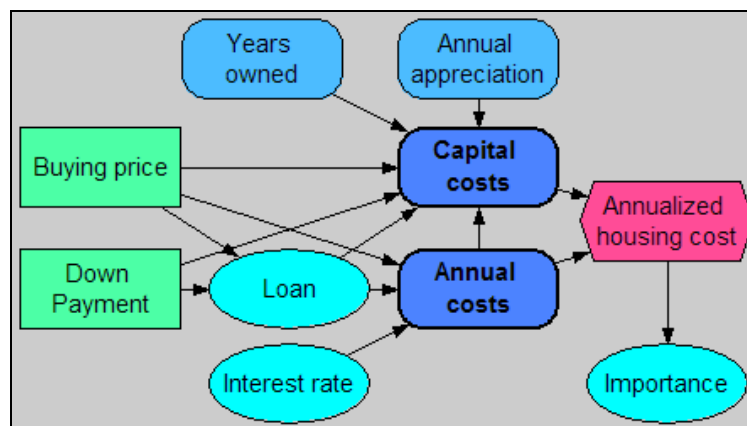
Keep diagrams compact

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, first find a clear arrangement, which minimizes arrow crossing and avoids node overlaps. Then, 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. Finally, you can reduce the window size to fit the diagram.



A spread-out diagram



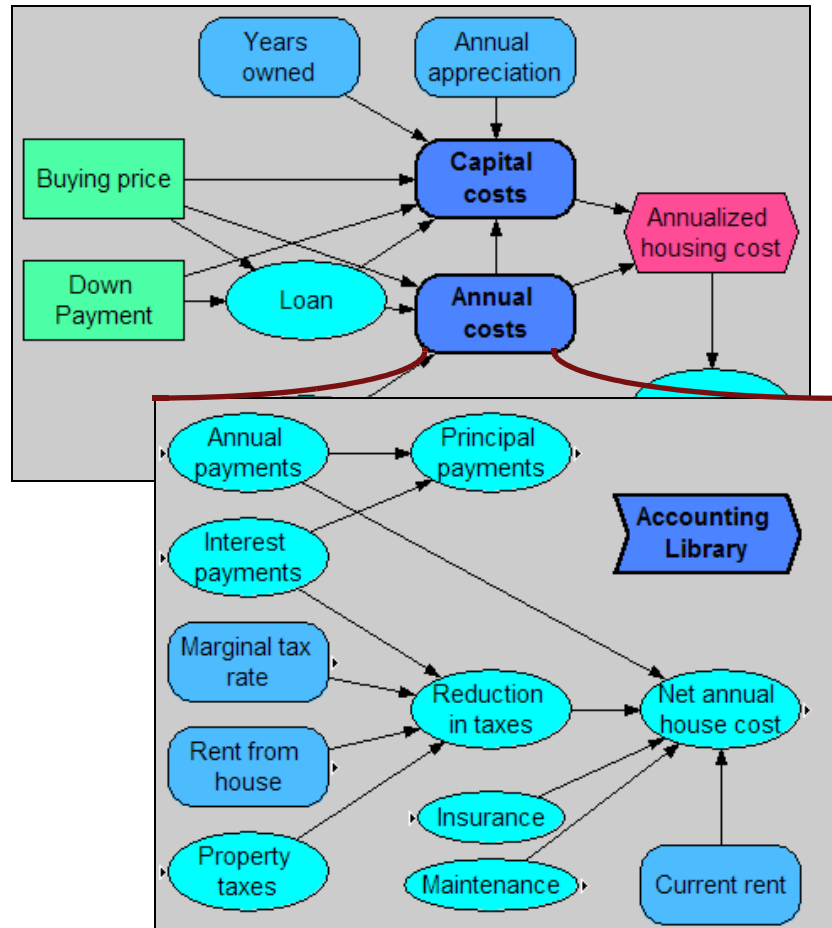
A compact diagram

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 that have 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 10 to 20 nodes per diagram

When creating a hierarchy of diagrams for a model with 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 20 nodes often looks overwhelmingly complicated, unless there are strong regularities in the structure. On the other hand, if modules have fewer than 5 nodes, you need so many modules that it is easy for users to get lost.

The range of 10 to 20 nodes per diagram is a good general goal. But don't feel too constrained by it if a few diagrams are outside this range.

Contrast the module hierarchy in the illustration on page 79 with the spaghetti on page 73. 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 79) than in the model with 24 top-level nodes (page 73).

Color in influence diagrams

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

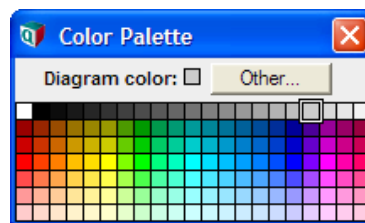
Use colors judiciously

Garish or uncoordinated colors can be distracting. It generally looks messy to have nodes in many different colors. Sometimes it's useful to use color coding beyond the default colors by class of node. For example, you might want to color all input nodes to identify them clearly.

Light colors work best because it's easier to see the black arrows and text over light backgrounds. Analytica's default colors provide a light neutral color for the background and a slightly stronger color for the nodes.

Recoloring nodes or background

1. In edit mode, select **Show Color Palette** from the **Diagram** menu:



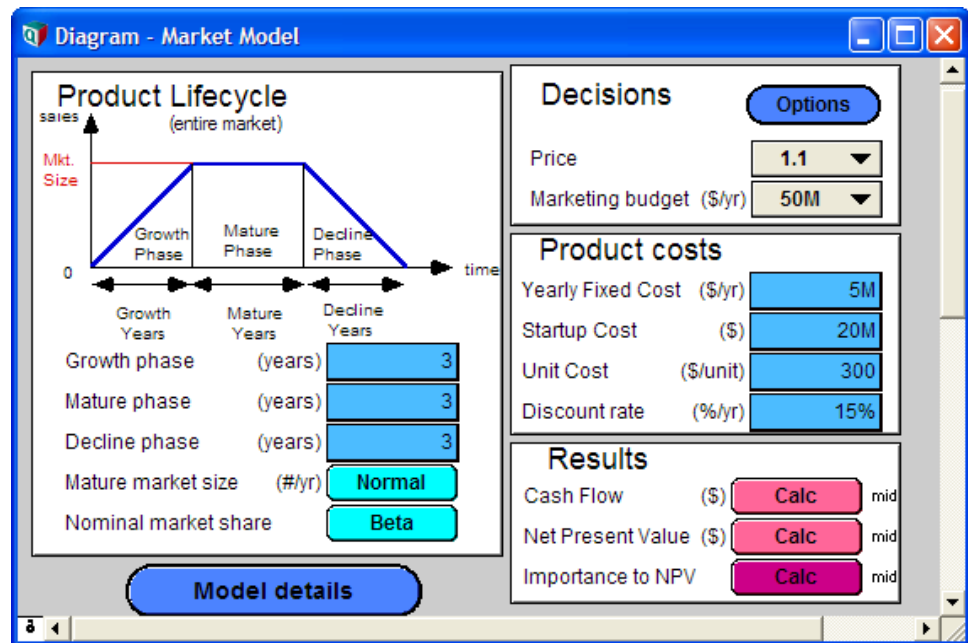
2. Select the node or nodes you want to recolor, or to recolor the background, just click the background. The current color of the node(s) or background appears in the single square at the top of the color palette.
3. Click a color square to apply the new color to the nodes or background.

For a wider range of colors, click **Other** to display a full color chart.

Grouping nodes with a text box

It often improves the look and clarity of a user interface to group related nodes in rectangular boxes with a contrasting color, white in this case:

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



To create a grouping rectangle using a text box:

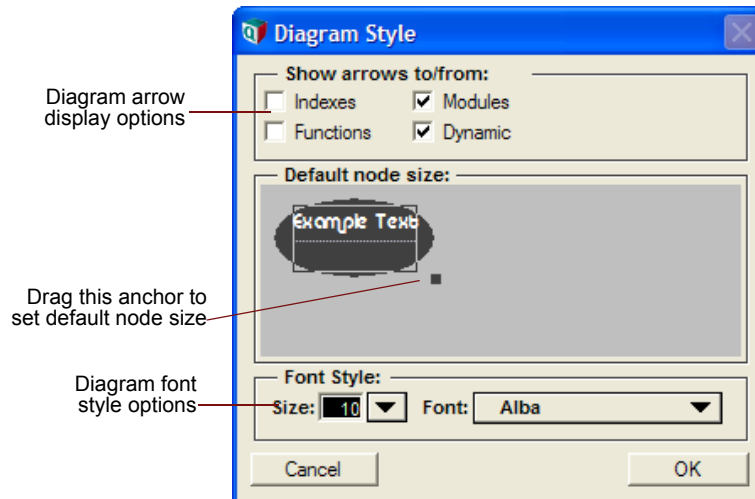
1. With the diagram in edit mode, create a **text node** by dragging from **T** on the node toolbar onto the diagram
2. Type a title into the text node, or leave it blank as desired.
3. Move and resize the node to enclose the group of inputs or outputs. You may find it convenient to deselect the **Resize centered** option from the **Diagram** menu.
4. With the node selected, open the **Set Node Style** dialog from the **Diagram** menu, check the **Border** and **Fill color** options (and **Bevel**, if you like), and click **OK**.
5. Select the **Color palette** from the **Diagram** menu, and click the preferred color for the node, e.g., white.

Usually, text nodes appear behind all other nodes, which is what you want for organizing groups. But if a node is not in the back and is obscuring other items, you can select **Send to Back** from the right-click button menu.

Tip The background color of a diagram also applies to the background color of any modules contained in the diagram — unless you explicitly override the default by setting a different background color for each submodule. Similarly, the color you apply to a module node will also apply to any submodule nodes inside the module — unless you override the default by recoloring any submodule node(s).

Diagram Style dialog

Use the **Diagram Style** dialog to customize the font size and typeface for nodes, whether arrows are displayed for specified node classes, and the node size. To display the **Diagram Style** dialog, select **Set Diagram Style** from the **Diagram** menu.



Show arrows to/from Check the corresponding boxes to display (or hide) arrows that go to and from nodes of each type, **Indexes**, **Functions**, **Modules**, and **Dynamic**. **Dynamic** controls the display of time-lagged dependencies to variables defined with **Dynamic**, usually displayed as gray arrows. See “Dynamic(initial1, initial2..., initialn, expr)” on page 298.

By default, diagrams show arrows to and from modules and dynamic, but not indexes and functions. Showing more arrows can render some diagrams confusing with criss-crossing arrows. But, showing fewer arrows makes important dependencies (influences) invisible. The best balance depends on the model.

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, it tries to make the node this size — if the node title is too large, it expands the node vertically until it fits. It is usually best to size the default to include at least two lines of text at the selected font size. Input and output nodes do *not* use this default; they extend horizontally to fit their text plus field or button.

Font Style To change the default font size, use the menu or type in a font size (in typographic points). Select the default typeface from the font menu.

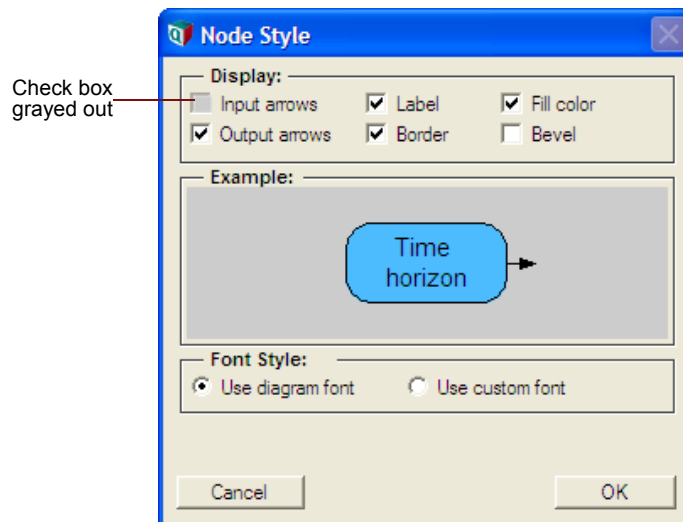
Overriding diagram defaults The **Diagram Style** dialog sets defaults for the diagram and for any modules contained in that diagram. You may override these defaults for particular nodes with the **Node Style** dialog (below), or for a submodule by using the **Diagram style** dialog for the submodule.

Node Style dialog

The **Node Style** dialog lets you customize the display of one or more nodes in a diagram. You can change the typeface and font size, and whether to display the incoming and outgoing arrows, the text label, border, fill color, and bevel, and the text size and font. These options override any defaults set for in the **Diagram Style** dialog.

- To open the Node Style dialog**
1. Select one or more nodes in a diagram.
 2. Choose **Set Node Style** from the **Diagram** menu or the **right-click** menu.
 3. Select the options for which you want to override the default styles.

4. Click **OK**.



Input arrows Check to display arrows into this node.

Output arrows Check to display arrows out of this node.

By default, input and output arrows are not displayed for index and function nodes.

Label Check to display the title in the node. By default, this is checked for all nodes.

Border Check to display a thin black border around the node.

Fill color Check to display the color in the node. Otherwise the node appears transparent, and any nodes or arrows under it will be visible.

Bevel Check to show a bevel effect around the node. By default, this is checked only for button nodes.


By default, text nodes, input and output nodes do not show arrows, border or fill color.

Tip A grayed out check box indicates that this option is not the same for all selected nodes. If you leave it unchanged (gray), each node keeps its current setting. If you change it (on or off), it changes all nodes to the new setting.

Font style To override the default diagram font, select Use custom font. Then you can select the font size and typestyle.

Taking screenshots of diagrams

These are some tips for taking good screenshots of **influence diagrams** and other Analytica windows for use in other documents or printing.

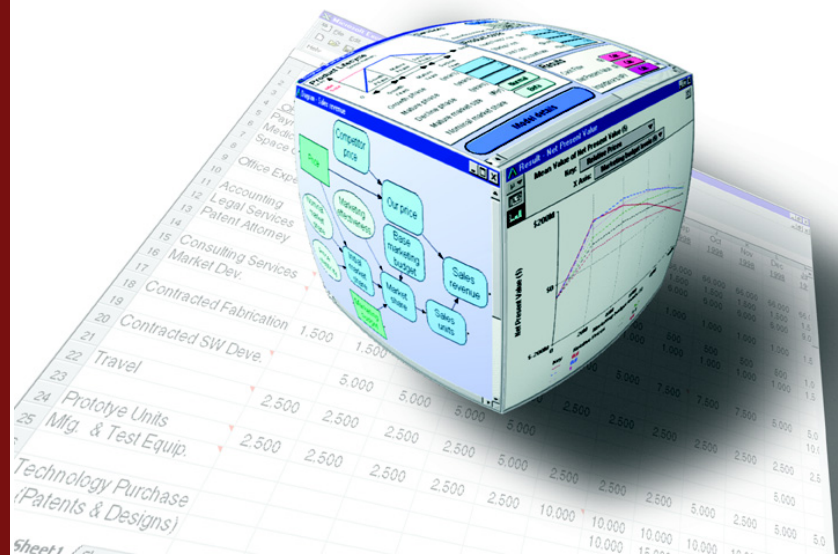
Use browse mode When making screen captures of a **Diagram** window, select browse mode  rather than the edit or arrow mode to switch off the background grid, which makes the diagram clearer.

- Switch off cross-hatching** By default, the nodes of undefined variables show a cross-hatched pattern around the title. To remove this pattern, uncheck **Show undefined** in the **Preferences** dialog from the **Edit** menu (see “Preferences dialog”).
- 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). The print command lets you not print the background color if any.

Chapter 7

Formatting Numbers, Tables, and Graphs

This chapter shows you how to control the display of numbers, including Booleans and dates, in tables and styles and options for graphs.



Number formats

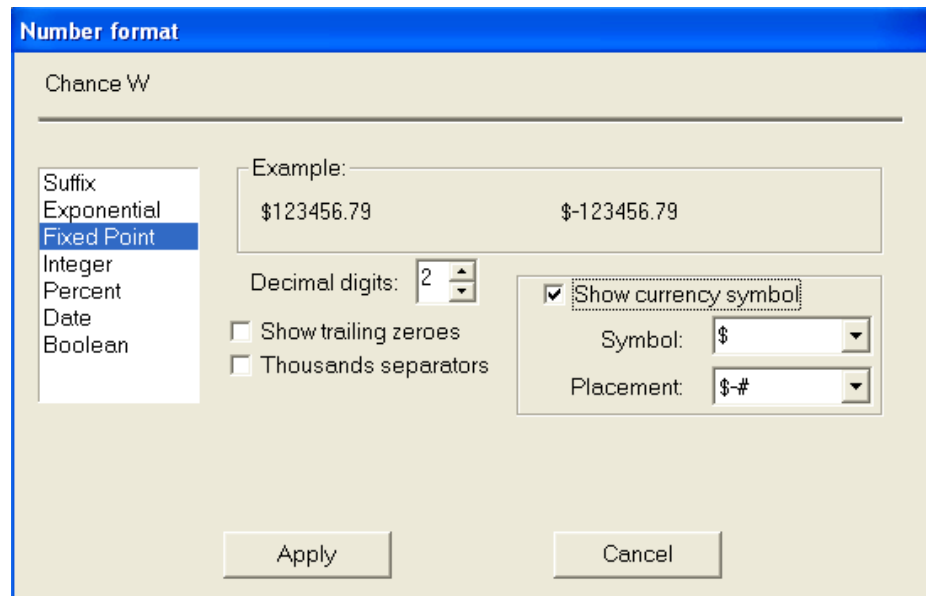
The **Number format** dialog box lets you control the format of numbers to display in result tables and graphs — including dates and Booleans. You can select options like the number of decimal digits, currency signs, and commas to separate thousands. The default number format is *suffix*, which uses a letter following the number, such as 10K to mean 10,000 (where *K* means Kilo or thousands).

The number format of a variable is used wherever the value of that variable appears — in a result table, graph, input or output field. The number format of an Index applies wherever that index is used, including row or column headers of a table, or along an axis of a graph that uses that index.

You may enter a number into an expression or table in any format, no matter what output format it uses — except for dates, where you need to specify a date format, so that it will interpret 10/10/2007, for example, as a date, not two divisions.

To set the number format for a variable:

1. Select a variable by showing its edit table, result table, or graph, or by selecting its node in a diagram. To apply the same format to multiple variables, select their nodes together in a diagram.
2. Select **Number format** from the **Result** menu, or press *Control+b*, to show this dialog:



3. Select the format you want from the list on the left.
4. Select options you want, such as **Decimal digits**, **Show trailing zeroes**, or **Thousands separators**, or **Show currency symbol**.
5. Check the example at the top of the dialog to see if it's what you want.
6. If so, click the **Apply** button.

Format types Choose one of these number formats:

Format	Description	Example
Suffix	A letter after the number specifies powers of ten (see below for details)	12.35K
Exponential	Scientific or exponential notation. The number after the "e" gives the powers of ten.	1.235e04
Fixed Point	A decimal point with fixed number of decimal digits	12345.68
Integer	A whole number with no decimals	12346
Percent	A percentage	12%
Date	Date (see below for details)	12 Jan 2007
Boolean	Displays 0 as False, any other number as True.	True, False

Suffix characters Suffix is Analytica's default format. It uses a conventional letter after each number to specify powers of 10: 12K means 12,000 (*K* for kilo or thousands), 2.5M means 2,500,000 (*M* for Mega or millions), 5n means 0.000,000,005 (*n* means nano or billionths). Here are all the suffix characters:

Power of 10	Suffix	Prefix	Power of 10	Suffix	Prefix
			10^{-2}	%	percent
10^3	K	Kilo	10^{-3}	m	milli
10^6	M	Mega or Million	10^{-6}	μ	micro (mu)
10^9	G or B	Giga or Billion	10^{-9}	n	nano
10^{12}	T	Tera or Trillion	10^{-12}	p	pico
10^{15}	Q	Quad	10^{-15}	f	femto

Tip Note the difference between 'M' for Mega or Million and 'm' for milli (1/1000). This is the only situation in which Analytica cares about the difference between upper- and lowercase. Otherwise, it is insensitive to case (except when matching text values).

Tip In suffix format, it displays four-digit numbers without the 'K' suffix, which looks better for years — e.g., 2010, not 2.010K. For suffix, integer, or fixed point formats, it uses exponent format for numbers too large or small — e.g., numbers larger than 10^9 in integer or fixed point format, or larger than 10^{18} in suffix format.

Maximum precision The maximum number of digits including decimal digits is 15 (14 for fixed point and percent); the maximum number precision is 15 digits (9 for integers).

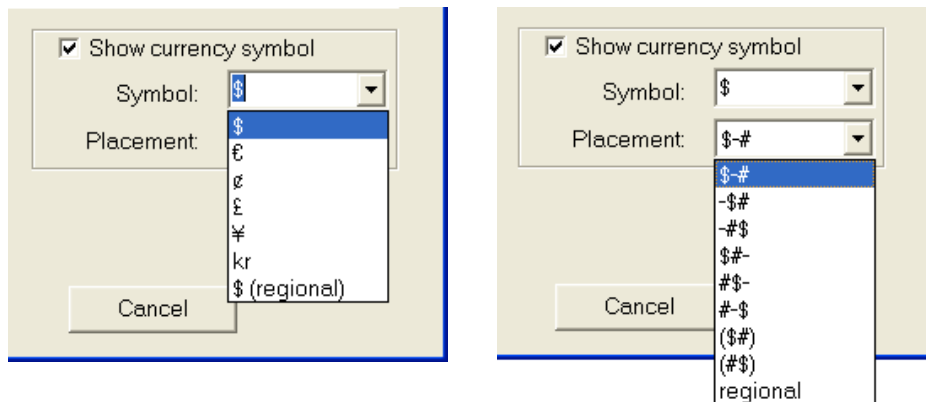
Number format options

Decimal digits The number of digits to show after the decimal point.

Show trailing zeroes Check to show trailing zeroes in decimals, e.g., 2.100 instead of 2.1, when decimal digits are set to 3.

Thousands separators Check to show commas between every third digit of the integer part, e.g., 12,345.678, instead of 12345.678.

Show currency symbol Check to show a currency symbol. Select the symbol and placement from these menus:

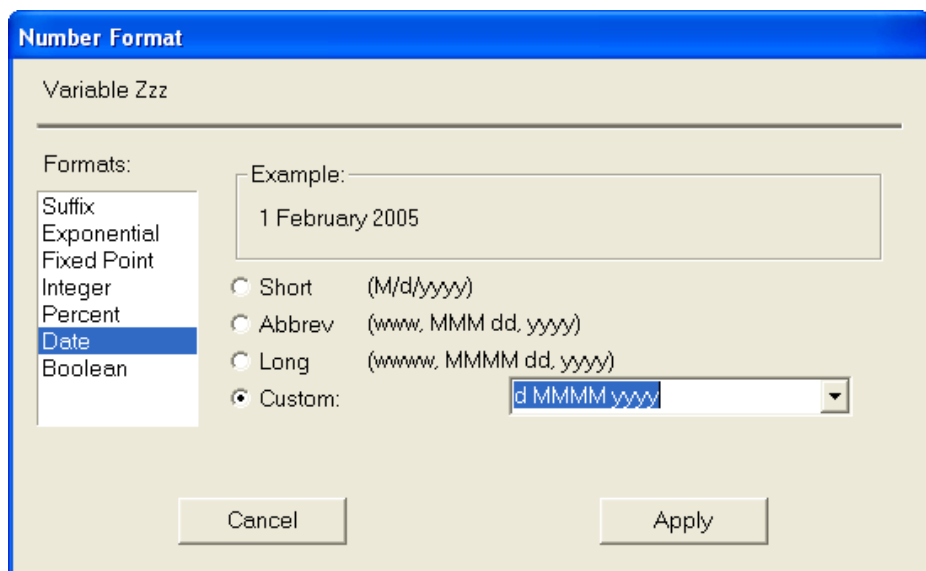


Placement controls the relative location of the currency symbol, e.g., \$200 or 200DM, and whether to use a minus sign -\$200 or parentheses (\$200) to indicate a negative number.

Regional settings If you select the last entry, “regional”, from the **Symbol** or **Placement** menu, it uses, respectively, the regional currency or placement settings set for your computer. You can modify these settings in the **Regional and Language** options available from the Windows Control Panel.

Date formats

A date is a number shown in Date format. The date number represents the date as the number of days since the **date origin**, usually Jan 1, 1904.



The **Date** format in the **Number format** dialog offers these options:

Short: e.g., 8/5/2006

Abbrev: e.g., Aug-5-2006

Long: e.g., Thursday, 05 August, 2006

Custom: Use an existing custom format or set up a new one, for example:

Date format	Displays as
dd-MM-yy	05-8-06
'Q'Q YYYY	Q2 2006
www, d MMM yyyy	Thu, 5 Aug 2006
www, d of MMMM, yyyy	Thursday, 5 of August, 2006

Date format codes Custom date format uses these letter codes, conventional for Microsoft Windows:

Code	Description	Example
d	numeric day of the month as one digit	1, 2,... 31
dd	numeric day of the month as two digits	01, 02,...31
ddd	ordinal day of month in numeric format	1st, 2nd, ..., 31st
dddd	ordinal day of month in text format	first, second, ..., thirty-first
Dddd	capitalized ordinal day of month	First, Second, ... Thirty-first
www	weekday in three letters	Mon, Tue,.. Sun
www	weekday in full	Monday, Tuesday, ... Sunday
M	month as a number	1, 2, ... 12
MM	month as two-digit number	01, 02, ... 12
MMM	month as three letter name	Jan, Feb, ... Dec
MMMM	month as full name	January, February, ... December
q	quarter as one digit	1, 2, 3, 4
YY	year as two digits	e.g., 99, 00, 01
YYYY	year as four digits	e.g., 1999, 2000, 2001

Tip To show literal text within the date, enclose it in quotes, e.g., 'q'q → q2.

Interpreting input dates

If you specify any date format for an input variable or edit table, you can enter dates in any acceptable date format. For example, a variable with a date format, interprets 1/5/2005 as 5 **January**, 2005 on a computer set to **USA region** or 1 **May**, 2005 elsewhere. Without the date format, it would interpret 1/5/2005 as (1 divided by 5) divided by 2005!

Regional and language settings

The language for day and month names and the formats used for **Short** and **Long** dates depend on the regional settings for Windows. In the U.S., you might see a short date as 9/11/2001, but in Denmark you might see 11.9.2001. You can review and change these settings in **Regional and Language options** available from the Windows Control Panel. These apply to Analytica and all standard Windows applications. To modify settings, click the **Customize** button and select either the **Date** tab or **Languages** tab. For example, if you set the language to *Spanish (Argentina)*, a variable with the **Long** date setting, the date displays as:

`StartDate` → **Sábado, 03 de Febrero de 2007**

where

```
Variable StartDate := Makedate(2007, 2, 3)
```

Date numbers and the date origin

Analytica represents a date as a **date number**, that is, the number of days since the *date origin*. By default, the date origin is Jan 1, 1904, as used by most Macintosh applications, including Excel on Macintosh, and all releases of Analytica on Macintosh and Windows up to Analytica 3.1. If you check **Use Excel date origin** in the **Preferences** dialog, the date origin is Jan 1, 1900, as used by default in Excel on Windows and most other Windows software.

With **Use Excel date origin** checked, the numeric value of dates are the same in Analytica and Excel for Windows for dates falling on or after 1 Mar 1900. Because of a bug in Excel, in which Excel incorrectly treats Feb 29, 1900 as a valid day (1900 was not really a leap year), dates falling before that date do not have the same numeric index in Analytica as they do in Excel.

When using models containing dates or date functions from Analytica releases 3.1 or earlier, you should keep **Use Excel date origin** unchecked. If you want to paste or link values from Excel or other Windows software to or from Analytica, you should check this option.

Range of dates

Analytica can handle dates from 1 CE to well beyond 9999 CE (CE means Common Era or Christian Era, and is the same as AD.). Dates earlier than the date origin are represented as negative integers. Dates use the Gregorian calendar, so years divisible by 4 are leap years and those divisible by 100 are not leap years, except those divisible by 400 which are leap years.

Date arithmetic and functions

You can simply add an integer n to a date to get the date n days ahead. See “Date functions” for **MakeDate()**, **DatePart()**, **DateAdd()**, and **Today()**.

Multiple formats in one table

Usually, the same number format applies to all numbers in a table (except its index values in column or row headers, which use the format set for the index variable.) Sometimes, you may want to use different formats for different rows (more generally, *slices*) of a table. You can do this if you define the table as a list of variables, for example:

```
Index Years := 2007..2012
Variable DollarX := Table(Years)(...) { Formatted as dollars }
Variable PercentX := DollarX/40JK { Formatted as percent }
Variable MultifomatX := [DollarX, PercentX]
MultifomatX →
```

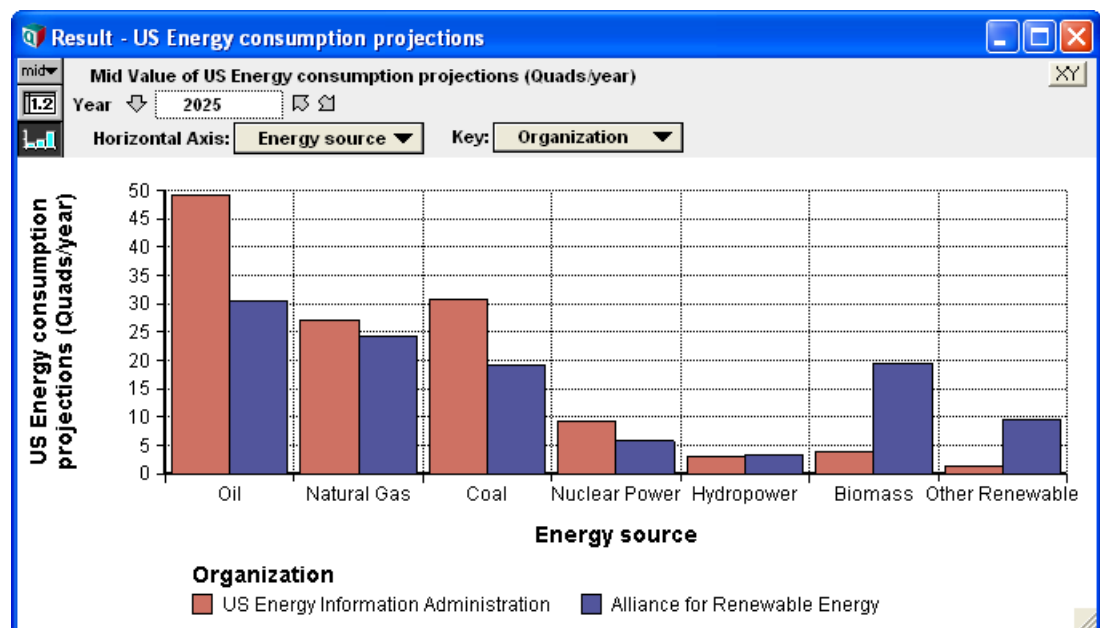
	2007	2008	2009	2010	2011	2012
DollarX	\$10,432	\$11,234	\$12,034	\$17,091	\$12,234	\$21,201
PercentX	26.08%	28.08%	30.09%	42.73%	30.59%	53%

This table uses the number format set for each variable responsible for a row here — as long as you don't override their settings by setting a format for **MultiFormatX**.

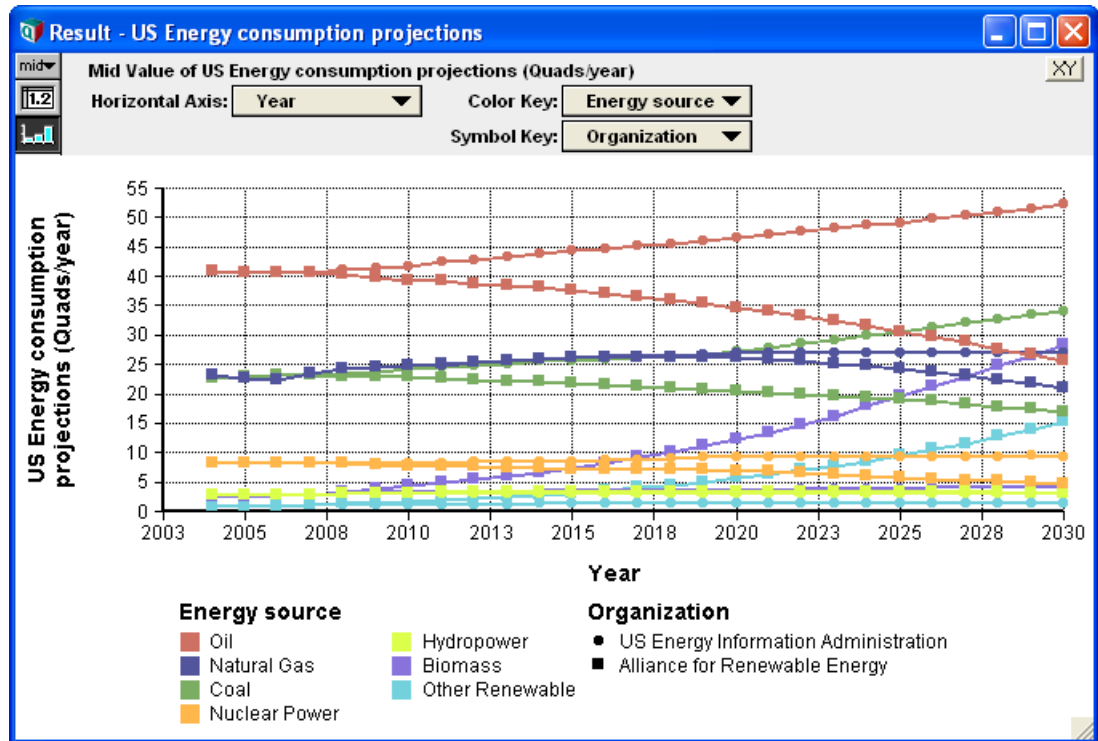
Graphing roles

A **graphing role** is an aspect of a graph or chart used to display a dimension (or index) of an array value; they include the horizontal axis, vertical axis, and key. A simple key uses colors, but you can expand it to include a symbol shape and size for each data point. When the array has too many dimensions to assign them all graphing roles, you can assign the extra indexes as slicer dimensions, from which you can select any value to display. For each available role, a graph shows a menu from which you can select the index you want to assign to that role. The flexibility of being able to directly assign graphing dimensions (such as indexes) to roles on the graph helps you find the best way to communicate multidimensional results. Graphing roles can display a continuous numeric scale or a discrete numerical or categorical scale — except for symbol size which must be numerical.

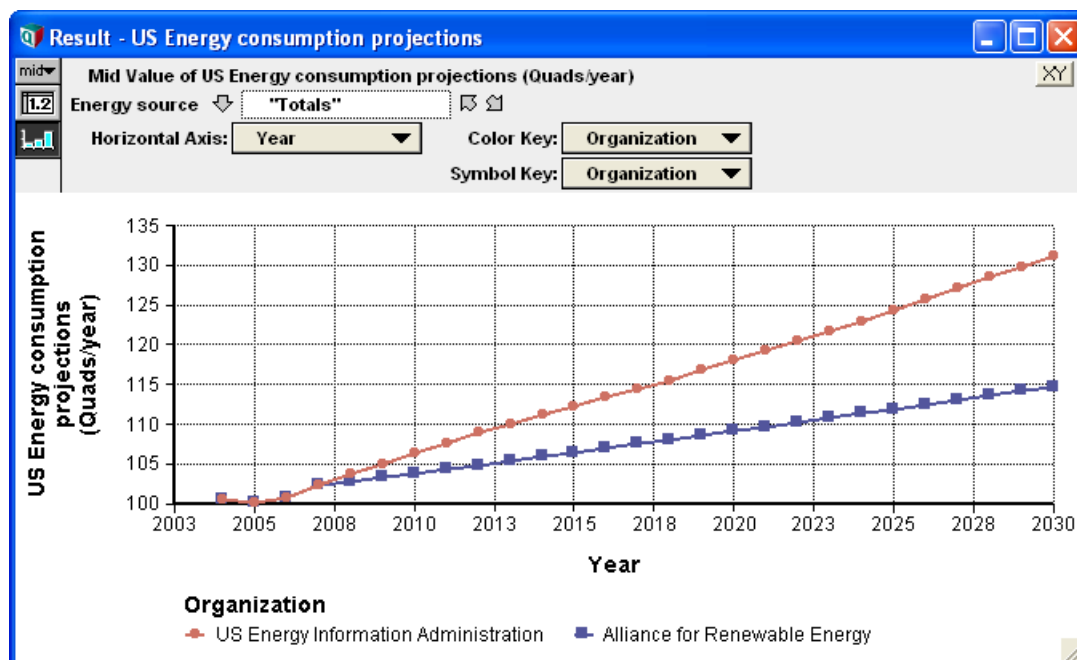
This example shows projections of US energy consumption made by two organizations, the US Energy Information Administration (actual) and the Alliance for Renewable Energy (fictional). The horizontal axis is set to **Energy source**, the key (color) is set to **Organization**, leaving the **Year** as a slicer, from which we have selected 2025:



Here we have changed graphing roles, assigning **Year** to the horizontal axis, **Energy source** to the color key, and **Organization** to the symbol key, leaving no need for a slicer:



In this version, the color key and symbol key both show the **Organization** index. The index **Energy source** is not assigned a visible graphing role, so shows up as a slicer. It is set to **Totals**, to show total over energy sources for each organization:



These are the graphing roles available:

- Vertical axis** Simply the vertical direction, labeled along the left edge of the graph. By default, it shows the actual values in the array — other roles usually show values of an index. All graphs use this role, but the **Vertical Axis** menu only appears if you have set **Swap horizontal and vertical** in the **Graph setup** dialog (page 94) or for XY graphs (page 105).
- Horizontal axis** The horizontal direction, labelled with numbers or text along the lower edge of the graph. It always appears, except when you set **Swap horizontal and vertical** for a 1D array. In the table view, it becomes the column headers.
- Key** Defines the color of lines or symbols. By default, it appears for the second index, if the array has more than one dimension. The key appears below the graph — unless reset in the **Style** tab of **Graph setup** (page 94). In the table view, it becomes the row headers.
- Color key and symbol key** If you check **Use separate color/symbol keys** in **Graph setup** (page 94) (available for the two line styles that show symbols), it expands the key into two graphing roles, **color key** and **symbol key**. Each has its own role menu, letting you assign a second and third index.
- Symbol size key** If you further check **Allow variable symbol size**, it adds symbol size as a fourth graphic role. You can specify the range of sizes from smallest to largest in typographic points, corresponding to smallest and largest values of the corresponding index. (It only works for a numerical index.) Symbol key and symbol size key do not appear in the Table view.
- Slicers** If the array has a dimension not assigned to a visible graphing role, it appears as a **slicer** — a menu above the graph. The value you select from a slicer menu applies to the entire graph, so the graph does not show values for other elements of the slicer. You can also select “Totals” from a slicer to show the total over all numerical values over that index. Slicers appear the same in the table as in the Graph view. If you have more than

one slicer, you can reorder them from top to bottom, in edit mode, simply by dragging a slicer up or down.

Graph setup dialog box

The **Graph setup** dialog box lets you apply a wide variety of graphing styles and options to the selected graph, or as the new defaults for all graphs in this model. It also lets you use or define graph templates, to apply a standard collection of styles and options to a graph.

When you display the result of a variable, it shows it as a table or graph, according to how you last viewed it. The first time you view a result, it appears as a graph, unless you changed the default result view in the **Preferences** dialog.

When displaying a graph, Analytica uses the default graphing settings, unless you have selected other settings for it. You can modify these with the **Graph setup** dialog.

To open the Graph setup dialog

First display a graph. Then do one of these:

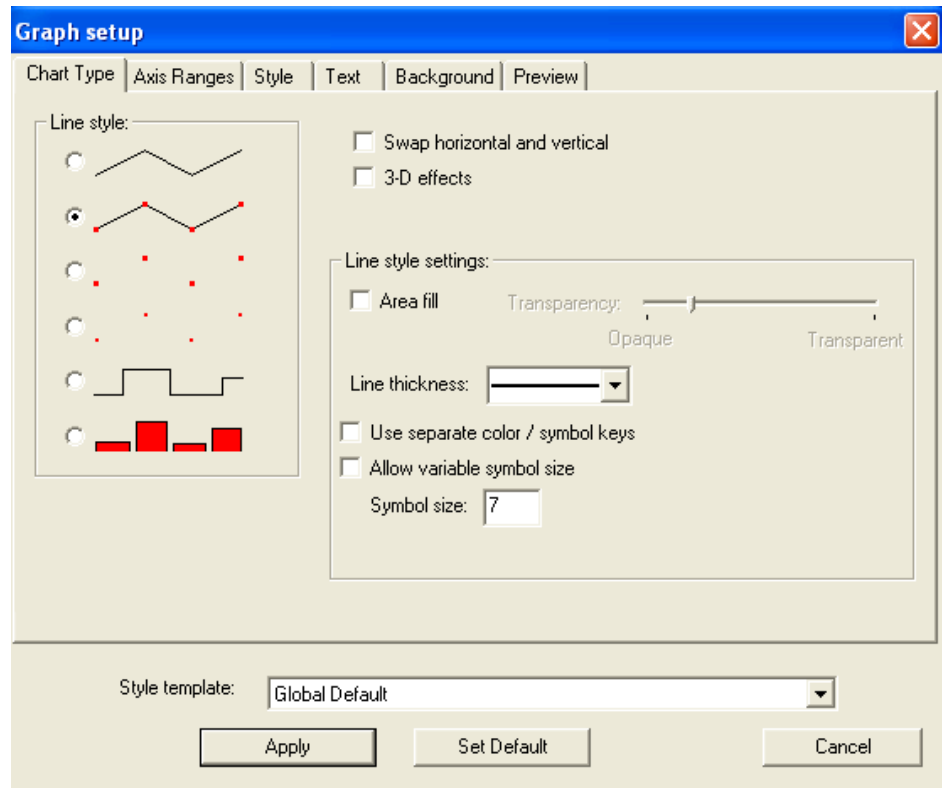
- Select **Graph Setup** from the **Result** menu.
- Select **Graph Setup** from the right mouse button menu.
- Double-click the graph in the **Result** window.

The graph setup dialog has six tabs. All tabs show the template panel and these three buttons:

- **Apply:** Apply any changes to settings to the current graph, and close the dialog.
- **Set Default:** Save any changed settings on the current tab as the default for all graphs, and close the dialog. It does not affect any settings that you have not changed since you opened the **Graph setup** dialog. Changing a default affects all graphs that use the default, but not graphs for which you override the default (in the past or future).
- **Cancel:** Close the dialog without changing or saving anything.

Chart Type tab

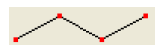
This tab shows options for modifying the style and arrangement of the graph:



Line style



Line segments join the data points.



Line segments, with a symbol at each data point.



A symbol at each data point with no lines.



A pixel at each data point, with no line.



A histogram or step function, with a vertical line and horizontal line from each data point to the next.



A bar centered on each x value, with height showing the y value. Forces the graph to be discrete.

Swap horizontal and vertical

Check this box to exchange the x and y axes, so that x axis is vertical and y axis is horizontal. If x values are discrete with long labels, swapping axes gives a more easily legible bar graph.

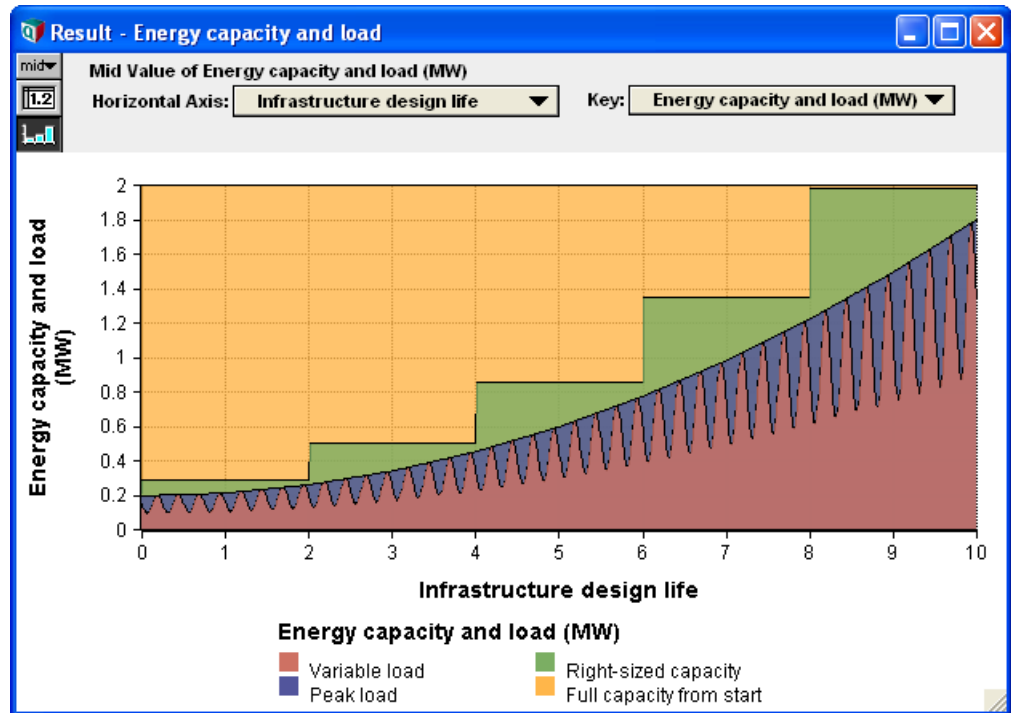
3-D effects

Check to use three-dimensional style to view graphs. For a bar graph line style, it offers the choice of **Box** or **Cylindrical** shapes for the bars.

Line style settings

Displays when you select a line style showing lines.

- **Area fill:** Check to fill in the area beneath each line with a solid color. If there are multiple lines, the graph will have a **key index**. It draws the fill areas from last to first element of the key index, which works well if the y values are sorted from smallest to largest over the key index. Otherwise, later values will obscure earlier ones. Here's an example:



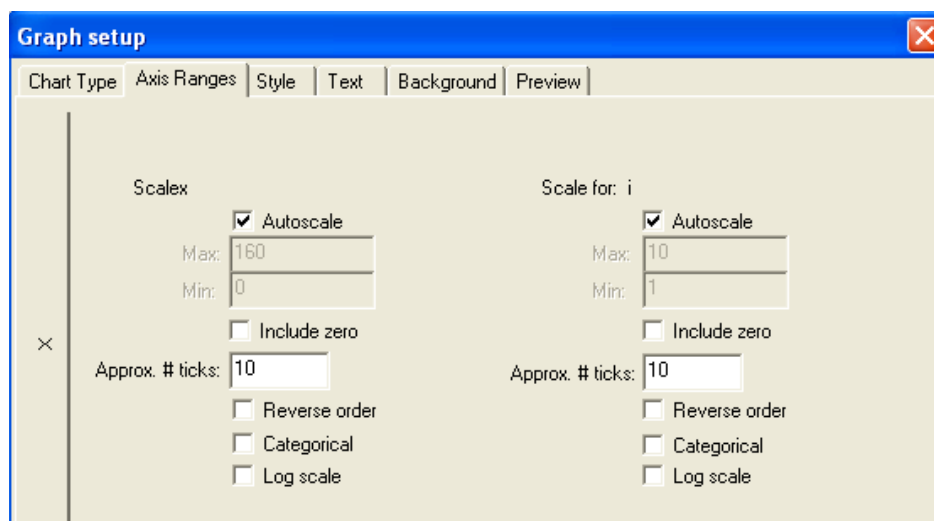
- **Transparency:** Drag the cursor to change transparency of fill colors between opaque and transparent. Transparency lets you see fill lines and areas that would otherwise be obscured behind others.
- **Line thickness:** Select the thickness of lines to display. (Only for styles that show lines.)
- **Use separate color/symbol keys:** Check to display two key index roles, one indicated by color and the other by symbol type or size.
- **Allow variable symbol size:** Check to have the size of symbols vary with their value.
- **Symbol size:** Enter a number to specify size of symbols in typographic points.
- **Min symbol size and Max symbol size:** If you check **Allow variable symbol size**, use these fields to specify the range of symbol sizes from smallest and largest.

Bar graph settings Displays when you select **Bar** graph line style:

- **Stacked bars:** Check to show bars stacked one on top of the other over the key index, instead of side by side. The values for each bar are cumulated over the key index.
- **Variable origin:** Check if you want to set the origin (starting point) for each bar other than zero (the default). The graph will then display a **Bar Origin** menu to let you select the bar origin.
- **Bar overlap:** With stacked bars, they overlap 100%. You can specify partial overlap between 0 and 100%.

Axis Ranges tab

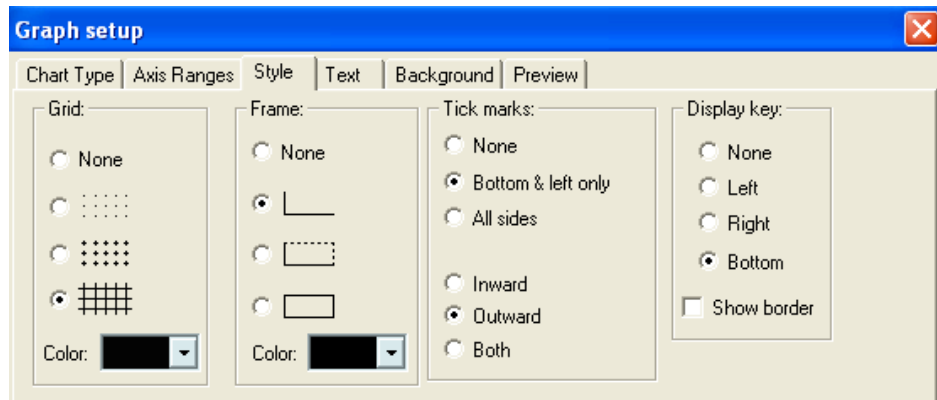
This tab lets you control the display for each axis, vertical and horizontal, including scaling, range, and tickmarks:



- Autoscale** Uncheck this box if you want to specify the range for the axis, instead of letting Analytica select the range automatically to include all values.
- Max and Min** The maximum and minimum values of the range to use when you have unchecked **Autoscale**.
- Include zero** Check if you want to include the origin (zero) in the range.
- Approx. # ticks** Specify the number of tick marks to display along the axis. Analytica might not match the number exactly, in the interests of clarity.
- Reverse order** Check this box if you want to show the values ordered from large to small instead of the default small to large.
- Categorical** Treat this axis as categorical. Usually, Analytica figures out the quantity is categorical without help. Occasionally, if the values are numerical, you might want to control it yourself. See “Probability density and mass graphs”.
- Log scale** Check if you want to display this on a log scale. This is useful for numbers that vary by several orders of magnitude. It uses a “double log” scale with zero if the values include negative and positive numbers.
- Set default** If you have changed settings for an axis that is an index of the variable being graphed, clicking this button applies these changes to that index for *all* graphs that use that index. For example, if the scale is the **Index Time**, you can use this to change the **Time** scale (e.g., start and end year) for every graph that displays a value over **Time**, unless you want to override that default in another graph.

Style tab

The **Style** tab lets you modify the display of the style and color of the grid, frame, and tick marks, and where to display the key.



Grid Select the radio button to control the display of the grid over the graphing area. You can also select the color. A light or medium gray is often a good choice.

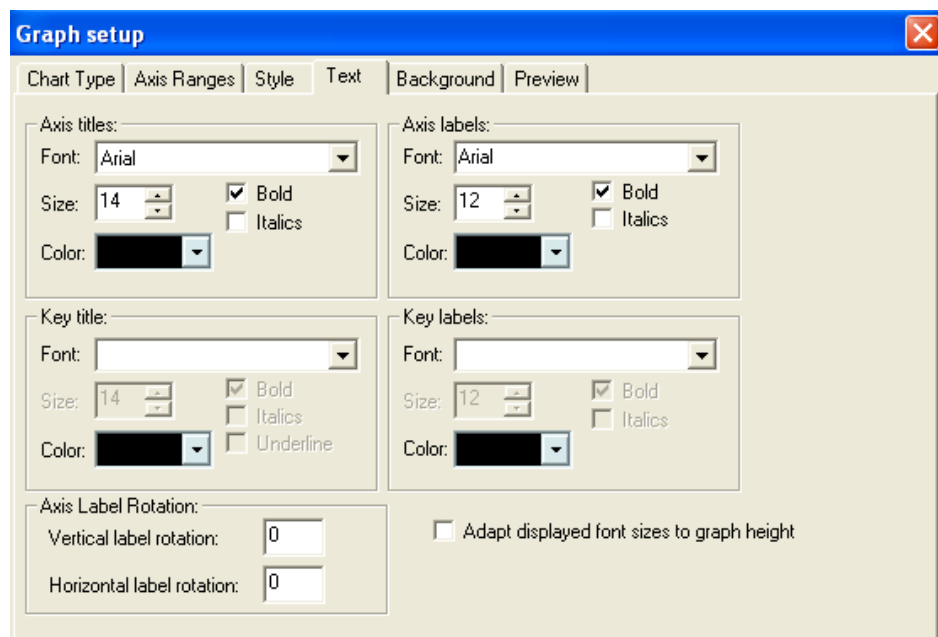
Frame Select the radio button to control the display of the lines framing the graphing area. You can also select the color for the frame. It is usually best to make the frame the same color as the grid, or a darker shade of the same color.

Tick marks: The top radio buttons control where to show tick marks. The lower ones control how they are displayed.

Display key Select the radio button to control where to display the key on the graph. Select the **Show border** check box to display an outline rectangle around the key.

Text tab

The **Text** tab lets you change the font, size, style, and color on the graph for the text of axis titles, axis labels (i.e., numbers or text identifying points along each axis), key titles, and key labels (i.e., identifying values in the key).

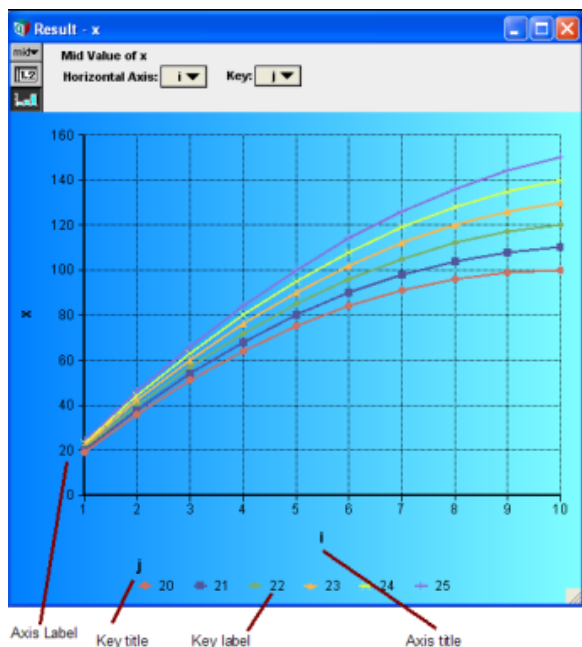


Font Select the font family. Graphic designers recommend using the same font for all text, which you can easily do by leaving all except axis titles as “(Same as axis titles)”.

Size The size in typographic points. Set to 0 if you want that type of text to not display.

Color Select the color.

Bold, Italics, and Underline Check these boxes to add bold, italic, and underlined formats to the text.

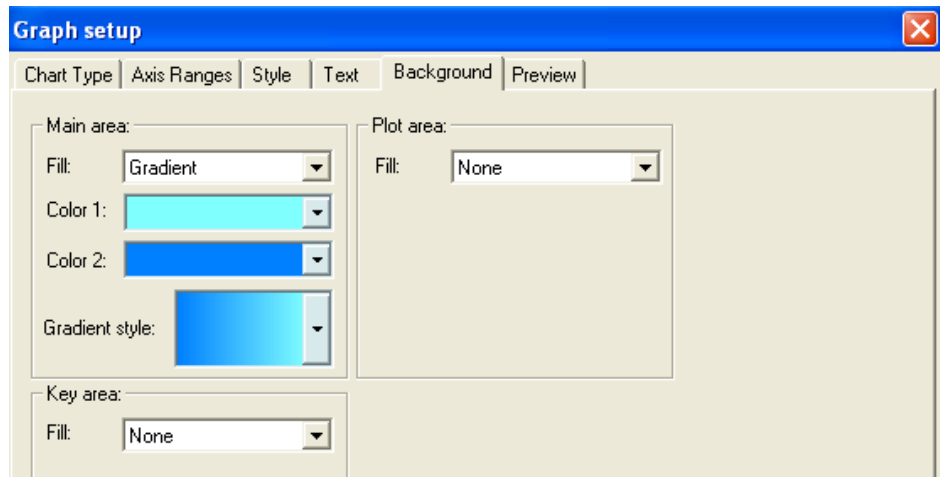


Axis label rotation Enter a number from -90 to 90 degrees to rotate the labels for each axis. For example, for a bar graph with many long labels along the horizontal axis, they won't all fit. By rotating them by 45 or 90 degrees, you can make them all fit without getting truncated.

Adapt displayed font sizes to graph height If you check this box, the font size automatically adjusts when you make the graph window larger or smaller. This can be useful when you give a demo and want to expand graphs so they are easily readable to people at the back of the room. The font sizes match those specified at the default graph height of 300 pixels.

Background tab

This tab lets you control the fill color, gradient, or pattern on the graph background. The main area covers the entire graph window (exclusive of the top area containing indexes). The plot area is the rectangle showing the graph values. If you leave or set the **Fill** to **None** for the **Plot area** or **Key area**, they will show the same fill settings (if any) as the **Main area**.

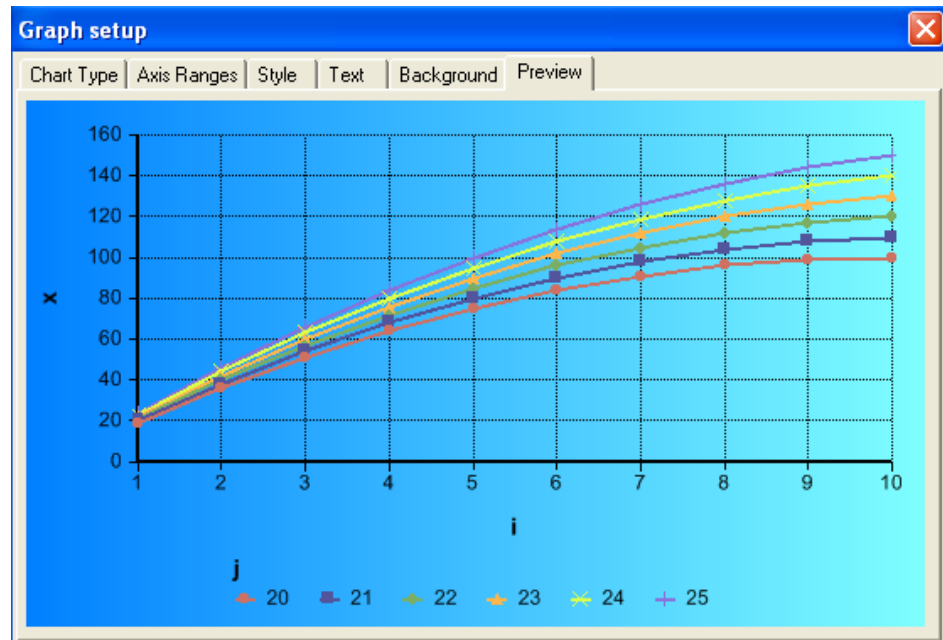
**Fill** Select from:

- **None:** No fill. Default to blank (white) background.
- **Solid:** Use a solid fill with the selected **Color 1**.
- **Gradient:** Use a gradient of color, going from **Color 1** to **Color 2**, in the direction you specify in **Gradient style**.
- **Hatch:** Use a hatched fill using the selected **Hatch Style** with **Color 1** and **Color 2**.

Graphic designers recommend avoiding hatched backgrounds, and using solid or gradient backgrounds with pale colors, if at all. The data should not be overwhelmed by the background.

Preview tab

This tab shows the graph using the current settings so that you can see their effects before you decide to **Apply** or **Cancel** them.



Categorical and Continuous Plots

Analytica 4.0 gives much more attention and consistency to the treatment of **categorical**, **continuous**, and **discrete** results.

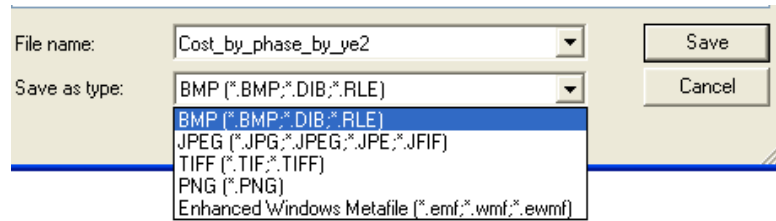
The discrete vs. continuous distinction is determined by the domain attribute, and determines whether probability plots are density and cumulative density plots (continuous) or probability mass and cumulative probability (discrete) plots.

The categorical vs. continuous distinction determines how a graphing axis is laid out. Continuous dimensions require numeric values. The determination of whether a graphing dimension is categorical or continuous is partially determined by the domain attribute. However, the values actually occurring in the dimension are determined by the chart type (bar or non-bar chart) and by the **Categorical** checkbox in the axis range setting.

Exporting graph image type

You can export a graph as an image file in most common formats, including BMP, JPEG, TFF, PNG, and Enhanced Windows Metafile (EMF):

1. Display the graph the way you want.
2. Select **Export** from the **File** menu, to open the **Save Graph Image as** file browser dialog.



3. If you want to change the defaults, edit the **File name** and select the **Save as type**, i.e., the file format.
4. Click **Save**.

Graph templates

Graph templates let you apply a collection of graph settings to several graphs, or even to all the graphs in a model. Analytica 4.0 includes several standard templates. You can also define your own templates to create standard graphing styles for a model, project, or an entire organization.

To use a graph style template

To apply an existing graph template to a graph:

1. Double click your graph to open the **Graph setup** dialog.
2. From the **Style template** menu at the bottom of the dialog, select the template you want.
3. To see what the templates look like, click the **Preview** tab. As you select each template from the **Style template** menu, it applies it to the selected graph. All template settings will be reflected in the settings in the other tabs.
4. If you want to modify any other settings beyond what the template specifies, you can do so now.
5. When you are happy with the results (check them in the **Preview** tab), click **Apply**, or if you don't like any of them, click **Cancel**.

To stop using a graph style template

If you have a graph that uses a template **T**, and you wish to unlink it from the template, change the **Style Template** menu back from **T** to **Global Default**. It asks "Do you want to retain these styles for this graph?" If you answer **yes**, it copies the template settings to be local for this variable, so it looks the same, but future changes to the template have no effect. If you answer **no**, it removes the template settings from this graph so it reverts to the global defaults.

To define a new graph style template

To create a new graph template so you can reuse a collection of graph settings for other variables:

1. Open the **Graph setup** dialog by double-clicking the graph with the settings you want to reuse, or if you want to save only new settings, open it for a new variable.
2. If you want to modify or add any settings, make those changes. You can also make a new template with changes to an existing template: In that case, select the existing template and click **Apply template**.
3. Click the **Preview** tab to see what all settings will look like.
4. From the **Style Template** menu, select **New Template**.
5. Type in a name for the template.
6. Click the **Set Template** button.

You have now created a new template, which will be saved with the model. You can apply it to any graph in the model.

To modify a graph style template

To modify an existing graph style template **T**:

1. Open the **Graph setup** dialog by double-clicking a graph for variable **v**.
2. If variable **v** does not already use template **T**, select **T** from the **Style template** menu.
3. Modify any **Graph settings** you want for **T**.
4. Check the effect in the **Preview** tab.
5. When satisfied, click **Set template**.

Any changes you make to a template will affect all variables that use it, except for any local settings that override them for a particular variable.

Combining local, template, and model default settings

You can apply graph settings, and most uncertainty settings, at three levels:

- | | |
|-----------------------|---|
| Local | Clicking Apply in the Graph setup or Uncertainty Setup dialog applies any settings you have modified in the dialog to the current variable. These settings override any global or template settings. |
| Graph template | By selecting a style template in the Graph setup dialog and clicking Apply , you apply the template settings to the current variable. The template overrides any global settings, but not local settings. |
| Model defaults | Clicking Set Default in the Graph setup or Uncertainty Setup dialog changes the global defaults for the model for any settings you have modified in the dialog. |

Tip If you change a global setting by clicking **Set default**, that change will change that setting for all graphs that do not override it by a template or a local setting.

The **Uncertainty sample** tab of the **Uncertainty Setup** dialog is an exception: Settings on that tab — e.g., **Sample size** — are always defaults that affect the entire model. They cannot be local and are not saved in a graph template.

Saving defaults as a template model

Analytica comes with a wide variety of standard defaults for graph settings, uncertainty options, preferences, diagram style, and more. If you want to save nonstandard default settings for these, perhaps also including graph templates and libraries so that you can use them for new models, the easiest method is to create a new template model:

1. Find or build a model that has all the default settings you want, including any graph settings, uncertainty settings, preferences, diagram style, graph templates, and user-defined attributes. It could also contain any libraries that you will want in all the new models.
2. Select **Save as** from the **File** menu to save the model under a new name, e.g., **Template.ANA**.
3. Delete all the contents of the model that you won't need for new models.
4. Select **Exit** from the **File** menu and save the model.

Whenever you want to start a new model using these defaults, double-click **Template.ANA**, and save the model under a new name. To protect your template model from you accidentally changing it by saving a new model over it with the same name:

1. In the Windows Finder, open the folder containing **Template.ANA**.
2. Right-click **Template.ANA**, and select **Properties**.
3. Check the **Read-only** attribute, and click **OK**.

Graph templates and setting associations

Chart type and uncertainty views	Graph settings from the Chart type tab are associated with particular uncertainty views . For example, if you set Line style to symbols only (instead of the default pixel per data point) for a Sample plot, that line style will apply to any sample plot, but not to other uncertainty views Mid , Mean , Statistics , PDF , or CDF . Thus, you may set a different Style setting for each uncertainty view, except Mid , Mean , and Probability Bands , which share the same style.
Settings for discrete vs. continuous	It maintains separate line-style settings for continuous and discrete (categorical) plots. So, pivoting a continuous dimension to the x-axis to replace what was a discrete dimension may change the plot from a bar graph to line graph, and use the corresponding settings.
Axes and indexes	If the horizontal axis is an index (as it usually is), any settings on the Axes Ranges tab apply to that index only. For example, suppose variable Earthquake_damage is indexed on the horizontal axis by Richter_scale . You set Richter_scale to Log scale , and save into a template T . If you use template T for another variable y also indexed by Richter_scale , it also displays Richter_scale on a log scale. But, if y is not indexed by Richter_scale , the Axis setting will have no effect.

Uncertainty options and graph templates

A graph template also saves non-default settings made in the **Uncertainty setup** dialog tabs: **Statistics**, **Probability bands**, **Probability density**, or **Cumulative probability**. These settings apply to the corresponding uncertainty view of any variable using the template. Changes to the **Uncertainty sample** tab, however — e.g., to **Sample size** — set global defaults, which affect the entire model. They are not associated with particular variable, or saved in a graph template.

Changing the global default

Global default specifies the default settings used by every graph unless overridden in the **Graph setup** dialog for that graph or by a template that it uses. If the **Style Template** menu says **Global default**, it means that the graph uses the global defaults with no template.

To modify the global defaults:

1. Select a new variable with no graph settings, or a graph whose settings you want to make the global default.
2. Double click the graph to open **Graph setup** dialog.
3. If you want, make further changes to the settings, and review them in the **Preview** tab.
4. From the **Style template** menu, select **Global Default**, if it isn't already selected.
5. Click **Set default** button.

Note: Changes to global defaults change all existing and new graphs that use those global defaults; that is, all that are not overridden by any graph settings specifically set for that graph or by a template that it uses.

To rename a graph style template

1. Open the **Graph setup** dialog, by double-clicking a graph.
2. In the **Style template** menu, select the graph template you want to rename.
3. Click the **Style template** menu to select the old name.
4. Type in the new name.
5. Click **Set template**.

*Note: The template "name" is actually its **Title** attribute, not its identifier. So, renaming a template does not affect any variables that use it.*

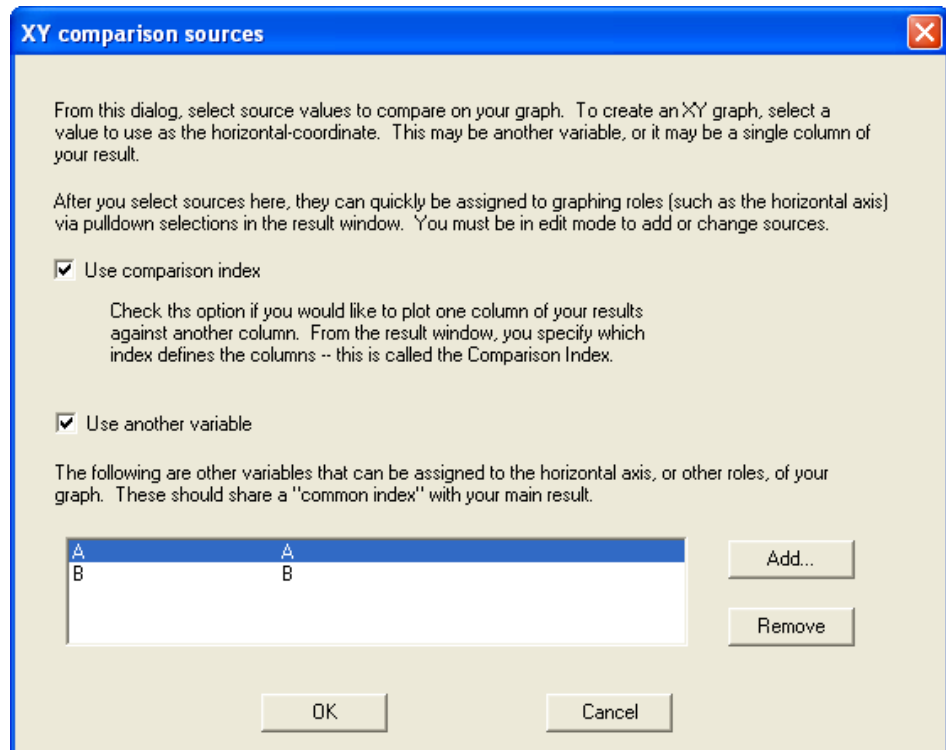
XY comparison

When you display a standard (non XY) graph of a variable, **V**, it plots the values of **V** up the vertical (Y) axis against an index of **V** along the horizontal (X) axis. If **V** has more than one dimension, you can choose which index to plot horizontally from the **Horizontal Axis** menu. With **XY Comparison**, in contrast, you can plot **V** against another variable, **U**, along the horizontal (X) axis, over a **Common index** of **V** and **U**. You can also

plot one slice of **V** against another slice over a **Comparison Index**. (See “Scatter plots” to use XY comparison for scatter plots.)

XY comparison sources dialog

This dialog lets you set options for XY comparison and extends or adds menus to the XY graph described below.



To open the dialog Click the **XY** button in top right corner of **Result** window (graph or table). You must be in edit or arrow mode, so it is not available in Analytica editions or models confined to browse mode.

Use comparison index Check this box if you want to compare one slice of the variable against another slice, slices selected from the Comparison index. The graph will show the **Comparison Index** menu from which you can select the index you want. The **Vertical Axis** and **Horizontal Axis** menus will then offer slices from the comparison index so that you can choose which two slices to plot against each other.

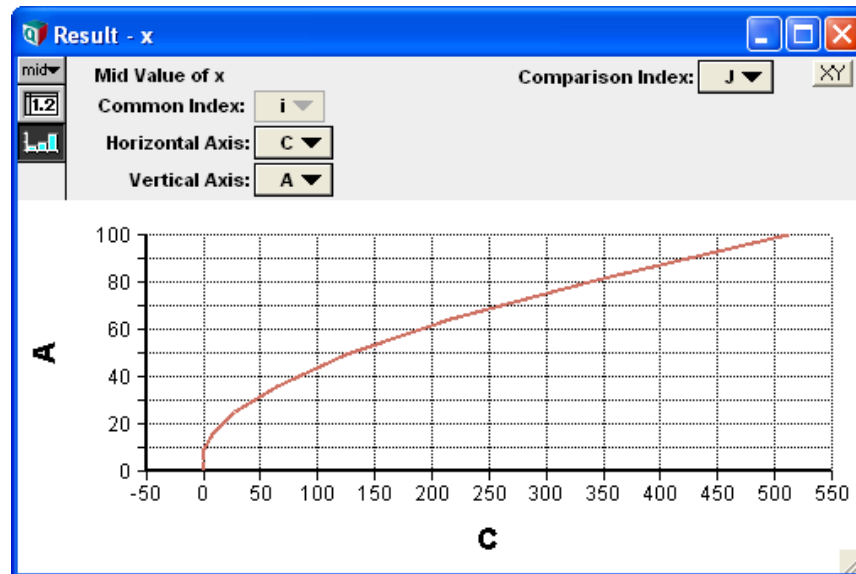
Use another variable Check this box if you would like to compare the base variable by plotting it against one or more other variables (or simple expressions). When you check it, the following items appear:

Add Click this button to open the **Object Finder** dialog to select a variable against which to plot the base variable. You can also use the **Object Finder** to select a function or operation from one of the relevant libraries. You can add up to five items.

Remove Select a item from the list of other variables, and click this button to remove it from the list of variables for comparison.

Menus added to XY Comparison graph

An XY comparison graph adds a **Common index** and, sometimes, a **Comparison index** to the usual graphing roles menus on a graph or table.



Comparison index This menu lists the indexes of the base variable. The **Horizontal Axis** and **Vertical Axis** menus each let you choose a slice from the selected comparison index to plot against each other. It appears on the graph when **Use comparison index** is checked in the **XY comparison sources** dialog.

Common index It defines the correspondence among the variables or slices to be plotted against each other: Each value of the Common index identifies a data point on the graph, with vertical (X) and horizontal (Y) values from the variables or slices you have selected for those graphing roles. For a scatter plot, the common index should be **Iteration (Run)**. It appears on the graph when one or both checkboxes on the **XY comparison sources** dialog are selected.

If **Use another variable** is checked in the **XY comparison sources** dialog, the common index will be an index in common to the base variable and other variable(s). If the variables have more than one index in common, **Common index** will be a menu from which you can choose the index you want.

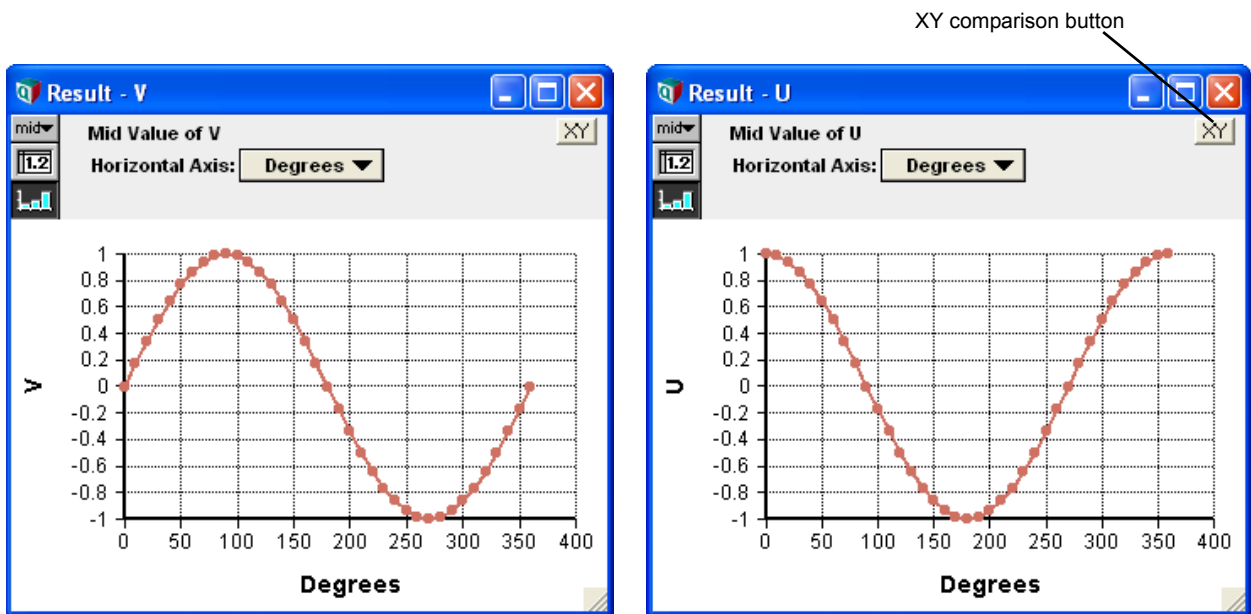
If **Use comparison Index** is checked in the **XY comparison sources** dialog, Common index will show the index(es) of the base variable not selected for the Comparison index. Common index will be a menu if the variable has more than two indexes — leaving more than one for the Common index.

Example: Plot one variable against another

For example, suppose you have an index and two variables:

```
Index Degrees := Sequence(0, 360, 5)
Variable V := Sin(Degrees)
Variable U := Cos(Degrees)
```

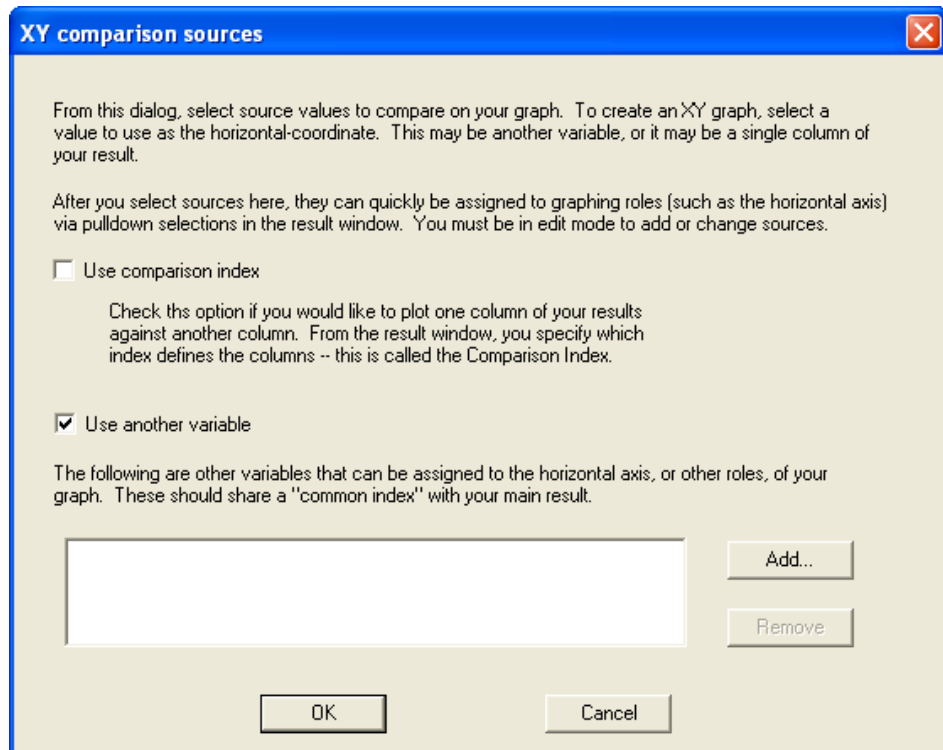
For a standard graph of **V** against its index, **Degrees**, select **V** from the diagram and click the **Result** button (*Control+r*). Repeat with **U** to display the graph for **U** against **Degrees**:



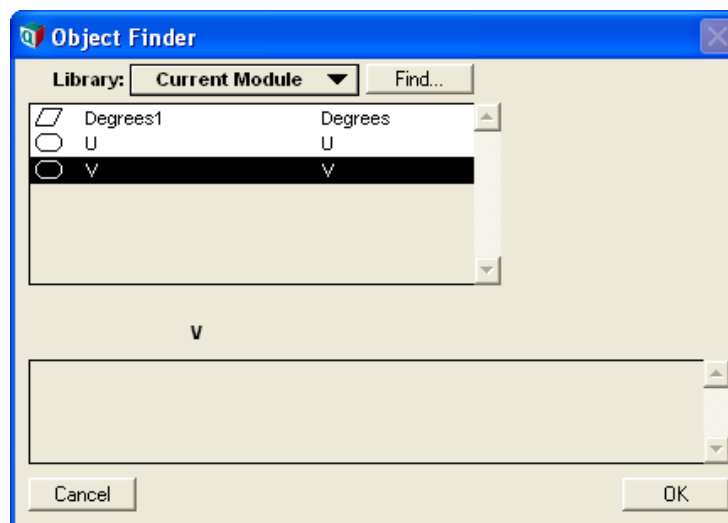
For these graphs, we selected the *symbol plus line* line style from **Graph setup** to show the data points for each value of **Degrees** (page 95).

With **XY Comparison**, you can graph **U** against **V**, instead of against its index **Degrees**:

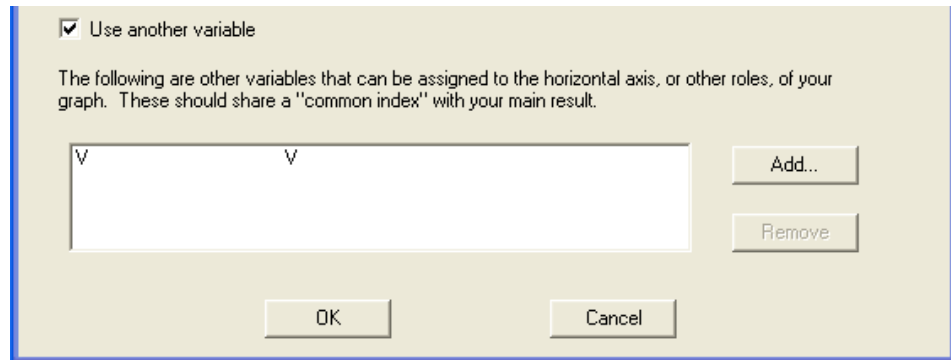
1. Change to *edit mode*. In the **Graph** window for **U**, click the **XY** button in the top right corner (above) to open the **XY Comparison sources** dialog box.



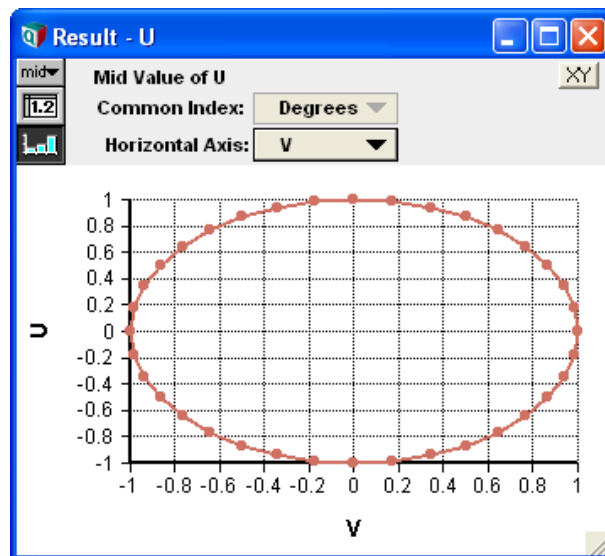
2. Select checkbox **Use another variable**.
3. Click **Add** button to open the **Object Finder** dialog.



4. Select the variable **V**, and click **OK**. You can now see **V** listed in the **XY comparison sources** dialog.



5. Click **OK**.



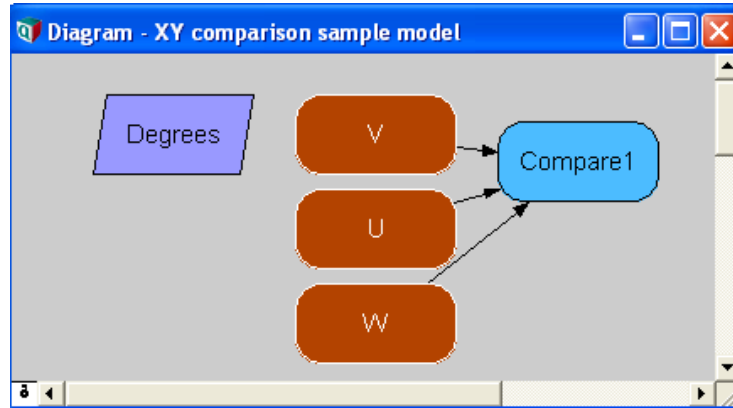
The graph of **U** now plots the values of **U** on the vertical (Y) axis against corresponding values of **V** on the horizontal (X) axis. By "corresponding" we mean for each value of **Degrees**, in the **Common index**. If **U** and **V** had more than one index in common, it would show a menu from which you could select the index you want.

Example: Compare variables using comparison index

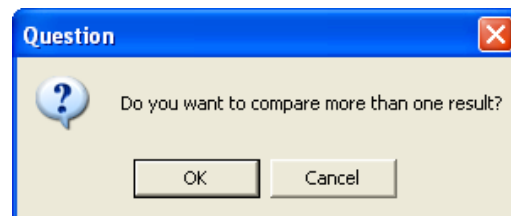
You can also use **XY Comparison** to compare one slice of a variable against another slice of the same variable. This is especially useful when you combine several variables as a list. Let's add a third variable to **U** and **V** defined above:

```
Variable W := Sin(2*Degrees)
```

The parameter **2*Degrees** creates a sine curve with twice the frequency. Here is an easy way to create a list to compare several variables:

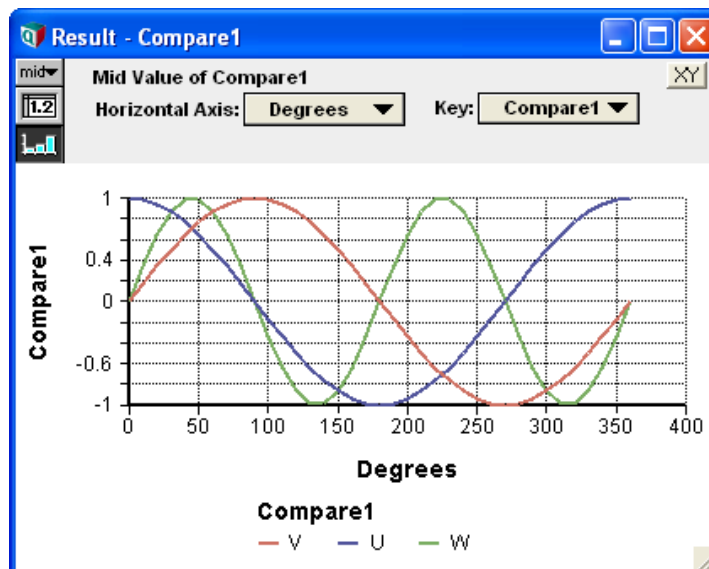


1. Select the three nodes for the variables to compare, **U**, **V**, and **W**, and click **Result** (*Control+r*).
2. When it prompts,

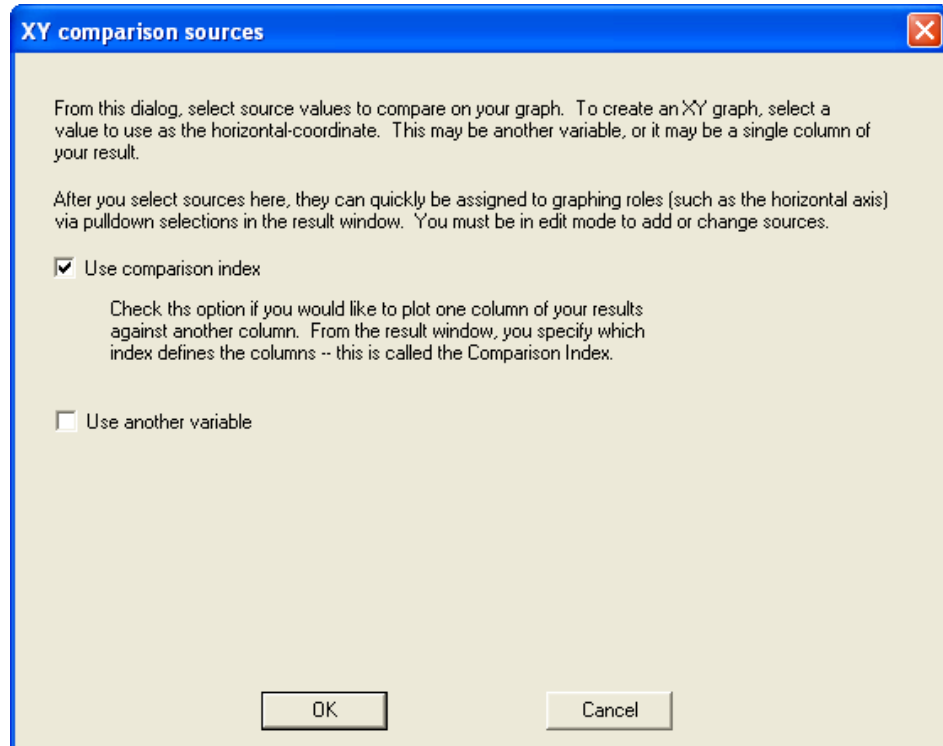


click **OK**.

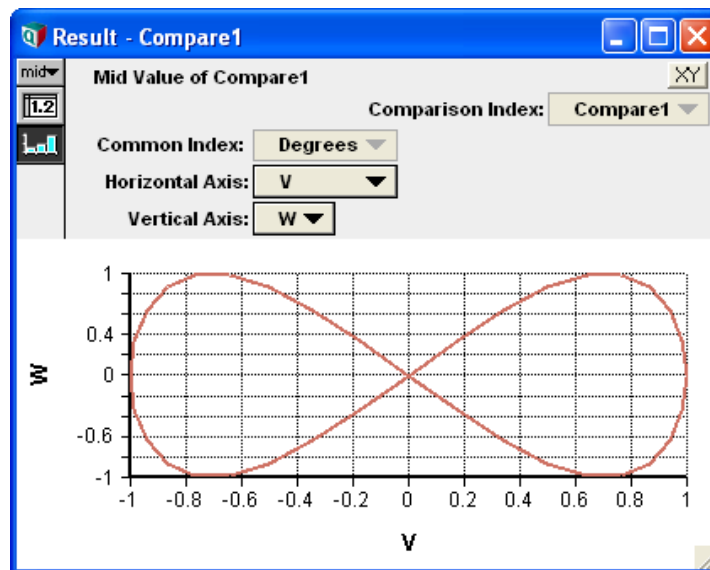
It creates a new variable **Compare1**, and shows the standard (not XY) graph comparing **U**, **V**, and **W** against Index **Degrees**:



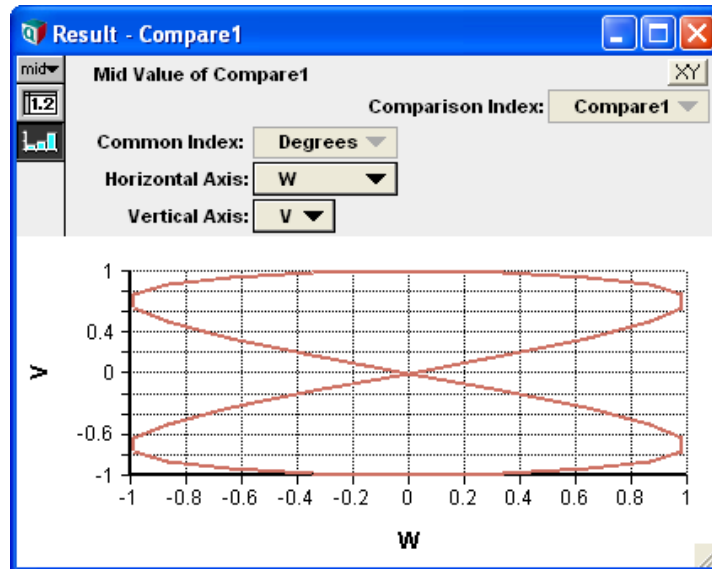
3. Make sure you are in *edit mode*. In the graph window for **Compare1**, click the **XY** button in the top right corner to open the **XY comparison sources** dialog box.



4. Select checkbox **Use comparison index** and click **OK**.



This sideways figure of 8 results because **W** is a sine wave with twice the frequency of **V**. You can select other pairs of variables to compare, from **U**, **V**, and **W**, from the **Vertical** and **Horizontal Axis** menus — for example, changing to **W** against **V** puts the figure of 8 the right way up:



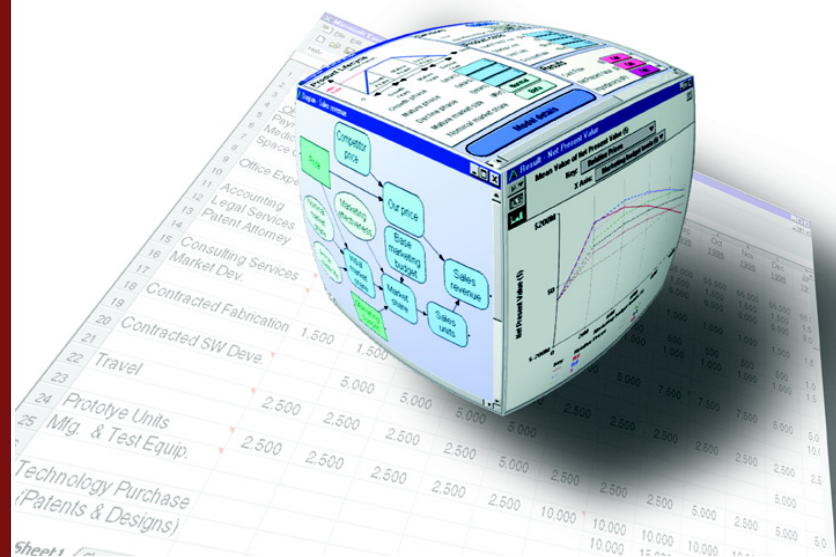
You can also select **Degrees** from the **Horizontal Axis** menu to revert to a standard (non XY) graph of the selected variable against **Degrees**.

Chapter 8

Creating and Editing Definitions


This chapter shows you how to:



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

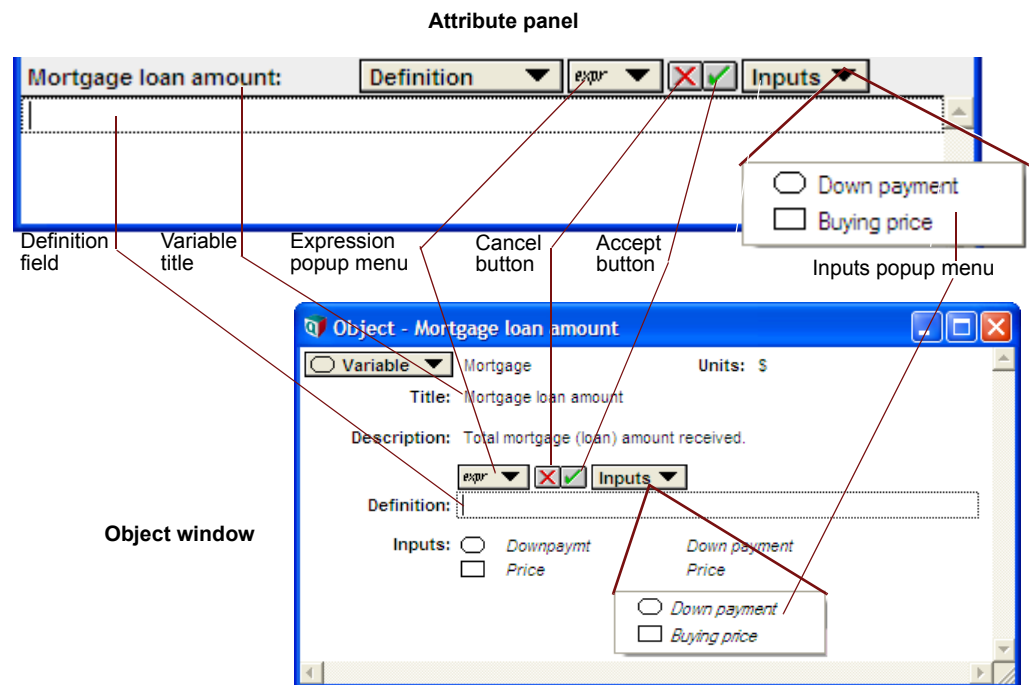


This chapter introduces the tools for creating and editing mathematical models by giving each variable a formula that defines how to compute its value in its **definition**. The definition of a variable can be a simple number, text, a probability distribution, or a more complicated expression. It can 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 of interest and do any of the following:

- Click  in the toolbar, or press *Control+e*.
- Select **Edit Definition** from the **Definition** menu.
- Double-click the variable to open its **Object** window. Then click in the definition field.
- Click 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 you have drawn arrows into this variable from other variables (**Down_payment** and **Buying_price** in this example), they appear in the **Inputs** menu. Select an input to paste its identifier into the definition. (The menu doesn't appear if the variable has no inputs.)

Tip If you are editing in the attribute panel, a handy way to insert the identifier of a node into the definition is to click the node while pressing the *Alt* key. It only works for nodes in the same diagram.

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** (*Control+c*), **Cut** (*Control+x*), and **Paste** (*Control+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”).

Special editing key combinations

These special mouse and key combinations are useful when editing a definitions:

Key or key combination	Action
<i>double-click</i>	Selects the entire identifier containing the cursor.
<i>option-click a node</i>	Inserts identifier of the node at the cursor position.
<i>left-arrow ←, right-arrow →</i>	Moves cursor one character left or right.
<i>Up-arrow, down-arrow</i>	Moves one line up or down.
<i>Control+left-arrow, Control+right-arrow</i>	Moves to the beginning or end of the next word or identifier.
<i>Alt+Control+left-arrow, Alt+Control+right-arrow</i>	Moves the cursor from the adjacent parenthesis to the next matching parenthesis, left or right.

If you also press *Shift* with any arrow movements, it selects the text between old and new cursor positions for copy/paste operations, etc.

Parenthesis matching

Analytica expressions sometimes contain several levels of nested parentheses. To help keep parentheses clear, when you place the cursor just to the right of a parenthesis, it makes it and its matching parenthesis bold. This works for left or right parentheses, square brackets, or curly brackets (used for comments). It helps you see whether you have the right number and types of parentheses in complex expressions, without resorting to counting.

The *Alt+Control+arrow* keys also help: For example, pressing *Alt+Control+right-arrow* when the cursor is at **A** moves the cursor to **B**. Then pressing *Alt+Control+left-arrow* moves it back again:

$$c * (- (\ln(\text{Uniform}(1f, 1)))) ^{(1/k)}$$

A
B

Comments in definitions

It is wise to document your models generously. Usually, it's best to document what a variable or function represents in its **Description** attribute, and also explain its algorithm if it's not obvious. For complex, multiline definitions, it's also useful to insert comments within the definition. Comments can also be used to disable portions of expressions while debugging.

Enclose comments in curly brackets:

```
Variable X := -b*Sqrt(B^2 - 4*A*C)/A { Positive quadratic root }
```

You may insert a comment at any point in an expression where whitespace is allowed. Analytica ignores anything inside a comment when parsing or evaluating an expression. If you start a comment with "{", then your comment cannot contain the "}" character within the comment.


It does not preserve comments in the cells of an edit table — so it's not worth entering comments there.

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 labeled by their identifiers (instead of their titles), select **Show By Identifier** from the **Object** menu (*Control+y*). (Note that entering *Control+y* a second time switches the diagram back to displaying the nodes by their titles.)
- Select **Paste Identifier** from the **Definition** menu and use the **Find** button or identifier menu items (see "Object Finder dialog").
- If the definition is being edited from the **Attribute** panel, you can insert the identifier of a variable in the same module window by holding down the *Alt* key and clicking 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 in different modules.

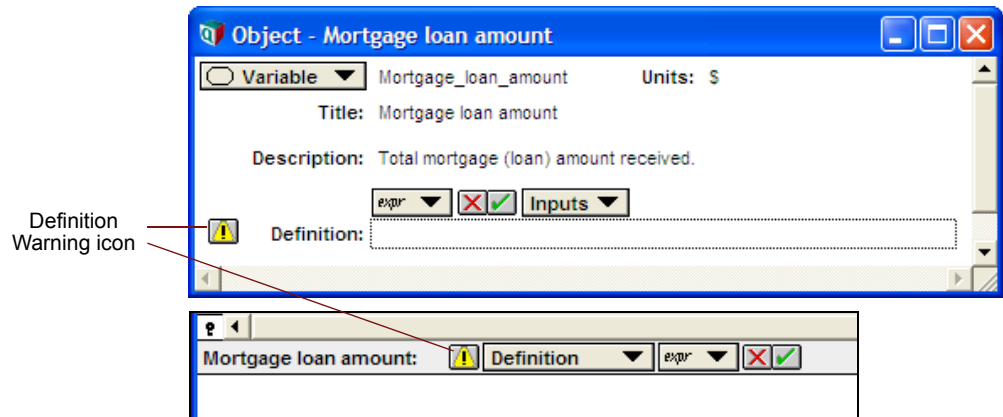
Functions You can paste functions at the insertion point by doing either of the following:

- Select **Paste Identifier** from the **Definition** menu to open the **Object Finder** (see "Object Finder dialog").
- Select the function from its library in the **Definition** menu (see "Using a function or variable from the Definition menu" on page 123).

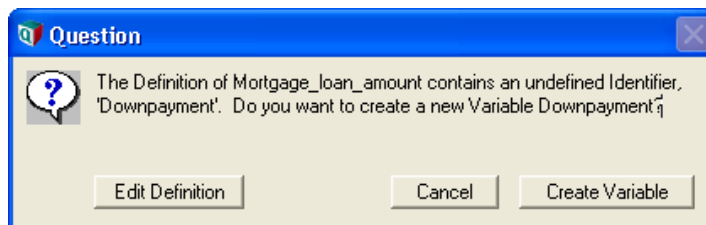
Syntax check After entering or editing a definition, press *Alt-Enter* or click the accept button  to perform a syntax check of the revised definition and accept the changes.

Click the cancel button  to cancel your changes.

The definition warning icon  appears next to the definition if it is not syntactically correct. Click the icon to see a message about what may be wrong.



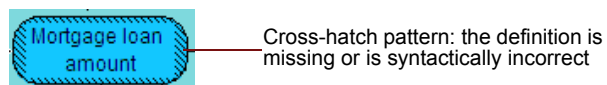
A definition's syntax check may reveal syntax errors (see "Syntax error"). For example, if a definition contains text that is not an identifier, the following dialog box appears.



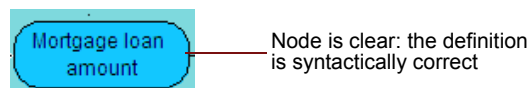
Automatically updating the diagram

After you give a variable a valid definition, the influence diagram containing that variable might change.

Cross-hatching disappears Normally, any node whose definition is missing or invalid displays with a cross-hatch pattern:



After you enter a valid definition, the node becomes clear.

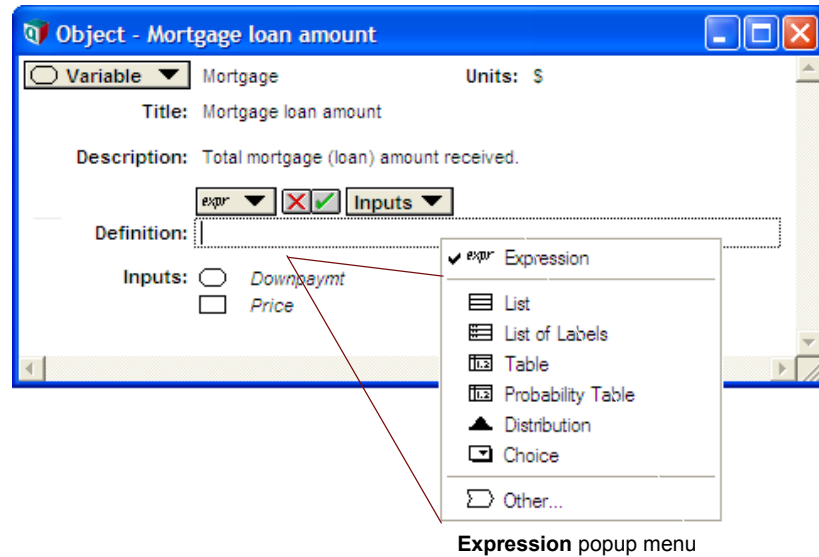


You can remove cross-hatching even from invalid variables by unchecking **Show Undefined** in the **Preferences** dialog from the **Edit** menu.

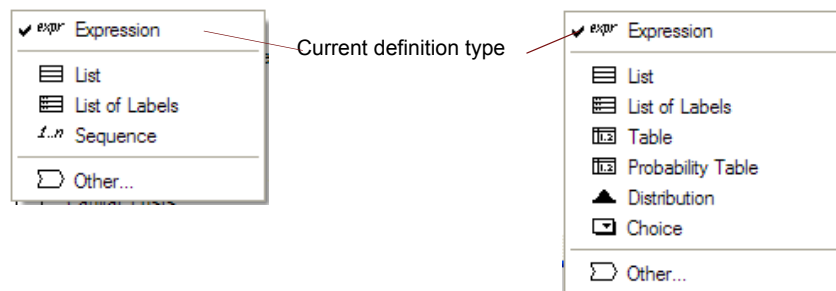
Arrow updating After you enter or edit a definition, it ensures that the arrows going into the node to properly reflect its inputs: It adds an arrow from any extra variable you mentioned. It removes an arrow from any variable you didn't use in the definition.

The Expression popup menu

Click **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.



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.



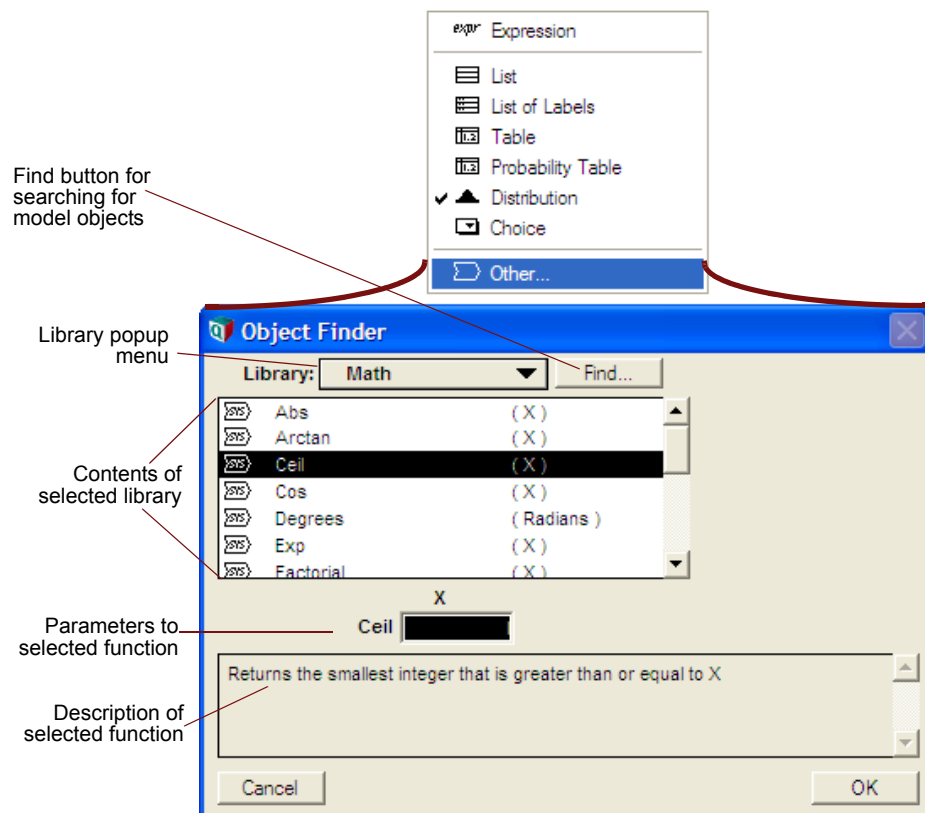
- Expression* Shows the definition as a mathematical expression, even if it was defined using the other expression types in this popup menu. See Chapter 10, "Using Expressions".
- List* Creates an ordered set of expressions or numbers. See "Creating an index".
- List of Labels* Creates an ordered set of text labels. See "Creating an index".
- Sequence* Creates a list of numerical values. See "Sequence (start, end, stepSize)".
- Table* Creates an array of numbers or expressions. See "Arrays and Indexes".
- Probability Table* Creates an array defining probabilities (numbers or expressions) across the domain of a discrete (chance) variable. See "Probable(): Probability Tables".

<i>Distribution</i>	Creates an uncertain definition by selecting a function from the Distribution System library. See “Defining a variable as a distribution”.
<i>Choice</i>	Creates a popup menu for choosing one or all elements from a list. See “Creating a choice menu”.
<i>Other</i>	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 .

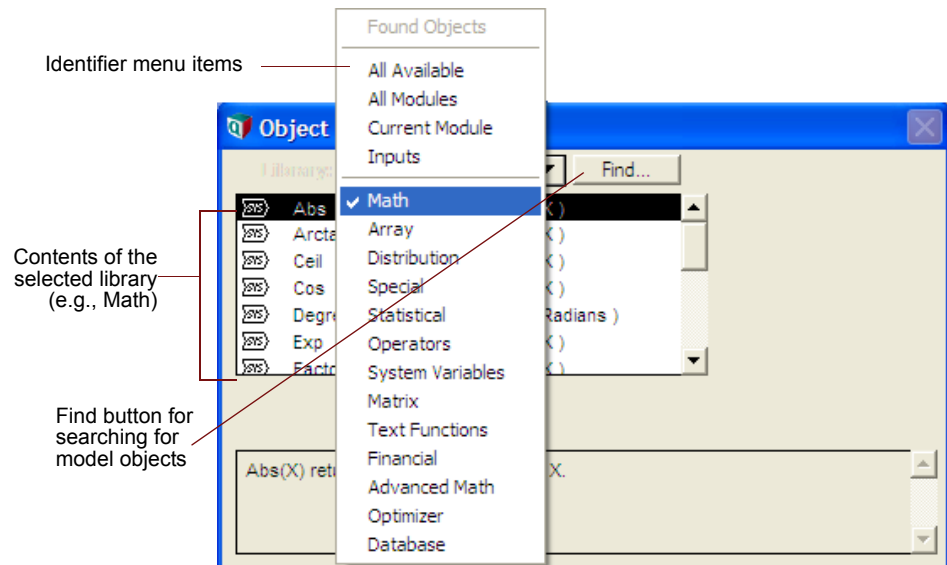
Object Finder dialog

The **Object Finder** dialog lets you browse built-in functions, your own library functions, and all the objects in your model to insert into a definition. You can open the **Object Finder** in two ways:

- To insert the desired function or identifier at the cursor position in the definition, select **Paste Identifier** from the **Definition** menu, or
- To replace the entire definition with the desired function, select **Other** from the **expr (Expression)** menu.



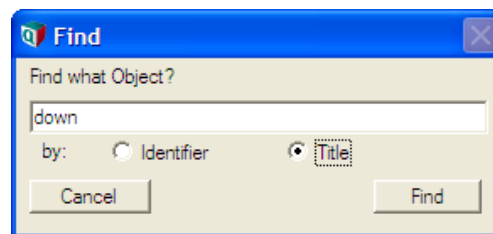
Select the desired set or library from the **Library** menu:



These are the top items in the library menu:

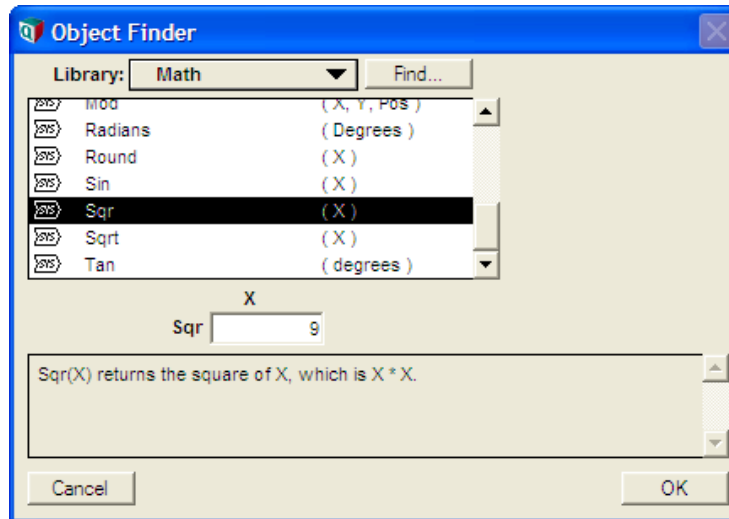
- Found objects** Objects found from **Find** button (see below)
- All Available** All objects and functions, from model and built-in
- All Modules** Objects from all module in the models
- Current Module** Objects in the current module
- Inputs** Inputs to the selected node

Use the **Find** button to search for an object by its identifier or title

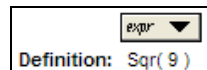


The **Found objects** library in the **Object Finder** dialog will then list all objects whose identifier or title matches in their first *n* characters (the *n* characters you type into the search box).

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



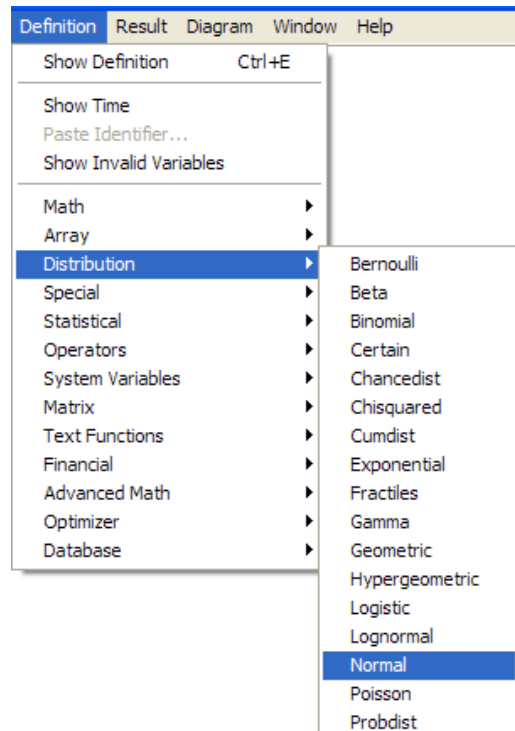
Click **OK** to place the function, identifier, or expression in the definition.



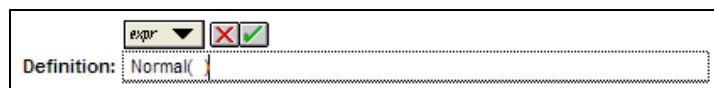
Using a function or variable from the Definition menu

The **Definition** menu lists built-in libraries of functions, system variables, and operators, as well as any libraries you have added. It shows these as a hierarchical menu that so you can rapidly find what you need and paste it into the definition you are editing. To find and paste a function or other object from a library:

1. Move the cursor to the place in the definition that you want to insert a function or other item.
2. From the **Definition** menu, select the library you want, and then the function or other item.



3. This pastes the item function into the definition, along with its formal parameters or operands, if any, each enclosed in << >>:



4. Now edit each parameter or operand to replace it with the appropriate identifier or expression. As usual, you may type it, select an item from the **expr** menu or the **Inputs** menu, or paste another object from the **Definitions** menu.

Automatic checking for valid values

You can create an automatic check on the validity of the value of a variable by setting its **check** attribute. For example, to check that the value of **Percent_damage** is between 0 and 100, set its **check** attribute:

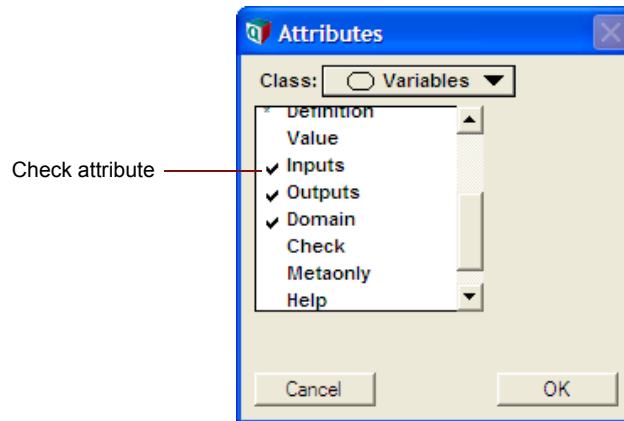
```
Check: Percent_damage>=0 AND Percent_damage<=100
```

If the **check** attribute evaluates to False, whenever the variable is evaluated, it shows a warning dialog and the opportunity to edit the definition.

You can always view and edit the **check** attribute in the **Attribute** panel, if you open it below a diagram. If you want to view or edit it in **Object** windows, you must first cause it to be shown:

Displaying the check attribute

1. Select **Attributes** from the **Object** menu to open the **Attributes** dialog. (For more see “Managing attributes”.)



2. Scroll down the attribute list and find **Check**.
3. Click **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.
4. Click **OK**.

Now the **check** attribute will appear in **Object** windows for all *variables*. You can also set it to appear for *functions* by repeating the steps above but selecting **Functions** from the **Class** menu in the **Attributes** dialog.

Defining the check

Either open the **Object** window for the variable, or open the **Attribute** panel below the diagram and select **Check** from the **Attribute** menu. Enter a Boolean (logical) expression in the **Check** field that will return true (1) if the value is acceptable, or false (0) if not. The expression should refer to the variable by its identifier or as **self**. For example, to check that the value for the **Lifetime** of a car is more than 0 and less than 12 years, 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, it makes a dependency from the variable being checked to the variable mentioned. It will usually show an arrow from that variable.

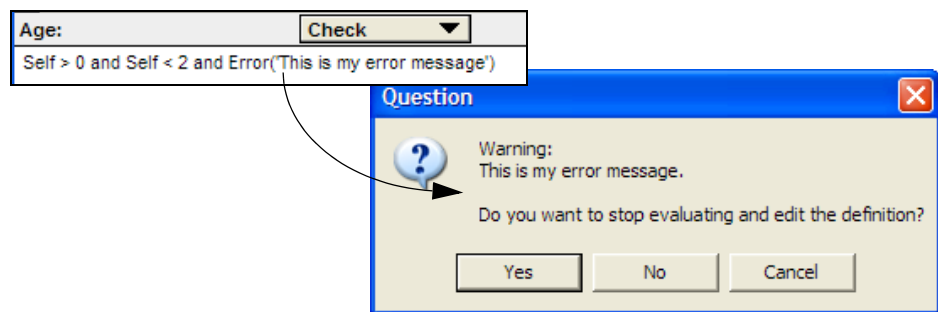
Triggering a check

If a variable **x** has an input node, it performs the check whenever you change its input value (or definition) (see “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.”). Otherwise, it performs the check each time it evaluates the checked variable, **x** — that is, when you first view a result for **x** or a variable on which **x** depends. If you view or compute a probabilistic value for **x**, it will warn if any sample value of **x** fails the check. More generally, if the value of the **Check** expression is an array, it will fail if any atom in the array is false (0). If you view first the mid value of **x** and then a probabilistic value, that causes two evaluations and so two checks.

If you change the definition of **x** or any variable on which it depends, *including* any variable mentioned in its **Check** expression, it will perform the check again next time you view **x** or a variable that depends on it.

If a check fails If a check fails — evaluates to False — the warning dialog offers the option of editing the variable's definition, cancelling, or continuing. If you continue, it will not perform the check again unless you change the definition of the variable or a variable it depends on.

Custom error messages The default warning when a check fails shows the **Check** expression. This is OK for modelers, but may be obscure for end users. If you call the **Error()** function (see “Error(message)” on page 368) in the check, it displays the message you pass to **Error()** instead of the default warning. Using this, you can craft a more helpful message. The warning gives the same options:

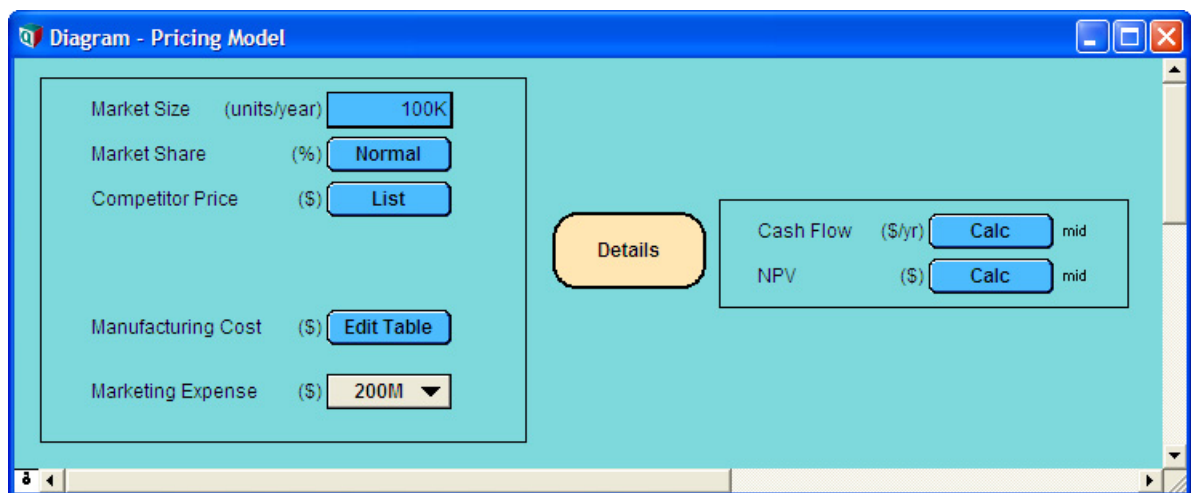


To disable checking You can disable all value checking by unchecking **Check value bounds** in the **Preferences** dialog from the **Edit** menu (see “Preferences dialog”). This check box is checked by default.

For a complex model, you can make it easier to use, especially by other people, by creating a user interface. A user interface is simply a diagram containing input and output nodes. These inputs and outputs are selected variables that users may change (inputs) or view (outputs). By gathering input and output nodes into a single user interface diagram, users have quick access from a central window, even if the underlying variables may be located in other parts of the module hierarchy.

Input nodes allow the user to see and change the values of variables directly in a diagram. Input nodes may be a field to enter a number or text value, a button that opens an edit table or probability distribution, or a pulldown menu. Output nodes show atoms (single numbers or text values) in the diagram, and show a button for uncertain or array-valued variables, so that users can open tables or graphs with a single click.

Input and output nodes are a kind of alias node linked to the original node. Input and output nodes usually show the title and units of a variable to the left of the input or output field or button:



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.

This diagram shows input nodes on the left side and output nodes on the right side. To see the details of the model, you would double-click the **Details** node to open up its diagram.

See Chapter 1, “Examining a Model.”

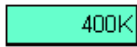
Using input nodes

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”). 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 “An alias is like its original”).

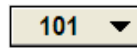
The type of definition of the original variable determines the appearance of the input node (see “The Expression popup menu”). If you want your users to be able to change the type of definition, instruct them on how to open an *Attribute* view or **Object** window and use the **Expression** popup menu.

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 “Automatic checking for valid values”); the check is performed on input of a new value.

Input popup menu



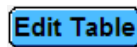
A choice displays as an input popup menu. To create an input menu for an input node, see “Creating a choice menu”.

List



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

Edit table



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

Probability distribution



A probability distribution displays a button with the name of the distribution (see “Defining a variable as a distribution”).

Creating an input node

To create an input node from a variable:

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

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

Creating a choice menu

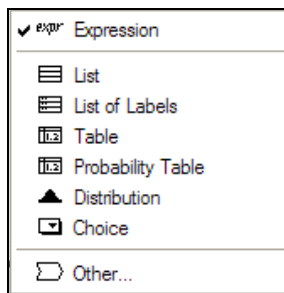
For the classes of nodes that may be used for parametric analysis, such as decision and chance, the **Expression** popup menu includes the **Choice** option. The **Choice** option provides a way to offer the user a choice of selecting one or all values from a list.

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.

- Click the **Expression** popup menu and select the **Choice** option. Click **OK** to "Replace current definition with a Choice?" Click **OK** to "Replace current definition?"



- 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 "Choice(i, n, inclAll)").

Tip 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).

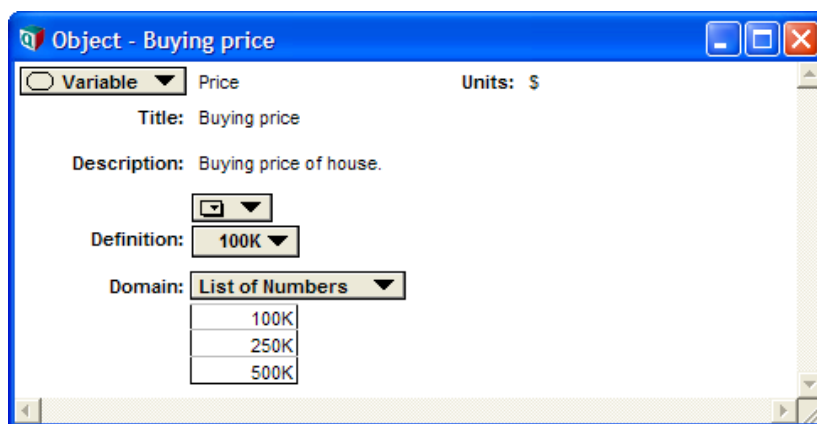
Tip To hide the **All** option on the popup, enter **inclAll=False** as the third parameter in the **Object Finder** dialog.

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").



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 — i.e., whether 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 views”).

61.73 mid

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

Result mid

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

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

Calc

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

Creating an output node

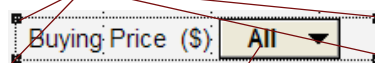
To create an output node from a variable:

1. Make sure you are in edit mode.
2. In a **Diagram** window, select the node of the variable for which you wish to create an output node.
3. Select **Make Output Node** from the **Object** menu. The output node will appear in the diagram next to the selected node.
4. Move the output node to the location you want.
5. 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 “Formatting Numbers, Tables, and Graphs”).

Resizing controls

Drag corners to resize node



Drag left or right to resize control

You can resize input and output nodes by dragging their corner handles, just like other nodes. But for these, it's usually most convenient to deselect **Resize centered** from the **Diagram** menu so you can align them either along their right edges, or both edges.

You can also drag the left edge of the control field, button, or menu left or right to change its width. This is especially useful for choice menus when you may want to expand the width to be large enough for the widest menu option.

When using a pull-down menu containing long text values, you may wish to adjust 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 cursor to the left or right.

Input and output nodes and their original variables

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 “To edit an attribute”). 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.

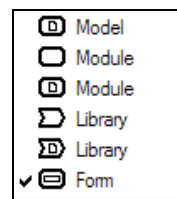
To change the appearance of an input or output node alone, use the **Set Node Style** and **Show Color Palette** options from the **Diagram** menu (see “Node Style dialog” and “Recoloring nodes or background”). When you use these options to change the appearance of an input or output node, its original 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 module

1. Make sure a diagram window is active with the edit tool selected.
2. Drag the module icon from the node toolbar and position it in the diagram.
3. Type in a title for the module — for example, **User interface**.
4. Open the **Attribute** panel at the bottom of the diagram window.
5. With the new form module still selected, press to open the **Attribute** popup menu, and select **Class**.
6. The class **Module** appears in the **Attribute** panel. Press to open a popup menu of module classes:





7. Select **Form** from the menu.

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”). 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 may be in the same or different diagrams.

2. In the toolbar, click  to enter arrow mode.
3. **To create an input node for variable X**, draw an arrow from the form node to **X**. It creates an input node for **X** inside the form module.
4. **To create an output node for variable Y**, draw an arrow from **Y** to the form node. It creates an output node for **X** inside the form module.
5. When you have finished creating input and output nodes, double-click the form node to open its diagram window.
6. In the toolbar, click  to enter edit mode.
7. Rearrange and resize the input and output nodes for clarity. It is sometimes clearest to arrange the input nodes on the left side and the output nodes on the right side.

A form module is like any other module, except when you draw arrows into or out of a form module, it creates outputs or inputs, instead of normal alias nodes in the module. But, you can also create standard variables and modules inside a form. If you have too many nodes to fit comfortably in a single diagram, you may wish to organize them into additional modules (which need not be forms) to enclose related groups of inputs and outputs.

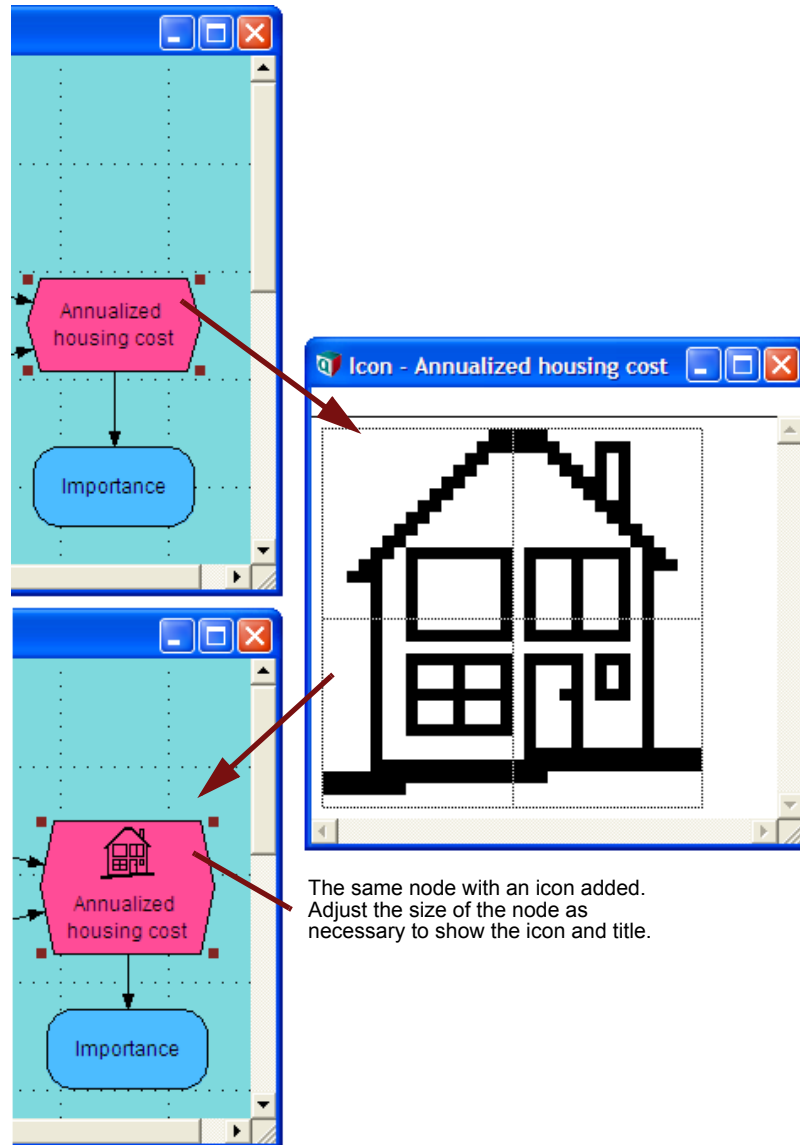
Adding icons to nodes

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 the pixel.
- To draw a line or curve hold down the mouse button while you move the cursor. If the starting pixel of the line or curve is black the line or curve will be black; if the starting pixel of the line or curve is white the line or curve will be white.
- To set the node's icon, click the **Accept button** .
- To restore the original icon in the window (or to clear the window if there was no previous icon), click the **Revert button** .

You can copy and paste an icon from one place in a model to another using the standard **Copy** (*Control+c*) and **Paste** (*Control-v*) commands. You can delete an icon from a node by selecting it and using the **Cut** (*Control+c*) command or the *Delete* key.

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 (*Control+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 (*Control+v*) the graphic from the clipboard.

When you paste a graphic into the diagram window, a special node of class **picture** is created. Picture nodes can be placed on top of **variable**, **module**, and **function** 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”) 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”) 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. To place the decision node underneath the other nodes, select the decision node while in edit mode, right mouse click and select the **Send to Back** command from the pop-up menu.
- Create a text node by dragging a text node from the text button  on the toolbar. Use the **Node Style** dialog box (see “Node Style dialog”) to add a fill color and border to the node.

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”) to change the font or to change the background from transparent to filled.

Models in XML file format

By default, Analytica 4.0 saves models in its own slot-filler format. You can also save Analytica models in XML format. 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 4.0 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 4.0 with users who are using earlier releases, 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 4.0 and described in the *Upgrade Guide*.

Sample old file format

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

```
{ From user Richard Morgan, Model Sample_old_file_format ~~
at Jun 1, 2007 3:56 PM}
Softwareversion 3.1.0
```

```
Model Sample_old_file_format
Title: Sample of old file format
Author: Richard Morgan
Date: Jun 1, 2007 11:55 PM
Savedate: Jun 1, 2007 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="Richard" project="Sample_XML_file_format" generated=" Jun
1, 2007 3:57 PM" softwareversion="4.0.0" software="Analytica">

<model name="Sample_XML_file_format">
  <title>Sample XML file format</title>
  <author>Richard Morgan</author>
  <date> Jun 1, 2007 11:55 AM</date>
  <saveauthor>Richard Morgan</saveauthor>
  <savedate>Fri, Jun 1, 2007 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

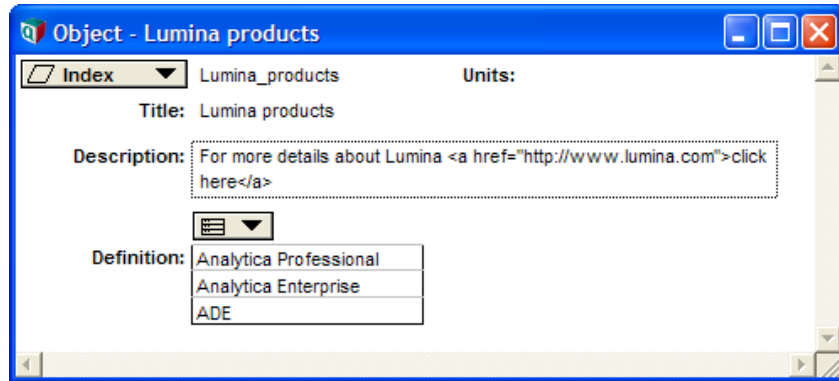
Any *description*, or other text attribute of a variable or other object, can contain a hyperlink to any Web page. This is useful for linking to detailed explanations, data, or references for a model, or even to related downloadable Analytica models. In browse mode, hyperlinks appear conventionally underlined in blue. When you click 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

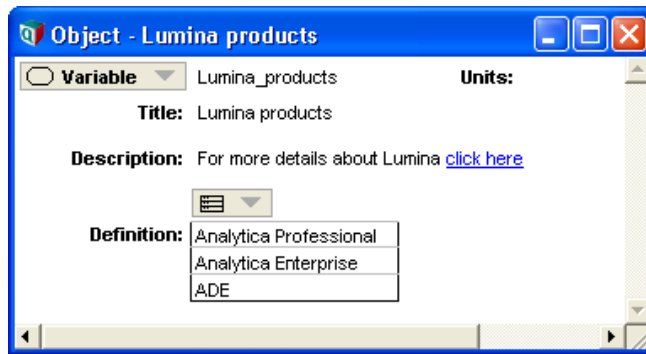
```
<a href="http://www.lumina.com">Click here</a>
```

When you switch to browse mode the HTML code will display as a hyperlink.

In edit mode



In browse mode

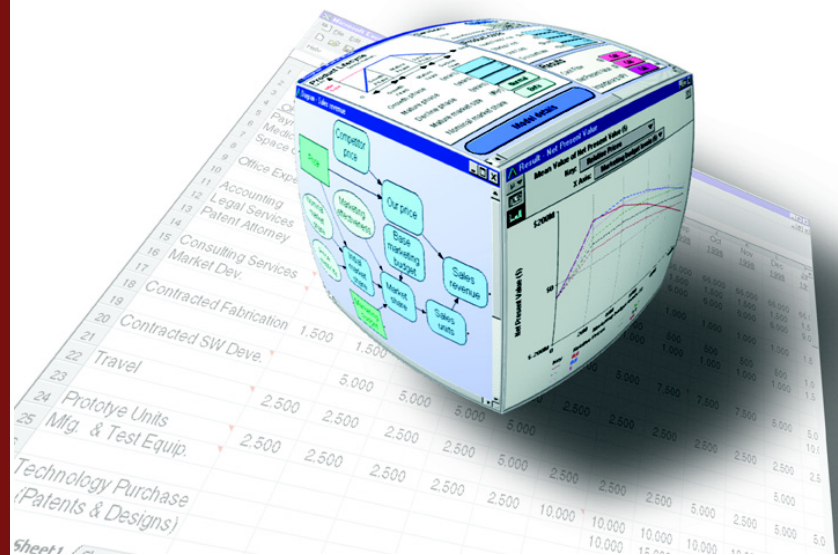


Chapter 10

Using Expressions

This chapter tells you how to:

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



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, -16
Decimals	32.5, .0002, 0.000012345
Suffix	250K, 10.5M, 10.5m, 22%
Exponential form	5.3e+11, 1e20, 4.5632e-25

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

$$5e4 = 5 \times 10^4 = 50,000$$

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

- 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 87).

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
12	T	Tera or Trillion	-12	p	pico
15	Q	Quad	-15	f	femto

Tip 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
-1 * 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 148.

Null, Undefined, NAN, and INF

Analytica may return the following system constants:

Constant	Meaning
Undefined	A value has never been defined or is uncomputed.
Null	There is no such item.
NAN	The result is numeric, but not a real number or infinity; (e.g., <code>Sqrt(-1)</code> or <code>0/0</code>).
INF	Infinity or a real number larger than can be represented (e.g., <code>1/0</code>).
-Inf	Negative infinity or a number smaller than can be represented (e.g., <code>-1/0</code>).

Functions such as `Slice()`, `Subscript()`, `Subindex()`, and `MDTable()` may return `Null`—for example, when trying to `Slice()` out the n th element 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 attempting to use the value of an optional parameter in a user-defined function that hasn't been provided by the caller. `Undefined` and `Null` are treated as the same in evaluation of expressions.

You can test for `Null` using the standard `=` or `<>` operators, or you can use `IsUndef()`.

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 specify a text value by enclosing text in single quotes, or in double quotes, for example:

```
'A', "A25", 'A longish text - with punctuation.'
```

A text value can contain any character, including any digit, comma, space, and new line. To include a single quote(') or apostrophe, 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” and “List vs. list of labels”) or edit table (see “Creating an array with an edit table”). Analytica displays text values in results without the enclosing quotes. Also see “Converting a number to text”.

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, respectively. For example:

```
False or True → True
1 and 0 → False
1 or 0 → True
```

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 formats”).

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

Operator	Meaning	Examples
*	multiplied by	$3 * 2 \rightarrow 6$
/, ÷	divided by	$3 / 2 (= \frac{3}{2}) \rightarrow 1.5$
^	to the power of	$3 ^ 2 (= 3^2) \rightarrow 9$
^fraction	root (fractional exponent)	$4 ^ .5 (= 4^{\frac{1}{2}}) \rightarrow 2$

Comparison operators Comparison operators apply to numbers and text values and produce Boolean values.

Operator	Meaning	Examples → (1 = true, 0 = false)
<	less than	$2 < 2 \rightarrow 0$ $'A' < 'B' \rightarrow 1$
<=	less than or equal to	$2 <= 2 \rightarrow 1$ $'ab' <= 'ab' \rightarrow 1$
=	equal to	$100 = 101 \rightarrow 0$ $'AB' = 'ab' \rightarrow 0$
>=	greater than or equal to	$100 >= 1 \rightarrow 1$ $'ab' >= 'cd' \rightarrow 0$
>	greater than	$1 > 2 \rightarrow 0$ $'A' > 'a' \rightarrow 1$
<>	not equal to	$1 <> 2 \rightarrow 1$ $'A' <> 'B' \rightarrow 1$

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:

1. Digits precede (are smaller than) letters, so

`'9' < 'A' → 1 (True)`

2. 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)`

3. 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()` 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 (::) Since new versions of Analytica introduce new functions that did not exist in previous releases of Analytica, it is possible that a model created in a previous release may contain a variable or other object with the same identifier as a new built-in variable or function. In this situation, an identifier name appearing in an expression may be ambiguous.

Prepending **::** to the name of a built-in function causes the reference to always refer to the built-in function. 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

<code>Irr(Payments, Time)</code>	User's <code>Irr</code> function
<code>::Irr(Payments, Time)</code>	The built-in function

Operator binding precedence

A precedence hierarchy resolves potential ambiguity when evaluating operators and expressions. The precedence for operators, from most tightly bound to least tightly bound is:

1. parentheses ()
2. function calls
3. Not
4. @I, \A, \[!A, #R.
5. A.I
6. A[l=x]
7. Attrib of Obj
8. ^
9. - (unary, negative)
10. *, /
11. +, - (binary, minus)
12. m..n
13. <, >, <=, >=, =, <>

14. And, Or
15. & (text concatenation)
16. :=
17. If ... Then ... Else, Ifonly ... Then ... Else, Ifall ... Then ... Else
18. Sequence of statements separated by semicolons, sequence of elements or parameters separated by commas

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)
```

IF a THEN b ELSE c

This conditional expression returns **b** if **a** is true (1) or **c** if **a** is false (0), for example:

```
Variable A := 1M
Variable B := 1
IF X > Y THEN X ELSE Y
```

will return the larger of X and Y.

It is possible to omit the ELSE clause:

```
IF X > Y THEN X
```

But, if the condition is false, it will give a warning. If you ignore the warning, it returns NULL.

Conditional expressions get more interesting when they work on arrays. See “IF a THEN b ELSE c with arrays”.

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. Other more advanced or specialized functions are described in Chapter 13, “Other Functions.” 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 (see “Building Functions and Libraries”)

Position-based function calls The conventional position-based syntax to call a function has this form:

FunctionName(param1, param2, ...)

The function name is followed by a comma-delimited list of parameters enclosed between parentheses. In most cases, parameters can themselves be expressions built out of constants, variable names, operators, and function calls. 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)
```

Name-based function calls Analytica also offers name-based parameter calling for most functions with multiple parameters.

Math functions

These functions can be accessed under the **Definition** menu **Math** command, or in the **Object Finder** dialog box, Math library.

Abs(x) Returns the absolute value of **x**.

```
Abs(180) → 180
Abs(-210) → 210
```

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
```

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
```

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
```

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
```

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
```

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

Mod(x, y) Returns the remainder (modulus) of **x/y**.

Mod(7,3) → 1 **Mod(12,4)** → 0
Mod(-14,5) → -4

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.

Cos(x), Sin(x), Tan(x) Returns the cosine, sine, and tangent of **x**, **x** assumed in degrees.

Cos(180) → -1
Cos(-210) → -0.866
Sin(30) → 0.5
Sin(-45) → -0.7071
Tan(45) → 1

Arctan(x) Returns the arctangent of **x** in degrees.

Arctan(0) → 0
Arctan(1) → 45
Arctan(Tan(45)) → 45

See also “Arccos(x), Arcsin(x), Arctan2(y, x)” on page 213.

Degrees(r), Radians(d) Degrees gives degrees from radians, and radians gives radians from degrees:

Degrees(Pi/2) → 90
Degrees(-Pi) → -180
Radians(-90) → -1.57079633
Radians(180) → 3.141592654

Numbers and text

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: Thursday, December 25, 2008'

Tip 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 Long 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()** 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 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 "Evaluate(t)".)

An alternative method, for converting text to a number is to use the **Coerce Number** qualifier on a user-defined function. (See "Parameter qualifiers".) For example, you could define a user-defined function such as:

```
ParseNum(X: Coerce Number) := X
```

Datatype functions

Non-array values in Analytica may be numbers, text, or the special value **undefined** (and its equivalent **null**). The functions in this section, found on the **Special** sub-menu of **Definition**, can be used to determine the value type.

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
```

Isnan(x) Returns True if **x** is "not a number," i.e. **NAN**. **INF** and a regular number do not qualify, nor does a text or **NULL**.

```
0/0 → NAN
IsNaN(0/0) → True
IsNaN(5) → False
IsNaN(INF) → False
IsNaN('Hello') → False
```

x = NULL To test if **x** is **NULL**.

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, 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.

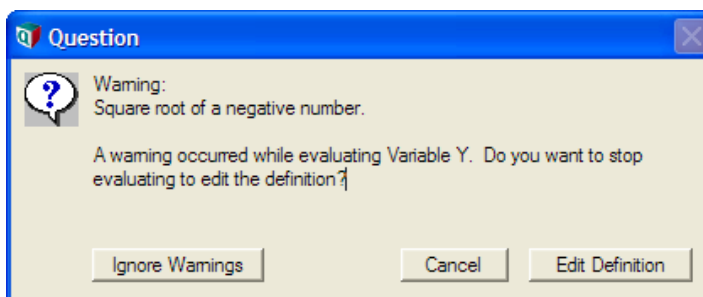
```
Isundef('hello') → False
Isundef(5) → False
Isundef(0/0) → False
Isundef(1/0) → False
Isundef(Subindex(Time*2,1000,Time)) → True
```

Note: In the last example, `Time*2` does not contain the value 1000, so `Subindex()` returns Null.

TypeOf(x) Returns the type of expression `x` as a text value, usually one of "Number", "Text", "Reference", or "Null".

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 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 `sqr(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
```

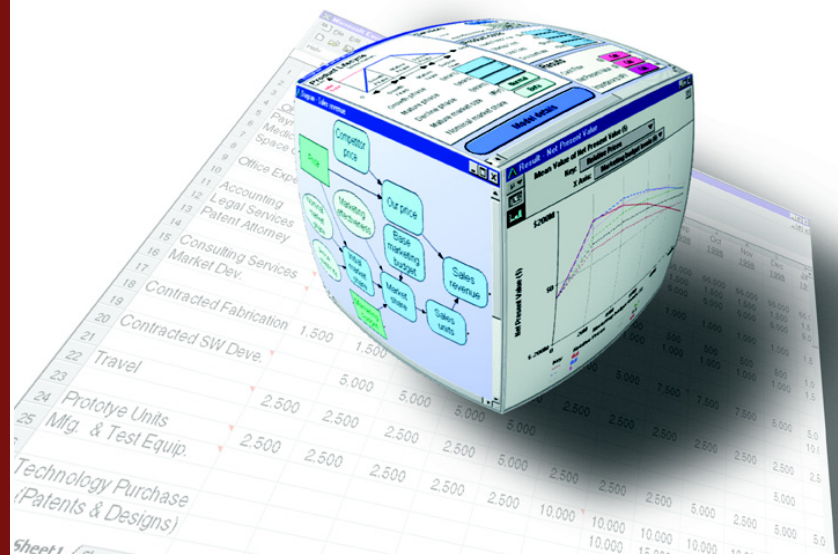
The commonly encountered conditions of "subscript or slice value out of range" are warnings with the return value of Null, for example:

```
Index I := 1..5  
Slice(I^2, I, 6) → Null
```


Chapter 11

Arrays and Indexes

This chapter introduces how to define and work with arrays and indexes. In particular, it explains Intelligent Arrays™, Analytica's powerful and convenient features for working with multidimensional arrays.



The value of a variable may be an **atom** — a single number, Boolean, text value, or reference — or it may be an **array** — a collection of such 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 large models. Each dimension of an array is identified by an **index variable**. You can extend a dimension simply by adding elements to its index. Or you can add a dimension to an array variable. These changes to dimensions will automatically carry through the rest of the model, often with no need to make other modifications.

There are some subtleties to the effective use of arrays. Your past experience with spreadsheets or programming languages may actually 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 sections, “Introduction to arrays” and “Operations on arrays”. The rest of this chapter gives 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. “More Array Functions”, describes the special functions that create and operate on arrays.

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 indexed by **car_type** with two values, \$2.70 for the small car (which uses regular gasoline) and \$2.90 for the large car (which uses premium gasoline):

Mid Value of Fuel price per gallon (\$)	
small car	\$2.7
large car	\$2.9

Maintenance_cost is a two-dimensional array, indexed by **car_type** and by **Year**:

Mid Value of Maintenance cost					
	1	2	3	4	5
small car	300	300	500	1000	1400
large car	700	700	700	800	900

The small car is cheap to maintain initially, but it gets more expensive than the large car after three years as its components start to wear out and need replacing.

Tip You can swap the rows (**car_type**) and columns (**year**) by using the row or column popup menus (see “Index selection”).

Intelligent Arrays™ refers to the full set of features in Analytica for handling array abstraction. See “Intelligent Arrays™” for more information.

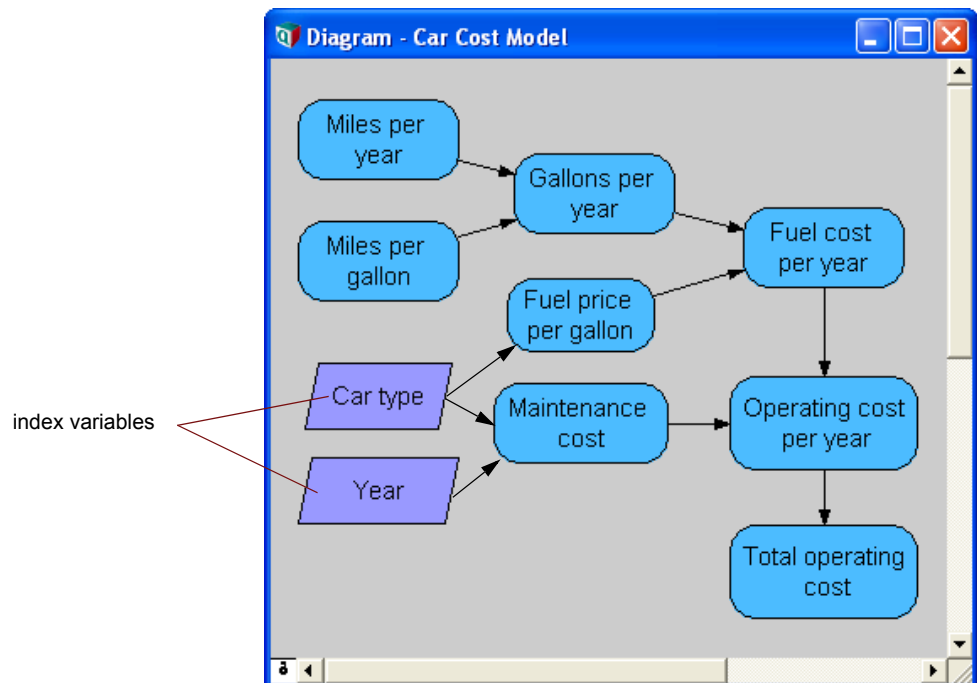
What is an index? Each dimension of an array is identified by an index variable. The index variable holds the possible values — a list of numbers or a list of labels. In the examples above, **car_type** is a list of labels, “small car” and “large car.” **year** is a list of numbers.

Car type:	Year:
small car	1
large car	2
	3
	4
	5

To create an index, see “Creating an index”.

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 This diagram of a **car_cost_model** includes the variables described above. The two indexes appear as purple parallelograms:



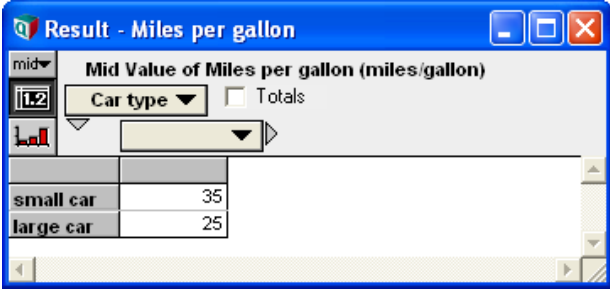
Fuel_price_per_gallon is the destination of an arrow from **Car_type** because it is defined as an array indexed by **Car_type**. Similarly, **Maintenance_cost** has arrows from **Car_type** and from **Year**, because it is indexed by both.

Tip By default, Analytica does *not* show arrows to and from index variables. This often makes diagrams simpler by avoiding excessive numbers of arrows. But, you *can* display these arrows, as above, by checking **Show arrows to/from Indexes** in the **Diagram Style** dialog from the **Diagram** menu (see “Diagram Style dialog”).

Viewing an array as an edit table

An **edit table** looks similar to a result table with the difference that you can edit the value of each cell, and sometimes also its index values. If you select a variable defined as an edit table and click the edit definition button , you will see its edit table.

`Miles_per_gallon` →



Mid Value of Miles per gallon (miles/gallon)	
small car	35
large car	25

To create or edit an array with an edit table, see “Creating an array with an edit table”.

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 automatically abstracts 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

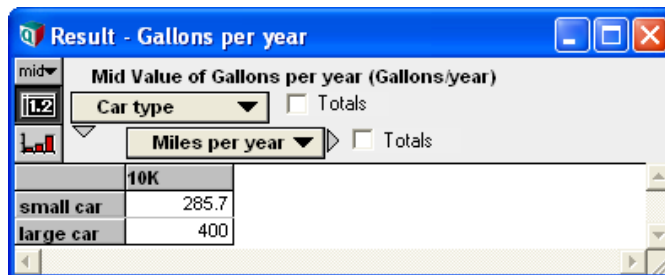
Arithmetic operations and simple functions generalize straightforwardly when they are applied to arrays, according to the dimensions of the arrays. This section gives some simple examples.

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.

```
Miles_per_gallon : Table( Car_type, Miles_per_year ) ( 35, 25 )
```

```
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:



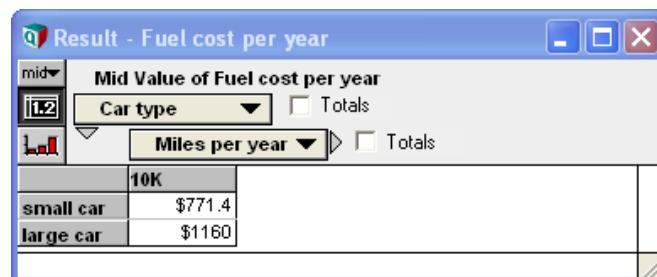
Mid Value of Gallons per year (Gallons/year)	
Car type	Miles per year
small car	285.7
large car	400

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.

```
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:



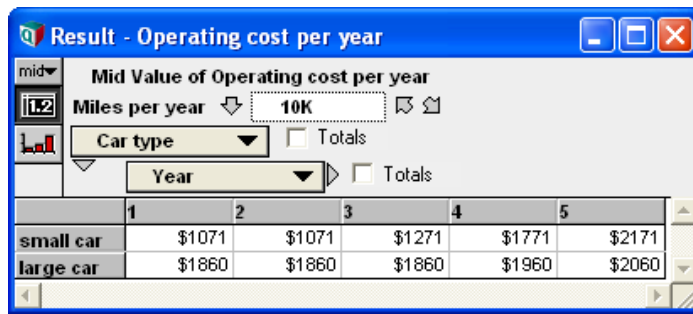
Mid Value of Fuel cost per year	
Car type	Miles per year
small car	\$771.4
large car	\$1160

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.

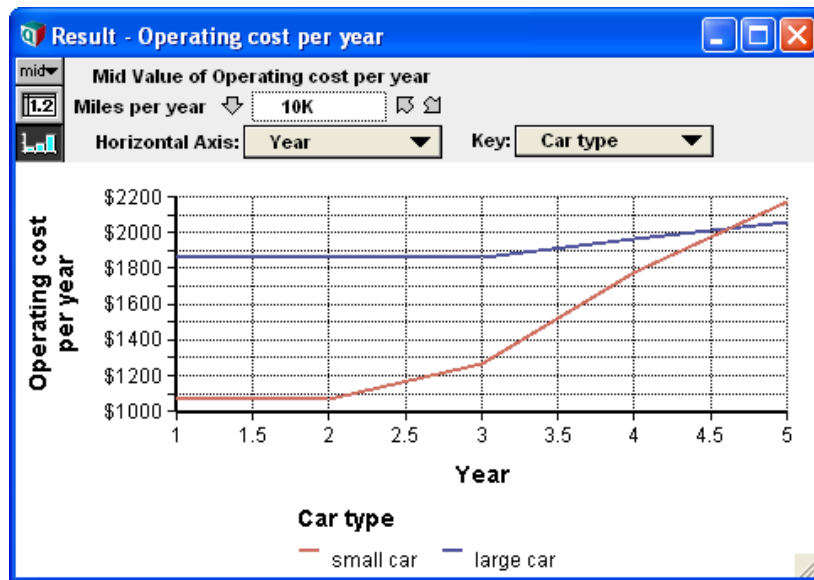
```
Op_cost_per_year:
Fuel_cost_per_year + Maintenance_per_year
```

`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 three years and grow to be larger in the fifth year, crossing over just after the fourth year.

Summing over an index variable

The `Sum()` function sums an array over one index, giving a result without that index.

```
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:

	10K
small car	\$7357
large car	\$9600

Tip 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”).

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.

`Miles_per_year` is redefined as a list (see “Create a list”):

```
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.

`Miles_per_gallon` remains an array indexed by `Car_type`. `Gallons_per_year` remains defined as follows:

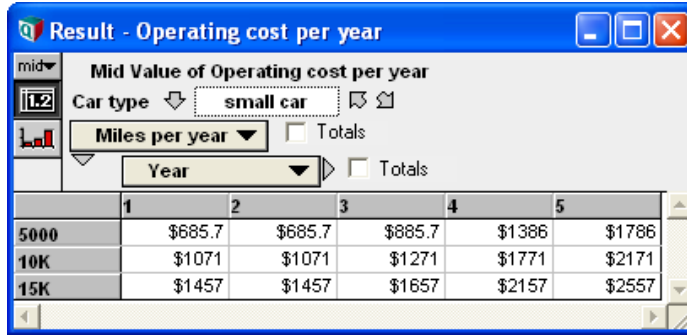
```
Gallons_per_year: Miles_per_year / Miles_per_gallon
```

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”):

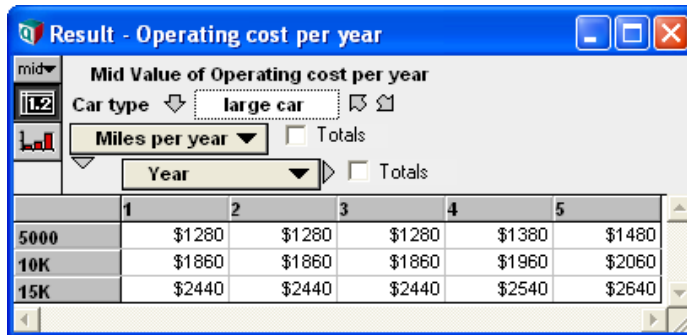
	5000	10K	15K
small car	142.9	285.7	428.6
large car	200	400	600

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 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.

IF a THEN b ELSE c with arrays

The **IF a THEN b ELSE c** construct (introduced in “IF a THEN b ELSE c”) generalizes appropriately if any or all of **a**, **b**, and **c** are arrays. In other words, it fully supports Intelligent Arrays. For example, if condition **a** is an array of Booleans (true or false values), it returns an array with the same index, containing **b** or **c** as appropriate:

```
Variable X := -2..2
If X > 0 THEN 'Positive' ELSE IF X < 0 THEN 'Negative' ELSE 'Zero' →
X ▶
```

	-2	-1	0	1	2
	'Negative'	'Negative'	'Zero'	'Positive'	'Positive'

If **b** and/or **c** are arrays with the same index(es) as **a**, it returns the corresponding the values from **b** or **c** according to whether **a** is true or false:

```
IF X >= 0 THEN Sqrt(X) ELSE 'Imaginary' →
```


x ►

	-2	-1	0	1	2
	'Imaginary'	'Imaginary'	0	1	1.414

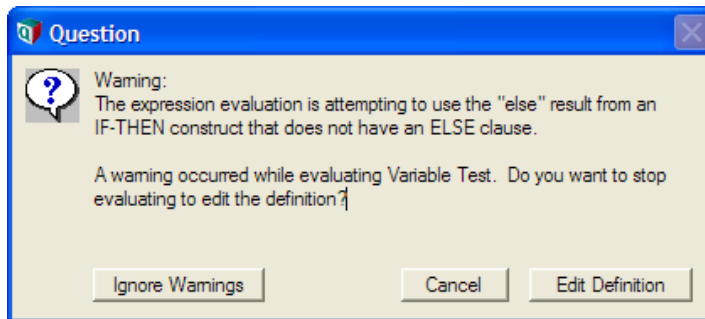
In this case, the expression `Sqrt(x)` is also indexed by `x`. The If expression evaluates `Sqrt(x)` for each value of `x`, even the negative ones, which return `NAN`, even though they are replaced by `'Imaginary'` in the result.

To avoid evaluating all of b or c

Sometimes, you want to avoid evaluating elements of `b` or `c` corresponding to elements of `a` that give errors or NULL results, to avoid wasting computation time on intermediate results that won't be used in the final result, or because the computations will cause evaluation errors, not just warnings. In such cases, you can use explicit iteration, using a FOR or While loop over index(es) of `a`. See "Begin-End, (), and ';' for grouping expressions"

Omitting ELSE

If you omit the `ELSE c` part, it usually gives a warning when it is first evaluated.



If you click **Ignore Warnings**, it returns NULL for elements for which `a` is false:

```
IF x >= 0 THEN Sqrt(x) →
```

x ►

	-2	-1	0	1	2
	<<null>>	<<null>>	0	1	1.414

After you have clicked **Ignore Warnings**, it will not give the warning again, even after you save and reopen the model.

Tip

Usually, you should omit the `ELSE c` part of an IF construct only in a compound expression (see "Begin-End, (), and ';' for grouping expressions"), when the `IF a THEN b` is not the last expression, but rather is followed by ";". In this situation, the NULL result is not part of the result of the compound expression, and it gives no warning:

```
BEGIN
  VAR A := Min([X,Y]);
  IF A<0 THEN A:=0;
  Sqrt(A)
END
```

The dimensions of the result

If `a` is an array containing some True and some False values, `IF a THEN b ELSE c`, evaluates both `b` and `c`. The result contains the union of the indexes of all operands, `a`, `b`, and `c`. But, if `a` is an atom or array whose value(s) are all true (1), it does not bother to evaluate `c` and returns an array with the indexes of `a` and `b`. Similarly, if all atoms in `a`

are false (0), it does not bother to evaluate **b** and returns an array with the indexes of **a** and **c**. This means that the values in the condition **a** can affect whether **b** and/or **c** are evaluated, and which indexes are included in the result.

IFALL a THEN b ELSE c If you don't want the dimensions of the result to vary with the value(s) in **a**, use the **IFALL** construct. This is like the **IF** construct, except that it always evaluates **a**, **b**, and **c**, and so the result always contains the union of the indexes of all of three operands.

IFALL requires the **ELSE c** clause. If omitted, it gives a syntax error.

IFONLY a THEN b ELSE c **IFALL** has the advantage over **IF** (and **IFONLY**) that the dimensions of the result are always the same, no matter what the values of the condition **a**. The downside is that if **a** is an array and all its atoms are True (or all are False), it wastes computational effort calculating **c** (or **b**) even though its value is not needed for the result. **IFALL** also may waste memory (and therefore also time) by including the index(es) that are only in **c** (or **b**) even though the result has the same values over those indexes. The standard **IF** construct may also waste some memory when all of the values of array **a** are True (or all are False), because the result includes any index(es) of **a** that are not indexes of **b** (or **c**), even though the result must be the same over such index(es). In situations, where this is a concern, you may use a third conditional construct, **IFONLY a THEN b ELSE c**. Like **IF**, when all atoms of **a** are True (or all False), it evaluates only **b** (or only **c**). But, unlike **IF**, the result in these cases does include any index(es) of **a** that are not indexes of **b** (or **c**, respectively). Thus, **IFONLY** can be more memory-efficient.

In the vast majority of cases, you may just use **IF** without worrying about **IFALL** or **IFONLY**. The only reason to use **IFALL** is if you don't want the dimensions of results to vary with values of inputs. The only reason to use **IFONLY** is when memory is tight and it's common for condition **a** to be all true or all false.

To summarize the differences between these three constructs: If condition **a** is an atom or array containing only true (only False) values, **IF** and **IFONLY** evaluate only **b** (only **c**), whereas **IFALL** always evaluates both **b** and **c**. The result of **IFONLY** contains the indexes of only **b** (only **c**). The result of **IF** contains the indexes of **a** and **b** (or **c**). The result of **IFALL** always contains the indexes of **a**, **b**, and **c**, and so its dimensions do not depend on the values of **a**.

If condition **a** is an array containing mixed true and false atoms, all three constructs behave identically: They evaluate **a**, **b**, and **c**, and the result contains the union of the indexes of **a**, **b**, and **c**.


IFALL requires the **ELSE** part. It is optional for **IF** and **IFONLY**, but recommended except when part of a compound expression, followed by ";",

Creating an index

An **index** is a class of variable used to identify a dimension of an array. The same index may identify the same dimension shared by many arrays. Sometimes, you may also use other classes of variables, such as a decision. Any variable defined as a list — one-dimensional array with no separate index — can serve as an index to an array. For clarity in your model, use an index variable whenever possible.

You create an index much like any other variable:

1. Select the edit tool  and open a **Diagram** window.

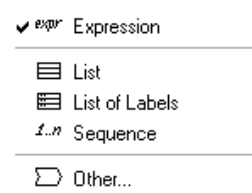
2. Drag the parallelogram shape  from the node palette to the diagram.
3. Type a title into the new index node.
4. You may define an index as a list (of numbers), list of labels, sequence, or other expression that generates a list. Here we define it as a list.

Create 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 “The Attribute panel” on page 23).
- In the **Attribute** panel, select **Definition** from the **Attribute** popup menu (see “Creating or editing a definition”) as the attribute to display.

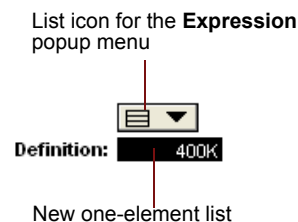
To create a list:

1. Press the **Expression** popup menu above the definition field and select **List** (of numbers) or **List of Labels** (for text).

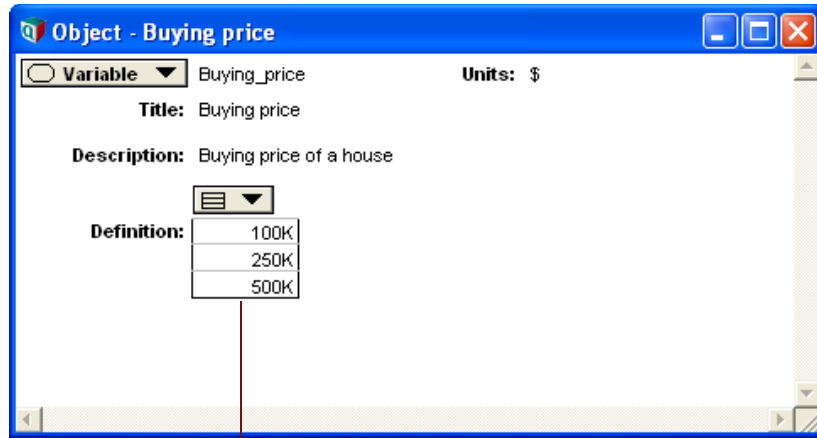


(If the variable already has a definition, Analytica confirms that you wish to replace it. Click **OK** to replace the definition with a one-element list.)

A one-element list is displayed in the definition field.



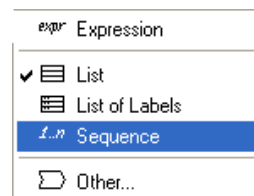
2. Select the element by clicking it.
3. Type in a number or expression (for **List**) or text (for **List of Labels**).
4. Press *Enter* and type in the next value.
5. Repeat step 4 until you have entered all the values you want.



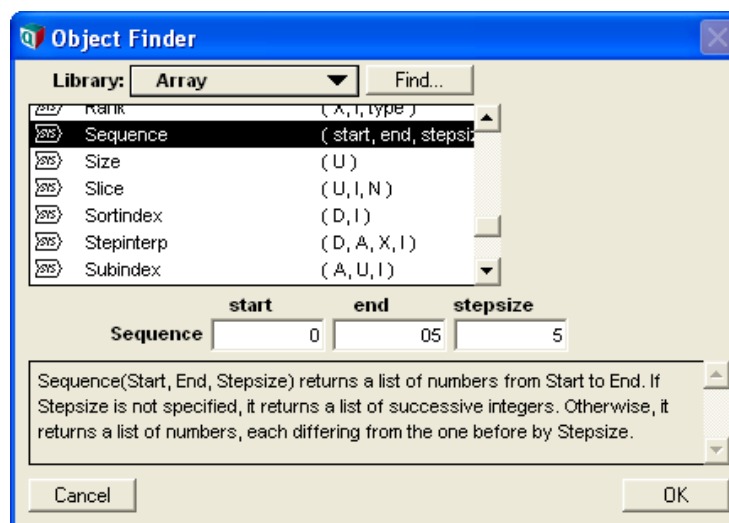
Values entered into a list

Autofill a list It gives the first cell of a list the default value of 1 (or the previous definition if it had one). When you press *Enter* or *down-arrow*, it adds a cell adding 1, or the increment between the two preceding cells, to the value of the preceding cell.

Create 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.




After entering the **start**, **End**, and **Stepsize** values, click **OK**; the definition field shows the **Sequence** button with its parameters.



Tip To edit the sequence, click 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.

The Expression view displays when you select  in the **Expression** popup menu.

List view

1
2
3
4
5

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

Alabama
Alaska
Arizona
Arkansas

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, for example:

['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

1
'Alabama'
2
'Alaska'

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

1
Alabama
2
Alaska

Expression view

```
['1', 'Alabama', '2', 'Alaska']
```

Tip 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).

Insert a cell 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** (*Control+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** (*Control+i*) from the **Edit** menu. The value of the last selected cell is duplicated in the new cells.

Delete a cell To delete one or more cells, select them and do one of the following:

- Choose **Delete Rows** (*Control+k*) from the **Edit** menu.
- Press *Delete*.

Tip 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”).

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 “Create a list”). You can also create a list within a variable definition using the constructs and functions described below.

[*u1, u2, u3, ... um*]

A list of expressions, separated by commas and surrounded by square 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 “Create a list”, according to the type of values, for example:

Examples `[8000, 12K, 15K]`
 `['VW', 'Honda', 'BMW']`

Tip If you draw an arrow from a node **x** into a variable **y** defined as list, it automatically adds **x** as the last element of the list. Or if **x** is already in the list it removes it. This is a handy way to make a list of variables.

m .. n

Returns an increasing sequence of integers from **m** to **n** if **n** \geq **m** or decreasing if **n** $<$ **m**. It is equivalent to **Sequence(m, n)**. For example,

`2003..2006` \rightarrow `[2003, 2004, 2005, 2006]`

Tip The parameters **n** and **m** must be atoms, that is single numbers. Otherwise, it would result in a non-rectangular array. See “Functions expecting atomic parameters” for information on using 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 “Create a list with the Sequence option”.

The expression **m .. n** using the operator “..” is a version of **Sequence(m, n, 1)**, so 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)` \rightarrow

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)` \rightarrow `[5, 4, 3, 2, 1]`

If **start** and **end** are not integers, and if **stepSize** is not specified, it rounds them first:

`Sequence(1.2, 4.8)` \rightarrow `[1, 2, 3, 4, 5]`

If `stepSize` is specified, it can create non-integer values from `start` to `end` by `stepSize`:

```
Sequence(0.5, 2.5, 0.5) → [0.5, 1, 1.5, 2, 2.5]
```

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: See “Example data” for the definition of example variables.

CopyIndex(i)

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.

Sortindex (d, i)

`d` is an array indexed by `i`. `Sortindex()` 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 the following :

```
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
--	-------	----	-----

Define Index_new as an index node:

INDEX Index_new := Sortindex(Maint_costs)

Subscript(Maint_costs, Car_type, Index_new) →

	Honda	VW	BMW
	1800	1950	2210

Note: See “Example data” for the definition of example variables.

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

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

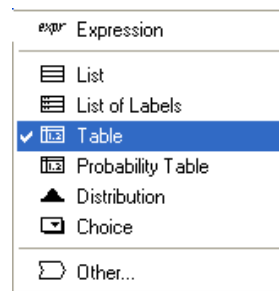
To define a variable as an edit table, you choose table from the expression menu above its definition:

1. Select the variable and open its definition by either:

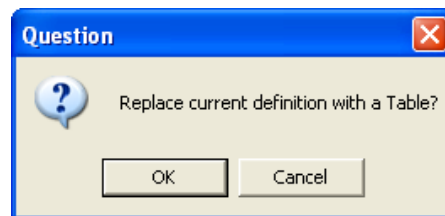
- The variable's **Object** window.
- The **Attribute** panel of the **Diagram** window (see “Creating or editing a definition”), and select **Definition** from the **Attribute** popup menu (see “Creating or editing a definition” on page 116), or
- Just press *Control-e*.

To create a table:

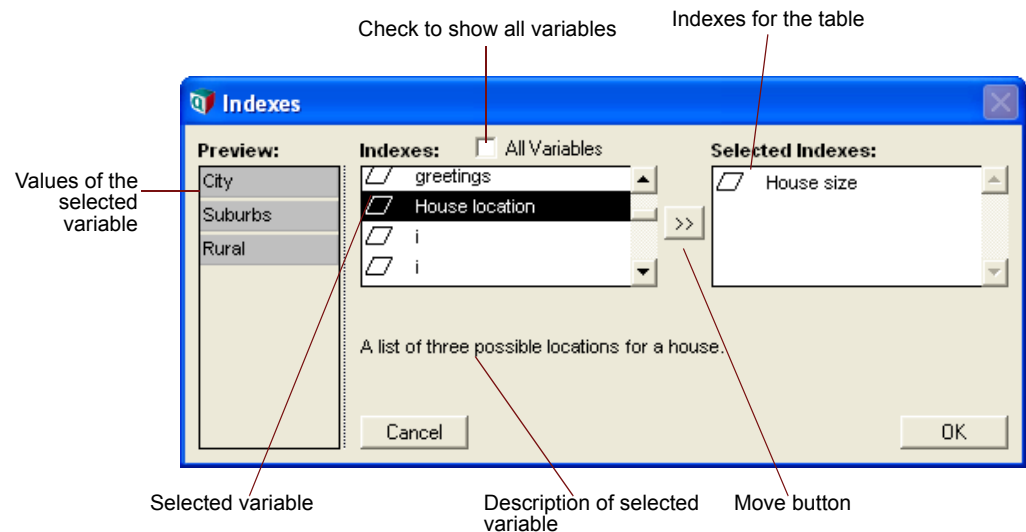
2. Press the **Expression** popup menu above the definition field and select **Table**:



If it already has a definition, click **OK** to confirm that you wish to replace it:



3. It opens the **Indexes** dialog so you can select the table's indexes (dimensions). It will already list under **Selected indexes** any index variables from which you have drawn an arrow to this variable. You may keep them, remove them, or add more indexes:



4. Select a variable from the **Indexes** list and click the move button **>>**, or double-click the variable, to select it as an index of the table. Repeat for each index you want.

- Click **OK** to create the table and open the **Edit Table** window for editing the table's values (see "Editing a table" on page 171).

Indexes dialog box

The **Indexes** dialog box contains these features (see figure above):

<i>Preview</i>	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."
<i>All Variables checkbox</i>	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.
<i>Selected indexes</i>	A list of all indexes already selected for this variable.
<i>New index</i>	Select to create a new index.

To create an index You can create an index variable in the course of creating a table, in the following way:

- Select **new index** from the variables list in the **Indexes** dialog box.
- Enter a title for the index.



- Click the **Create** button.
- To make the new index an index of the table, click the **>>** button.

Enter the values of the Index in the **Edit Table** window (see the following section).

To remove an index from an array

- Select the index from the **Selected Indexes** list.
- Click the **<<** button.

Removing an index leaves the subarray for the first item in that index as the value of the entire 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") or the **Sample()** function (see "Sample(x)").

Time is the index for dynamic simulation. It is the only index permitted for cyclically dependent modeling (see "Dynamic Simulation").

Editing a table

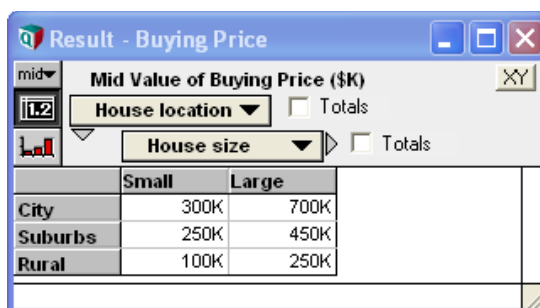
To open the **Edit Table** window, click the **Edit Table** button in either:

- The **Object** window (see “The Object window” on page 22)
- The **Attribute** panel of the diagram (see “The Attribute panel” on page 23)

In the **Attribute** panel, select **Definition** from the **Attribute** popup menu (see “Creating or editing a definition” on page 116).

The Edit Table window

The **Edit Table** window looks much like the **Result** window Table view (see “Viewing a result as a table” on page 30). The difference is that you can add indexes and edit the values in cells:



	Small	Large
City	300K	700K
Suburbs	250K	450K
Rural	100K	250K

Edit a cell Click the cell, and start typing to replace what’s in it. To add to what’s there, click three times to get a cursor in the cell, and type. You can use left-arrow and right-arrow keys to move the cursor. See “Shortcuts to navigate and edit a table” on page 174 for more. Press *Enter* to accept the value and to select the next cell, or click in another cell.

Tip You may enter an expression into a table cell with operations, function calls, and so on. But, if the expression is complex, it’s easier to enter it as the definition of a new variable, and then just type the name of the variable into the table.

Select a cell Click the cell once.

Select a range of cells Drag cursor from a cell at one corner of a rectangular region to the cell at the opposite corner.

Copy and paste a cell or region You can copy a cell or a range (two-dimensional rectangular region) of cells from a table or paste a cell into a region, just as with a spreadsheet:

1. Select the source cell or region as above, and choose **Copy** from the **Edit** menu or press *Control+c*.
2. Select the destination cell (or top-left cell of the destination region), and choose **Paste** from the **Edit** menu or press *Control+v*.

If you select a destination region that is *n* times larger (width, height, or both) than the source cell or region, it repeats the source *n* times in the destination.

Accept Click to accept all the changes you have made to the table. If you close a table, it will also accept the changes, unless you click .


Cancel Click to cancel all the changes you have made to the table since you opened it or last clicked .

- Copy and paste to or from a spreadsheet** Copy and paste of a cell or region works much the same from a spreadsheet to an Analytica table or vice versa. If necessary, you can easily pivot the Analytica table so its rows and columns correspond with those in the spreadsheet. It copies numbers in exponential format with full precision, no matter what number format is used in the table, so that other applications can receive them with no problems.
- Copy an entire table** To copy a table, including its row and column headers, click the top left cell to select the whole table. You can also copy a table with more than two dimensions: Select **Copy table** from the **Edit** menu. When you paste into a spreadsheet, it includes the name of the table, and all indexes, including the slicer index(es) for the third and higher dimensions.

Editing or extending indexes in an edit table

One convenient aspect of Intelligent Arrays is that you can edit and extend the indexes of an array right in the edit table, to change index values, insert or remove rows or columns, or, more generally, subarrays.

This works for an index defined as a list of numbers or list of labels. If an index is defined in another way — for example as $m \dots n$ or **Sequence**($x1, x2, dx$) — you must edit the original index. Either way, all edit tables that use the changed index are automatically modified accordingly. ([#xref See “Splicing” for details.]


To edit or extend an index, either you must be in edit mode  or the index variable you want to modify must have an input node. See “Creating an input node”.

- Edit a cell in a row or column index** Click the cell once to select its row or column. Then double-click the cell to select its contents. Start typing to replace the text or number. Remember, the same change will happen to all tables that use that index.
- Append a row** Click the bottom element of the row index to select the bottom row, and press the down-arrow key.
- Append a column** Click the rightmost element of the column index to select the right column, and press the right-arrow key.
- Insert a row or column**
1. Click the row or column header to select the row or column before which you wish to insert a new one.
 2. Select **Insert Rows** (or **Insert Columns**) from the **Edit** menu, or press or *Control+i*. Normally, the new row or column contains zeros, but see [#xref Splicing] for more.
- Delete a row or column**
1. Click the row or column header to select the row or column you wish to delete.
 2. Choose **Delete Rows** or **Delete Columns** from the **Edit** menu, or press *Control+k*.

Tip When you try to change an index that is used by more than one edit table, it warns you that “Changing the size of this index will affect table definitions of other variables.” and gives the option of whether to continue.


Tip If you intend your model to be used by end users with the Player or Power Player editions (that are fixed in browse mode) or intend to save your model as browse-only (if you have the Enterprise Edition), you can decide whether you want to allow your end users to be able to edit indexes as described above: Create an input node for each index that you want to let them change. Or don't, if you want to prevent them changing an index.

Add an index To add an index, use one of these two methods:

- Draw an arrow from the index to the node containing the table. When it asks if you want to add the index as a new dimension of the table, answer **Yes**.
- Click  in the edit table to open the **Indexes** dialog (see “Indexes dialog box”). Double-click the index you want to add, and click **OK**.

When adding a new dimension to an edit table, it copies the values of the table to each new subarray over the new index. Thus, the expanded table has the same values for every element of the new index. This has no effect on other edit tables.

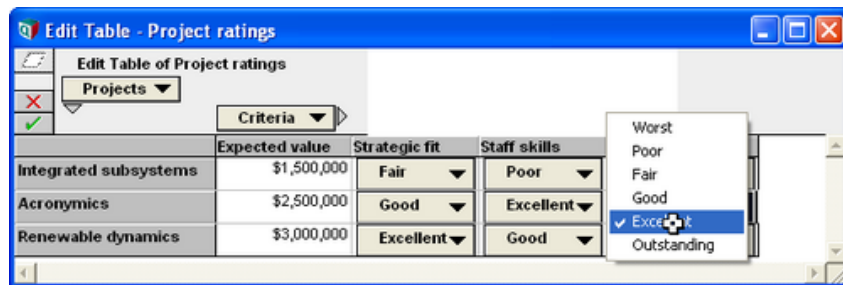
Remove an index To remove an index, use one of these two methods:

- Draw an arrow from the index to the node containing the table. When it asks if you want to remove the index as a dimension of the table, answer **Yes**
- Click  in the edit table to open the **Indexes** dialog (see “Indexes dialog box”). Double-click the index you want to remove, and click **OK**.

Tip When removing a dimension from an edit table, it replaces the entire table by its subarray for the first value of the index you are removing. It deletes all the rest. Be careful, because you will lose all the data in the rest of the table! This has no effect on other edit tables.

Choice menus in an edit table

You can include a dropdown (pulldown) menu in any cell of an edit table to let end users select an option for each cell. Here is an example, in browse mode:



You use the **Choice()** function in the edit table cells, similar to using **Choice** to specify a single menu for a variable (see “Choice(i, n, inclAll)”):

1. Create a variable **X** as an edit table, in the usual way, selecting **Table** from the *expr* menu above its definition.
2. Create an Index variable, e.g., **k**, containing the list of options you want to make available from the menu(s), usually as a list of numbers or a list of labels.
3. In the edit table of **X**, in edit mode, enter **choice(k, 1, 0)** into the first cell that you want to contain a menu. The second parameter 1 means that the first element of **k** is the default option. The third parameter 0 means that it does not show **All** as an option, normally what you want.

- Copy and paste `Choice(κ , 1, 0)` from the first cell to any others you want also to contain the menu. You can also use other indexes than κ if you want to include menus with other options. Here is an example viewed in edit mode, with dropdown menus in some but not all cells:

	Expected value	Strategic fit	Staff skills	Public goodwill
Integrated subsystems	\$1,500,000	Choice(Options,3,0)	Choice(Options,2,0)	Choice(Options,6,0)
Acronyms	\$2,500,000	Choice(Options,4,0)	Choice(Options,4,0)	Choice(Options,5,0)
Renewable dynamics	\$3,000,000	Choice(Options,5,0)	Choice(Options,4,0)	Choice(Options,1,0)

- Select **X**, then select **Make Input** from the **Object** menu to make an input node for it. Move the input node to a good location.

Tip The variable containing the edit table with menus *must* have an input node — otherwise, you won't be able to select from the menus or edit other cells in browse mode.

Shortcuts to navigate and edit a table

These mouse operations and keyboard shortcuts let you navigate around a table, select a region, and search for text. They are the same as in Microsoft Excel, wherever this makes sense. *Control-PgUp* and *Control-PgDn* are exceptions.

The *current cell* is highlighted, or the first cell you selected in a highlighted rectangular region. In a region, the *anchor cell* is the corner opposite the current cell. If you select only one cell, the Anchor and Current are the same cell.

Mouse operations

<i>Mouse Click</i>	Click in a cell to make it the current cell.
<i>Mouse Shift+Click</i>	Select the region from the previous anchor to this cell.
<i>Mouse drag</i>	Select the region from the cell in which you depress the left mouse button to the cell in which you release the button.
<i>Mouse wheel</i>	Scroll vertically without changing the selection.
<i>Control+mouse wheel</i>	Scroll horizontally without changing the selection.

Shortcuts to edit a table These shortcut keys speed up editing a table. Inserting and deleting rows and columns works only if the index(es) are defined as an explicit list, not if it is computed or a sequence:

<i>Downarrow</i>	If you have selected the last row, add a row.
<i>Leftarrow</i>	If you have selected the right column, add a column.
<i>Control-i</i>	If you have selected a row header, insert a row. If you have selected a column header, insert a column.
<i>Control+k</i>	Delete a selected row or column.

Control-v Paste copied cells from the clipboard into the table into the selected region. If you copy a region and have selected a single cell, it pastes into the region with the current cell as the top left, if it fits. If you paste a cell or region into a larger region, it repeats the copied material to fill out the destination region.

Search a table

Control+f Open the **Find** dialog to search for text in the table. It searches from the current cell and selects the first matching cell, if any.

Control+g Repeat the previous **Find**, starting in the next cell.

Arrow keys

arrow (right, left, up, down) Move one cell in the given direction. At the end of row, right arrow wraps to the start of the next row. At the end of the last row, it wraps to top-left cell. Similarly, for the other keys.

Shift+arrow Move the current cell one cell in the given direction. The Anchor cell stays put, causing the selected region to grow or shrink. It does not wrap.

Control+arrow Move to the end of row or column in the given direction.

Shift+Control+arrow Move current cell to the end of row or column in the given direction, leaving the Anchor where it is, causing the selected region to grow (or flip).

End, arrow Two key sequence. Same as *Control+arrow*.

End, Shift+arrow Two key sequence. Same as *Shift+Control+arrow*.

Home key

Home Move the anchor to the first column, and sets the current cell to be the anchor (so only one cell is selected). If you are in the row headers, moves the anchor and current to the first row.

Control+Home Select the top-left cell in the table. (Selects one cell.)

Control+End Select the bottom-right cell in the table. (Selects one cell.)

Shift+Control+Home Select the region between the anchor and the top left cell. (Leaves current as top left.)

Page key

Page Up, Page Down Move the current cell up or down by the number of rows visible in the window, and scrolls up or down to show that cell. (Selects one cell.)

Control+Page Up, Control+Page Down Move the current cell left or right by the number of columns visible in the window, scrolling horizontally to show the new current cell. (This is not the same as Excel, in which *Control+PgUp, Control+PgDown* toggle between worksheets. Since we don't have worksheets, these do something else useful.)

Shift+Page Up, Shift+Page Down Move the Current cell by the number of rows or columns that currently display on the screen, and scroll vertically by one page. Anchor stays the same, so that the currently selected region expands or shrinks by one page length.

Shift+Control+Page Up, Shift+Control+Page Down Same as *Shift+Page Up*, but horizontally rather than vertically.

Other keys

<i>Tab</i>	Move one cell right. Same as right arrow.
<i>Shift+Tab</i>	Move one cell left. Same as left arrow.
<i>Enter, Shift+Enter</i>	If editing, accept change, selection remains on cell just edited. If not editing, but in edit mode, current cell becomes anchor cell and begin editing that cell.
<i>Return</i>	If editing, accept changes. Move anchor down one cell, wrapping to top of next column if anchor is at the bottom. Set current cell to anchor (so only one cell is selected). If not editing, just move, do not start editing.
<i>Shift+Return</i>	If editing, accept changes. Move anchor cell up one cell, wrapping to bottom of previous column if at top. Set current to anchor, so only one cell is selected.
<i>Control+a</i>	Select all (body) cells. If a row/col header is selected, selects all rows/cols.

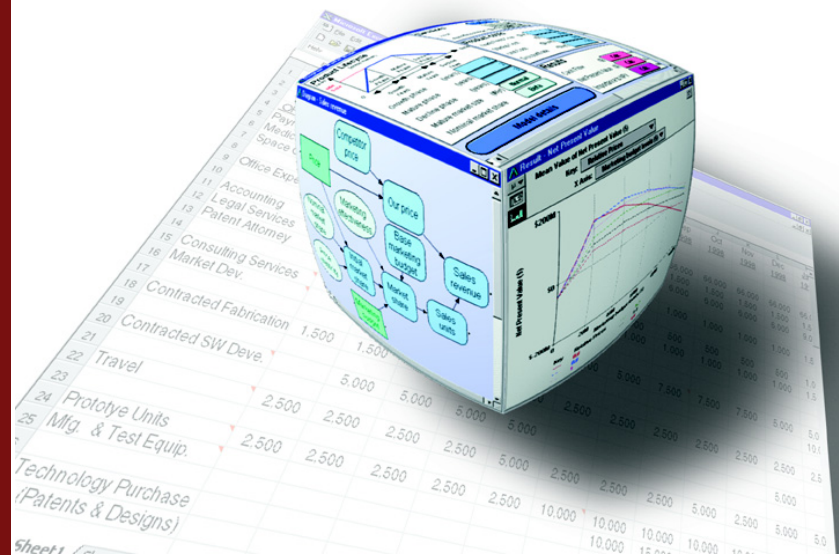
Chapter 12

More Array Functions

Analytica provides a large collection of built-in functions for performing common mathematical, financial, statistical, and array computations.

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.



This chapter describes Analytica's advanced built-in functions for dealing with arrays. It is organized by the type of function:

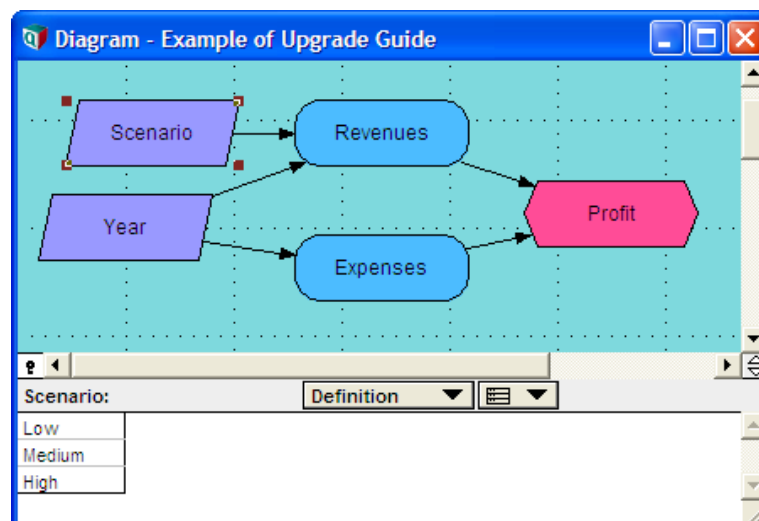
- Functions that create arrays (“Functions that create arrays”).
- Functions that reduce an array to another array with one fewer dimension (“Array-reducing functions”).
- Functions that return an array with the same number of dimensions as the input array (“Transforming functions”).
- Functions that select part or a slice of an array (“Selecting, slicing, and subscripting arrays”).
- Functions that interpolate values between array elements (“Interpolation functions”).
- Other array functions (“Other array functions”).
- Matrix functions for two-dimensional arrays (“Matrix functions”).
- Functions commonly used for financial computations (“Financial functions”).

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 indexed by **Year**, it computes the **Profits** for each **Year**, using the corresponding **Revenues** and **Expenses** for that **Year**, as in this example:

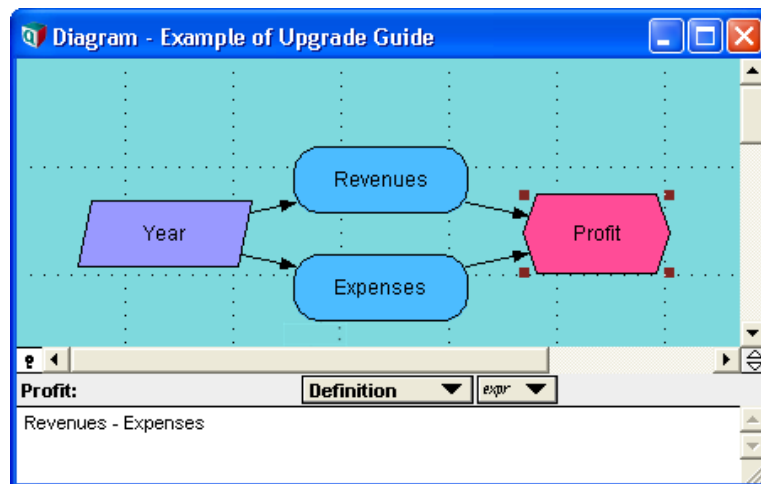


The figure below shows an influence diagram above and corresponding array values below.

	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**:

	2003	2004	2005	2006	2007
Low	\$600	\$625	\$650	\$700	\$750
Medium	\$600	\$700	\$800	\$850	\$900
High	\$600	\$750	\$900	\$1000	\$1100

The value of flexibility This flexibility is very convenient for the modeler. 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. The effort 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, and 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 conven-

tional 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 variable, 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.

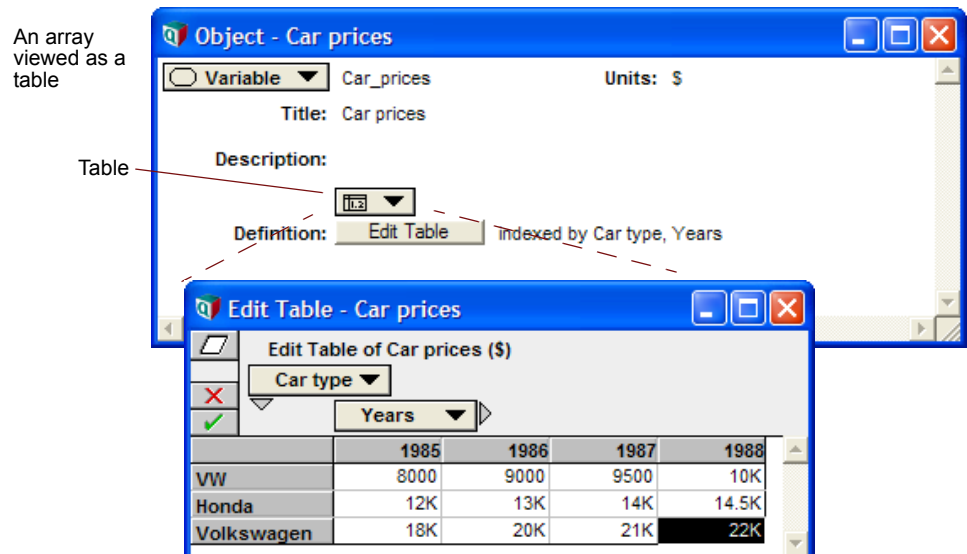
Exceptions to array abstraction

The vast majority of Analytica functions and constructs fully support Intelligent Arrays — that is, they automatically generalize from atomic values to multidimensional array — but a few do not. When you use these latter, you need to take special care to ensure that your models will array abstract conveniently when you add or modify dimensions. See “Ensuring array abstraction” on page 359 for details.

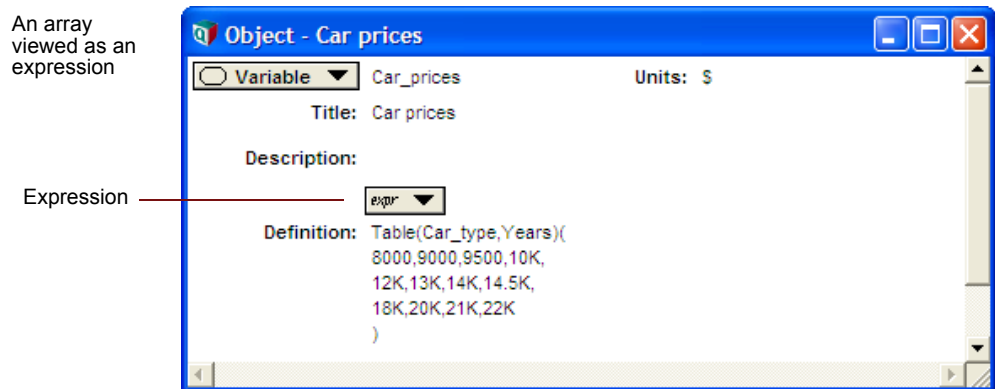
Functions that create arrays

Use the **Table** option in the **Expression** popup menu to define a variable as an array (see “Creating an array with an edit table”).

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()**; 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 in “Creating an array with an edit table”.

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(u)”.

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

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

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()** ; **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

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

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)

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`

Tip 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, i , is not a dimension of x , **Sum(x, i)** returns x unreduced, but multiplied by the size (number of elements) of i . The reason is that if x is not indexed by i , it means that it has the same value for all values of i . This is true even if x is an atom with no dimensions:

```
Variable x := 5
Sum(x, Car_type) → 15
```

because `Car_type` has three elements. In this way, if we later decide to change the value for x for each value of i , we can redefine x as an edit table indexed by i . Any expression containing a sum or other reducing function on x will work correctly as the number of dimensions changes.

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

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

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

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

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

Argmax(a, i)

Returns the item of index *i* for which array **a** is the maximum. If **a** has more than one value equal to the maximum, it returns the index of the last one.

Library Array

Example `Argmax(Car_prices, Car_type)` →
Years ▶

	1985	1986	1987	1988
	BMW	BMW	BMW	BMW

Argmin(a, i)

Returns the corresponding value in index *i* for which array **a** is the minimum. If more than one value equals the minimum, returns the index of the last occurrence.

Library Array

Example `Argmin(Car_prices, Car_type)` →
Years ▶

	1985	1986	1987	1988
	VW	VW	VW	VW

CondMin(X : Array[I], cond : Boolean[I] ; i : IndexType)

CondMax(X : Array[I], cond : Boolean[I] ; i : IndexType)

Conditional Min and Max. **CondMin()** returns the smallest, and **CondMax()** returns the largest values along a given index, *i*, that satisfies condition **cond**.

Tip These functions do not support named parameter syntax.

Library Array

Syntax `CondMin(X : Array[i], cond : Boolean[i] ; i : IndexType)`
`CondMax(X : Array[i], cond : Boolean[i] ; i : IndexType)`

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(x)**”).

Argmax() uses `Subindex(a, Max(a, i), i)` to return the index value corresponding to the maximum value in **a**. See “Transforming functions”.

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	

PositionInIndex(a, x, i)

Returns the position in index `i` — that is, a number from 1 to the size of index `i` — of the last element of array `a` equal to `x`; if no element is equal, it returns 0.

When array `a` is multidimensional, the result will be reduced by one dimension, dimension `i`.

Library Array

Examples When the array is one-dimensional.

```
Index I := ['A', 'B', 'C']
Variable A := Array(I, [1, 2, 2])
PositionInIndex(A, 1, I) → 1
PositionInIndex(A, 2, I) → 3
PositionInIndex(A, 5, I) → 0
```

Tip `PositionInIndex()` is the positional equivalent of `SubIndex()`. It is useful when `i` contains duplicate values, in which case `SubIndex()` is insufficient.

More examples and tips When the array is multidimensional:

`PositionInIndex(Car_prices, 14K, Car_type) →`
Years ▶

	1985	1986	1987	1988
	0	0	2	0

Tip Parameter `a` is optional. When omitted, it returns the position of `x` in the index `i`, or 0 if not found. The syntax `@[i=x]` (see returns the same result as `PositionInIndex(, x, i)`:

```
PositionInIndex('B', I) → 2@[I = 'B'] → 2
PositionInIndex('D', I) → 0@[I = 'D'] → 0
```

When the array parameter is omitted:

```
PositionInIndex(, 'Honda', Car_type) → 2
PositionInIndex(, 'VW', Car_type) → 1
```

@: Index Position Operator

The *position* of value **x** in an index **i** is the integer *n* where **x** is the *n*th element of **i**. *n* is a number between 1 and **Size(i)**. The first element of **i** is at position 1; the last element of **i** is at position **Size(i)**. The position operator **@** offers three ways to work with positions:

- **@i** → an array of integers from 1 to **Size(i)** indexed by **i**.
- **@[i=x]** → the position of value **x** in index **i**, or 0 if **x** is not an element of **i**.
- **e[@i=n]** → the *n*th Slice of the value of expression **e** over index **i**.

Examples Car_type:

VW	Honda	BMW
----	-------	-----

```
@[Car_type='Honda'] →
Car_type ▶
```

	VW	Honda	BMW
	1	2	3

```
@Car_type → 2
```

```
Car_type[@Car_type=2] → 'Honda'
```

More examples and tips

Assume **Time**:

0	1	2	3	4
---	---	---	---	---

and **Years** := Time+2007:

2007	2008	2009	2010	2011
------	------	------	------	------

```
@Time →
```

```
Time ▶
```

	0	1	2	3	4
	1	2	3	4	5

```
@[Time=2] → 3
```

```
@Time=3 →
```

```
Time ▶
```

	0	1	2	3	4
	0	0	1	0	0

```
Time[@Time=3] → 2
```

```
(Time+2007)[@Time=3] → 2009
```

```
Years[@Time=3] → 2009
```

Tip You can use the slice variation to re-index an array by another array having the same length but different elements. For example, Suppose **Revenue** is indexed by **Time**, then

the following expression returns the same array indexed by **Years**:

```
Revenue[@Time=@Years]
```

For additional information consult the Analytica wiki:

http://lumina.com/wiki/index.php/Ana:Alphabetical_Function_List

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 “Integrate(r, i)”).

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

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.

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 \mathbf{x} is not indexed by i , `Cumulate(x, i)` operates as if \mathbf{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

Uncumulate (x, i, firstElement)

`Uncumulate(x, i)` returns an array whose first element (along i) is the first element of \mathbf{x} , and each other element is the difference between the corresponding element of \mathbf{x} and the previous element of \mathbf{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 \mathbf{x} and the previous element of \mathbf{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

Cumproduct(x, i)

Returns an array with each element being the product of all of the elements of \mathbf{x} along dimension i up to, and including, the corresponding element of \mathbf{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

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

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 186).

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

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

Selecting, slicing, and subscripting arrays

These constructs and functions select a slice or subarray of an array. The result may be a single cell, or a subarray, with (usually) one less dimension than the array from which it was sliced.

X[I=V]: Subscript construct

The most commonly used method to extract a subarray is the subscript construct:

`X[I = V]`

This returns the subarray of *X* for which index *I* has value *V*. If *V* is not a value of index *I*, it returns NULL, and usually gives a warning.

If *X* does not have *I* as a index, it just returns *X*. The rationale is that array *X* gives the value of *X* for each combination of indexes that it is indexed by. If it is not indexed by *I*, that means *X* has the same value for all values of *I*.

You can apply the subscript construct to an expression:

`(Revenue - Cost)[Time = 2010]`

**Indexing by name
not position**

You may subscript over multiple dimensions, for example:

`X[I=V, J=U]`

You get the same result from:

`X[J=U, I=V]`

You can specify the indexes in any sequence. Unlike most computer languages, with Analytica you identify the dimension you want to subscript over by *naming* each 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 just a matter of how you choose to display the array in a table.

The value **V** can be an array with some index other than **I** of values from the index **I**. For example, **V** might be a subset of **I**. In that case, the result is an array with the index(es) of **V** containing the corresponding elements of **X**. See the description in “Subscript(*u1*, *i*, *u2*)” for an example. The **Subscript(x, i, v)** function is the same as the subscript construct `x[i=v]`, just using different syntax.

X[@I=n]: Slice construct

The slice construct has an @ sign before the index. It is different from the subscript construct in that it refers to the numerical *position* rather than the *value* of index **I**. It returns the **n**th slice of **X** over index **I**:

```
x[@I=n]
```

The number **n** should be an integer between 1 (for the first element of index **I**) and `size(I)` for the last element of **I**. If **n** is not an integer in this range, it returns NULL, and returns a warning (unless warnings have been turned off). Like the subscript construct, it can slice over multiple indexes, for example:

```
x[@I=n, @J=m]
```

You can mix slice and subscript operations in one expression:

```
x[@I=1, J=2, K=3]
```

The slice construct does just the same as the **Slice(X, I, V)** function (see “Slice(*u*, *i*, *n*)”), but with different syntax.

x[time-n]: Preceding time slice

`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 inside the **Dynamic()** function. See “Dynamic(*initial1*, *initial2*..., *initialn*, *expr*)” on page 298.

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 = u2]` 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

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 **n**th 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 **n**th 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'])
```

Tip 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

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 = u2]` is equivalent to `Subscript(u1, i, u2)` when `x` is a variable identifier that evaluates to `u1`.

`Subscript(x, i, u1)` and `x[i=u1]` just two ways to do the same thing.

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`

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 choice menu” on page 123). **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 |
|--|------|------|------|------|


```

Converting between multiD and relational tables

The **MDArrayToTable()** function "flattens" a multi-dimensional array into a two-dimensional relational 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, I)

Transforms a multi-dimensional array, **a**, into a two-dimensional array (i.e., a table) indexed by **i** and **I**. The result contains one row along **i** for each element of **a**. **I** must contain a list of names of the indexes of **a**, followed by one final element. All elements of **I** must be text values. The column corresponding to the final element of **I** contains the cell value. If **I** does not contain all the indexes of **a**, array abstraction will create a set of tables indexed by the dimensions not listed in **I**.

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

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**, where ($n = \text{size}(\text{cols})$). 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 to 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*. *Undefined* 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') →`

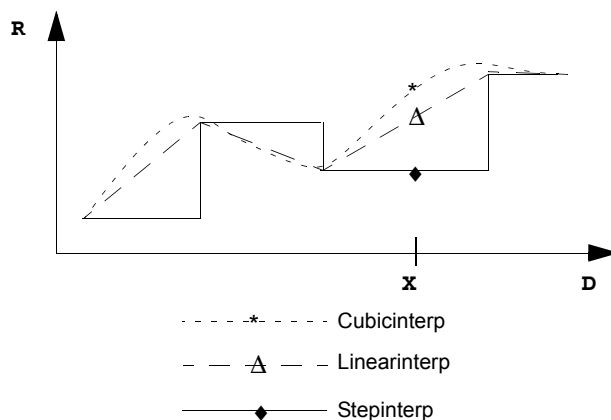
Car_type ▼ , Mpg ►

	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**, it returns the element for which **d** has the largest value.

When an optional parameter, **LeftLookup**, is specified as True, it returns the element or slice of **a** corresponding to the largest value in **d** that is less than or equal to **x**.

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(u1, i, u2)**; 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

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 defined in “*Example data*”, 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

	VW	Honda	BMW
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

ConcatRows(A : Array[I,J] ; I,J,K : Index)

Takes an array, **A** indexed by **I** and **J**, and concatenates each row, flattening the array by one dimension. The result is indexed by **ResultIndex**, which must be an index with **size(I) * size(J)** elements.

Library Concatenation.ana

To use this function, you must add the library to your model.

IndexNames(a)

Returns a list of the names of the indexes of the array **a**.

Library Array

Example `IndexNames(Car_prices) → ['Car_type', 'Years']`

IndexesOf(A : Array)

Returns a list of handles, each one serving as a handle to the indexes of array **A**.

This is similar to **IndexNames(A)**, except that **IndexesOf** returns actual handles, while **IndexNames** returns the index identifiers (as text strings).

It is possible for an array to have more than one local index having identical names. Obviously, this is not recommended, but in the unusual case where this occurs, the index handles returned by **IndexesOf** are unambiguous.

Library Array

Example `IndexesOf(Car_prices) → ['Car_type', 'Years']`

IndexValue(I)

Returns the index value for the given variable or index **I**. Some variables have both an index value and a result value. Examples include a self-indexed array; a variable or index defined as a list of identifiers or list of expressions; and a **Choice** list with a self-domain. **IndexValue(I)** returns the index value of **I**, where **(I)** alone would return its result value.

Library Array Functions

Details The **IndexValue** function, if it weren't built-in, could easily be defined within Analytica 4.0 as:

```
Function IndexValue( I : IndexType ) := I
```

This definition, however, requires Analytica 4.0 or later.

```

Example Index L := [I,J,K,"value"]
          Index rows := 1..Size(A)
          Variable Flat_A := MdArrayToTable( A, rows, IndexValue(L) )

```

Size(u)

Returns the number of array elements of **u**.

Library Array

```

Examples Size(Years) → 4
           Size(Car_prices) → 12
           Size(10) → 1

```

SubTable

The **SubTable** function allows a subset of another edit table to be edited by the user as a different view. To the user, it appears as if he is editing any other edit table; however, the changes are stored in the original edit table. The rows and columns can be transformed to other dimensions in the **SubTable**, with different index element orders, based on **Subset** indexes, and with different number formats.

SubTable(v[i = x]) **SubTable** lets you treat a slice of table **a** as an edit table, where variable **v** is defined as an edit table, probability table, deterrmtable, or another subtable. It lets a model offer alternative editable views of the same input data.

SubTable must appear as the top-level function in an expression. It must contain a slice or subscript operator. For example, in the simplest form:

```
SubTable( A[I=J] )
```

where **J** is an index containing a subset of the elements of **I**, and **A** is a variable containing an edit table, probability table, or deterrmtable. Many other variations are also useful including:

```

SubTable( A[I=x] )
SubTable( A[I1=J1, I2=J2] )
SubTable( A[I=B] )
SubTable( A[@I=C] )

```

where **x** is a scalar, and **b** and **c** are an array indexed by **J**.

ADE and Analytica Web Player

At present, ADE and AWP do not recognize subtables. Thus, in ADE you cannot (yet) obtain a **DefTable** for a variable defined using **SubTable**. This enhancement may be added in the future. In the Analytica Web Player, **SubTable** definitions do not allow editing in table mode.

Matrix functions

Matrix functions perform matrix operations. In Analytica, a **matrix** is defined as a two-dimensional array of numbers.

Dot product of two matrices

The dot product (i.e., matrix multiplication) of **MatrixA** and **MatrixB** is equal to

$$\text{Sum}(\text{MatrixA} * \text{MatrixB}, i)$$

where *i* is the common index.

Example MatrixA:
j ▼, *i* ►

	1	2	3
a	4	1	2
b	2	5	3
c	3	2	7

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

MatrixMultiply(a:Numeric all[aRow,aCol]; aRow, aCol:Index; b:Numeric all[bRow,bCol]; bRow, bCol:Index)

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. **bCol** and **bRow** must have the same length or it flags an error. If **bRow** and **bCol** are the same index, it returns only the diagonal of the result.

Library Matrix

Example Matrices

C index1 ▼, index2 ► x **D** index2 ▼, index3 ►

	1	2
1	1	2
2	1	0

	a	b	c
1	3	0	1
2	0	1	1

MatrixMultiply(C, index1, index2, D, index2, index3) →

index1 ▼ , index3 ►

	1	2	3
1	3	2	3
2	3	0	1

When the inner index is shared by **C** and **D**, the expression `Sum(C*D, index2)` is equivalent to their dot product (see “Dot product of two matrices”).

Tip 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 `MatrixMultiply(A,I,J,A,J,I)` because the result would have to be doubly indexed by **I**.

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

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

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`

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 $\mathbf{v} * \mathbf{c} * \mathbf{v} > 0$, for all vectors \mathbf{v} .)

Cholesky decomposition computes a lower diagonal matrix `L` such that $\mathbf{L} * \mathbf{L}' = \mathbf{c}$, where \mathbf{L}' is the transpose of `L`.

Library `Matrix`

Example `Matrix`

`l ▼ , 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) →`

`l ▼ , 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

EigenDecomp(a:Numeric[i, j]; i, j:Index)

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

$$\mathbf{a} \mathbf{x} = \lambda \mathbf{x}$$

An $N \times N$ matrix will have N (not-necessarily unique) Eigenvalue-Eigenvector pairs. When \mathbf{a} is a symmetric matrix, the Eigenvalues and Eigenvectors are real-valued. Eigen-analysis is widely used in Engineering and statistics.

Tip The matrix \mathbf{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 Covariance 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 >>

<<ref₁>>
stock1 ▼

INTC	0.2845
MOT	0.518
AMD	-0.8067

<<ref₁>>
stock1 ▼

INTC	0.6548
MOT	-0.7196
AMD	-0.2312

<<ref₁>>
stock1 ▼

INTC	-0.7002
MOT	-0.4625
AMD	-0.5439

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 \mathbf{a} , indexed by \mathbf{i} and \mathbf{j} , with $\text{Size}(\mathbf{i}) \geq \text{Size}(\mathbf{j})$, into three matrices, \mathbf{U} , \mathbf{W} , and \mathbf{V} , such that

$$\mathbf{a} = \mathbf{U} \cdot \mathbf{W} \cdot \mathbf{V} \quad (1)$$

where \mathbf{u} and \mathbf{v} are orthogonal matrices and \mathbf{w} is a diagonal matrix. \mathbf{u} is dimensioned by \mathbf{i} and \mathbf{j} , \mathbf{w} by \mathbf{j} and $\mathbf{j2}$, and \mathbf{v} by \mathbf{j} and $\mathbf{j2}$. In Analytica notation:

```
Variable A :=
  Sum(Sum(U*W,J) * Transpose(V,J,J2), J2)
```

The index $\mathbf{j2}$ must be the same size as \mathbf{j} and is used to index the resulting \mathbf{w} and \mathbf{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']`

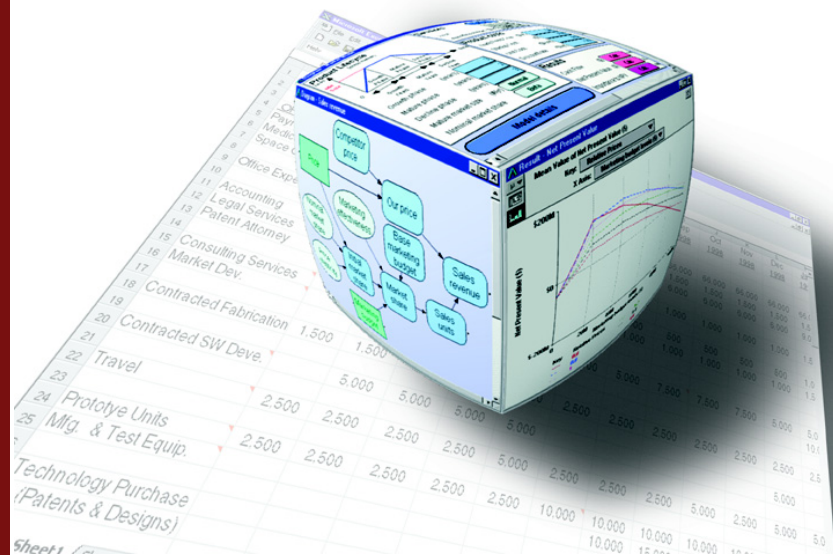
Tip Like most other matrix functions, **SingularValueDecomp()** requires its main parameter to be square, and will not work if indexes **i** and **j** are not the same size.

Chapter 13

Other Functions

This chapter describes a variety of useful functions from built-in and added libraries:

- Text functions that work with text values, to transform, search, split, and join them; see “Text functions”.
- Date functions for working with date numbers; see “Date functions”.
- Advanced math functions; see “Advanced math functions”.
- Built-in financial functions; see “Financial functions”.
- A library of extra financial functions, including functions for valuing options; see “Financial library functions”.
- Advanced probability functions; see “Advanced probability functions”.



Text functions

These functions work with **text values** (sometimes known as *strings*), available in the built-in Text library. See “Text values”.

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 numeric 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 a multi-line text file.

TextLength(t) Returns the number of characters in text **t**.

```
TextLength('supercalifragilisticexpialidocious') → 34
```

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'
```

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. **FindinText()** is case-sensitive. For example:

```
Variable People := 'Amy, Betty, Carla'
FindinText('Amy', People) → 1
FindinText('amy', People) → 0
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.

```
FindinText('i', 'Supercalifragilisticexpialidocious') → 9
FindinText('i', 'Supercalifragilisticexpialidocious', 10) → 14
```

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'
```

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 converts them to text using the number format of 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.

JoinText(a, i, separator, finalSeparator)

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 **separator** is specified, it is inserted as a separator between successive elements, for example:

```
JoinText(A, I, ', ') → 'VW, Honda, BMW'
```

The optional parameter **finalSeparator**, if present, specifies a different separator between the second-to-last and last elements of **a**.

```
JoinText(A, I, '; ', ' and') → 'VW; Honda; and BMW'
```

SplitText(t, separator)

Returns a list of text values formed by splitting the elements of text value **t** at each occurrence of separator **separator**. 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.'
```

Tip With **SplitText()**, **t** must be a single text value, not an array. Otherwise, it might generate an array of arrays of different length. See "Functions expecting atomic parameters" on what to do if you want apply it to an array.

TextLowerCase(t)

Returns the text **t** with all letters as lowercase. For example:

```
TextLowerCase('What does XML mean?')
→ 'what does xml mean?'
```

TextUpperCase(t)

Returns the text **t** with all letters as uppercase. For example:

```
TextUpperCase('What does XML mean?')
→ 'WHAT DOES XML MEAN?'
```

TextSentenceCase(Text ; preserveUC : optional)

Returns the text **t** with the first character (if a letter) as uppercase, and any other letters as lowercase. For example:

```
TextSentenceCase('mary ann FRED Maylene')
→ 'Mary ann fred maylene'
TextSentenceCase(SplitText('mary ann FRED Maylene', ' '))
→ ['Mary', 'Ann', 'Fred', 'Maylene']
```

```
TextSentenceCase('they are Fred and Maylene', true )
→ 'They are Fred and Maylene'
```

Date functions

These functions work with *date numbers* — that is, the number of days since the *date origin*, usually Jan 1, 1904. See “Date numbers and the date origin”. Date numbers display as dates if you select a date number format. **MakeDate()** generates a date number from the year, month, and day. **DatePart** extracts the year, month, or day from a date number. **DateAdd()** adds a number of days, weeks, months, or years to a date. And **Today()** returns today’s date.

MakeDate(year, month, day)

Gives the date value for the date with given **year**, **month**, and **day**. If omitted, **month** and **day** default to 1. Parameters must be positive integers.

Examples `MakeDate(2007, 5, 15) → 15-May-2007`
 `MakeDate(2000) → 1-Jan-2000`

Library Special Functions

DatePart(date, part)

Given a date value **date**, it returns the year, month, or day as a number, according to the value of **part**, which must be an uppercase character:

'Y' gives the four digit year as a number, such as 2006.

'M' gives the month as a number between 1 and 12.

'D' gives the day as number between 1 and 31.

'W' gives the day of the week as a number from 1 (Sunday) to 7 (Saturday).

Other options for **part** are: 'MMM' → 'Jan', 'MMMM' → 'January', 'ddd' → '1st', 'dddd' → 'first', 'Dddd' → 'First', 'www' → 'Mon', 'www' → 'Monday', 'q' → 1 to 4 for number of quarter of the year.

Examples `DatePart(MakeDate(2006, 2, 28), 'D') → 28`

This makes a sequence of all weekdays between **Date1** and **Date2**:

```
Index J:= Date1 .. Date2;
Subset(DatePart(J, "W")>=2 AND DatePart(J, "W")<=6 )
```

Library Special Functions

DateAdd(date, n, unit)

Given a date value **date**, it returns a date value offset by **n** years, months, days, or weekdays according to whether **unit** is 'Y', 'M', 'D', or 'WD'. If **n** is negative, it subtracts units from the date.

Examples **DateAdd** is especially useful for generating a sequence of dates, e.g., weeks, months, or quarters, for a time index:

```
DateAdd(MakeDate(2006,1,1), 0..12, "M")
→ ["1 Jan 2006", "1 Feb 2006", "1 Mar 2006", ... "1 Jan 2007"]
```

If an offset would appear to go past the end of a month, it returns the last day of the month:

```
DateAdd( MakeDate(2004, 2, 29), 1, 'Y' ) → 2005-Feb-28
DateAdd( MakeDate(2006, 10, 31), 1, 'M' ) → 2006-Nov-30
```

Since the dates 2005-Feb-29 and 2006-Nov-31 don't exist, it gives the last day of the preceding month.

Adding a day offset, `DateAdd(date, n, "D")`, is equivalent to `date+n`.

`DateAdd(date, n, "WD")` adds the specified number of weekdays to the first weekday equal to or falling after `date`.

Library: Special Functions

Today()

Returns the date number for the day on which the function is evaluated. Thus, it gives a different value when the same model is evaluated on a different day.

Library Special Functions

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". There are additional advanced math functions covered in "Importance weighting".

Arccos(x), Arcsin(x), Arctan2(y, x) The inverse trigonometric functions. For each the parameter x is between 0 and 1, and the result is in degrees. **Arccos** returns a result between 0 and 180 degrees:

```
Arccos(1) → 0
Arccos(Cos(45)) → 45
```

Arcsin returns a result between -90 and 90 degrees:

```
Arcsin(1) → 90
Arcsin(Sin(45)) → 45
```

Arctan2 gives the arctangent of y/x without losing 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):

```
Arctan2(-1,1) → -45
Arctan2(0,-1) → 180
Arctan2(0,0) → 0
```

Cosh(x), Sinh(x), Tanh(x) The hyperbolic cosine, sine, and tangent of x , x assumed to be in degrees.

```
Cosh(0) → 1
Sinh(0) → 0
Tanh(INF) → 1
```

Lgamma(x) Returns the Log Gamma function of **x**. Without numeric overflow, this function is equivalent to $\ln(\text{GammaFn}(x))$. Because the gamma function grows so rapidly, it is often much more convenient to use **LGamma()** to avoid numeric overflow.

LGamma(10) → 12.8

Financial functions

These functions can be accessed under the **Definition** menu **Financial** command, or in the **Object Finder** dialog box, Financial library. The function names and parameters match those in Microsoft Excel, where they are equivalent. Of course, the Analytica versions support array abstraction, which makes them more flexible.

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 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 <i>per period</i> . 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:

$$\sum_{n=sp}^{ep} \text{Ipmt}(\text{rate}, n, n\text{Per}, P\text{v}, 0, \text{Type})$$

Example Interest payments during the first year on a \$100,000 loan at 8% is:

$$\text{CumIPmt}(8\%/12, 360, 100\text{K}, 1, 12) \rightarrow -7,969.81$$

The result is negative since these are payments.

Cumprinc(rate, nPer, pv, startPeriod, endPeriod, 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:

$$\sum_{n=sp}^{ep} \text{PPmt}(\text{Rate}, n, N\text{per}, P\text{v}, 0, \text{Type})$$

Example The total principal paid during the first year on a \$100,000 loan at 8% is:

$$\text{CumPrinc}(8\%/12, 360, 100\text{K}, 1, 12) \rightarrow -835.36$$

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:

$$\text{Fv}(0.5\%, 12, 50, -1000) \rightarrow \$444.90$$

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):

$$-\text{Fv}(1\%, 15*12, -550, 50000) \rightarrow \$25,020.99$$

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:

$$\text{Fv}(0.5\%, [12, 24, 36], -100, 0, \text{True}) \rightarrow$$

[\$1239.72, \$2555.91, \$3953.28]

Ipmnt(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 an 8%/12 monthly interest rate 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 (IRR) 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:

$$\sum_{j=1}^n \frac{Values[I=j]}{(1 + DiscountRate)^j}$$

Tip 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 this cash flow is:

$$\text{Npv}(5\%, \text{Earnings}, \text{Time}) \rightarrow \$865,947.76$$

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):

$$-\text{Pmt}(8\%/12, 30*12, 100\text{K}) \rightarrow \$733.76$$

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):

$$-\text{PPmt}(8\%/12, 24, 12*30, 100\text{K}) \rightarrow \$78.18$$

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):

$$-\text{Pv}(6\%/12, 10*12, 100) \rightarrow \$9,007.35$$

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%`

Tip `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`

Financial library functions

The following functions are not built-in to Analytica, but are located in the Financial library that comes with Analytica.

Calloption (S, X, T, r, theta)

This function calculates the value of a call option using the Black-Scholes formula. For further information on the Black-Scholes model for option pricing see *Basic Black-Scholes: Option Pricing and Trading* by Timothy Falcon Crack.

- Parameters**
- **S** = price of security now
 - **X** = exercise price
 - **T** = time in years to exercise
 - **r** = risk-free interest rate
 - **theta** = volatility of security

Library Financial (add-in library)

Example `Calloption(50,50,0.25,0.05,0.3)` → 3.292

Syntax `Calloption (S,X,T,r,theta:Numeric)`

Putoption (S,X,T,r,theta)

This function calculates the value of a put option using the Black-Scholes formula. For further information on the Black-Scholes model for option pricing see *Basic Black-Scholes: Option Pricing and Trading* by Timothy Falcon Crack.

- Parameters**
- **S** = price of security now
 - **X** = exercise price
 - **T** = time in years to exercise
 - **r** = risk-free interest rate
 - **theta** = volatility of security

Library Financial (add-in library)

Example `Putoption(50,50,0.25,0.05,0.3)` → 2.67

Syntax `Putoption (S,X,T,r,theta:Numeric)`

Capm (Rf,Rm,Beta)

CAPM calculates the expected stock return under the Capital Asset Pricing Model. For further information on the Capital Asset Pricing Model see Black, F., Jensen, M., and Scholes, M. "The Capital Asset Pricing Model: Some Empirical Tests", in M. Jensen ed., *Studies in the Theory of Capital Markets*. (1972).

- Parameters**
- **Rf** = risk free rate
 - **Rm** = market return

- **Beta** = beta of individual stock. Beta is the relative marginal contribution of the stock to the market return, defined as the ratio of the covariance between the stock return and market return, to the variance in the market return.

Library Financial (add-in library)

Example `Capm(8%,12%,1.5) → 0.14`

Syntax `Capm(Rf,Rm,Beta:Numeric)`

CostCapme (rOpp,rD,Tc,L)

This function calculates Miles and Ezzell's (M/E) formula for adjusting the weighted average cost of capital for financial leverage. The M/E formula works when the firm adjusts its future borrowing to keep debt proportions constant.

- Parameters**
- **rOpp** = opportunity cost of capital
 - **rD** = expected return on debt
 - **Tc** = net tax saving per dollar of interest paid. This is difficult to pin down in practice and is usually taken as the corporate tax rate.
 - **L** = debt-to-value ratio

Library Financial (add-in library)

Example `CostCapme(14%,8%,35%,0.5) → 0.1252`

Syntax `CostCapme(rOpp,rD,Tc,L:Numeric)`

CostCapmm (rAllEq, Tc, L)

This function calculates Modigliani and Miller's (M/M) formula for adjusting the weighted average cost of capital for financial leverage. The M/M formula works for any project that is expected to

1. Generate a level, perpetual cash flow.
2. Support fixed permanent debt.

- Parameters**
- **rAllEq** = cost of capital under all-equity financing
 - **Tc** = net tax saving per dollar of interest paid. This is difficult to pin down in practice and is usually taken as the corporate tax rate.
 - **L** = debt-to-value ratio

Library Financial (add-in library)

Example `CostCapmm(20%,35%,0.4) → 0.172`

Syntax `CostCapmm(rAllEq, Tc, L:Numeric)`

Implied_volatility_c (S,X,T,r,p)

This function calculates the implied volatility of a call option, based on using the Black-Scholes formula for options

- Parameters**
- **S** = price of security now

- **X** = exercise price
- **T** = time in years to exercise
- **r** = risk-free interest rate
- **p** = option price

Library Financial (add-in library)

Example `Implied_volatility_c(50, 35, 4, 6%, 15)` → 3.052e-005

Syntax `Implied_volatility_c (S,X,T,r,p : atomic numeric)`

Implied_volatility_p (S,X,T,r,p)

This function calculates the implied volatility of a put option, based on using the Black-Scholes formula for options.

- Parameters**
- **S** = price of security now
 - **X** = exercise price
 - **T** = time in years to exercise
 - **r** = risk-free interest rate
 - **p** = option price

Library Financial (add-in library)

Example `Implied_volatility_p(50, 35, 4, 6%, 15)` → 9.416e-001

Syntax `Implied_volatility_p (S,X,T,r,p : atomic numeric)`

Pvperp (C,rate)

Pvperp() calculates the present value of a perpetuity (a bond that pays a constant amount in perpetuity).

- Parameters**
- **C** = constant payment amount
 - **rate** = interest rate per period

Library Financial (add-in library)

Example `Pvperp(200, 8%)` → 2500

Syntax `Pvperp(C,rate:Numeric)`

Pvgperp (C1, rate, growth)

Pvgperp() calculates the present value of a *growing* perpetuity (a bond that pays an amount growing at a constant rate in perpetuity).

- Parameters**
- **C1** = payment amount in year 1
 - **rate** = interest rate per period
 - **growth** = growth rate per period

Library Financial (add-in library)

Example `Pvgperp(200, 8%, 6%)` → 10K

Syntax `Pvgperp(C1, rate, growth : Numeric)`

Wacc (Debt,Equity,rD,rE,Tc)

Wacc() calculates the after-tax weighted average cost of capital, based on the expected return on a portfolio of all the firm's securities. Used as a hurdle rate for capital investment.

- Parameters**
- **Debt** = market value of debt
 - **Equity** = market value of equity
 - **rD** = expected return on debt
 - **rE** = expected return on equity
 - **Tc** = corporate tax rate

Library Financial (add-in library)

Example `Wacc(1M, 3M, 8%, 16%, 35%)` → 0.133

Syntax `Wacc(Debt,Equity,rD,rE,Tc : Numeric)`

Advanced probability functions

The following functions are not actual probability distributions, but they are useful for various probabilistic analyses, including building other probability distributions. You can find them in the **Advanced math** library from the **Definition** menu.

BetaFn(a, b) The beta function, defined as:

$$BetaFn(a, b) = \int_0^1 x^{a-1} (1-x)^{b-1} dx$$

Betal(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:

$$F(x, n_1, n_2) = BetaI(n_2/(n_1x + n_2))$$

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:

$$p = Pr[x \leq X]$$

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

That is, 68.3% of the area under a normal distribution is contained within one standard deviation of the mean.

CumNormalInv(p, m, s) The inverse cumulative probability function for a normal distribution with mean **m** and standard deviation **s**. Returns the value **x** where:

$$p = Pr[x \leq X]$$

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.

Gammal(x, a, b) Returns the incomplete gamma function, defined as:

$$\text{GammaI}(x, a, b) = \frac{1}{\Gamma(a)} \int_0^{x/b} e^{-t} t^{b-1} dt$$

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):

$$P(x < k) = \text{GammaI}(k, a)$$

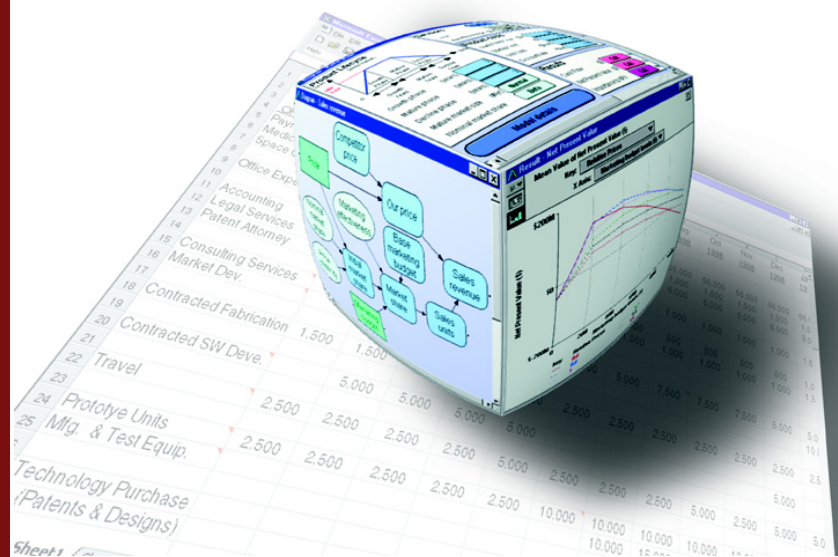
Gammalnv(y, a, b) The inverse of the incomplete gamma function. Returns the value **X** such that **Gammal(X, a, b) = y**. **b** is optional and defaults to 1.

Chapter 14

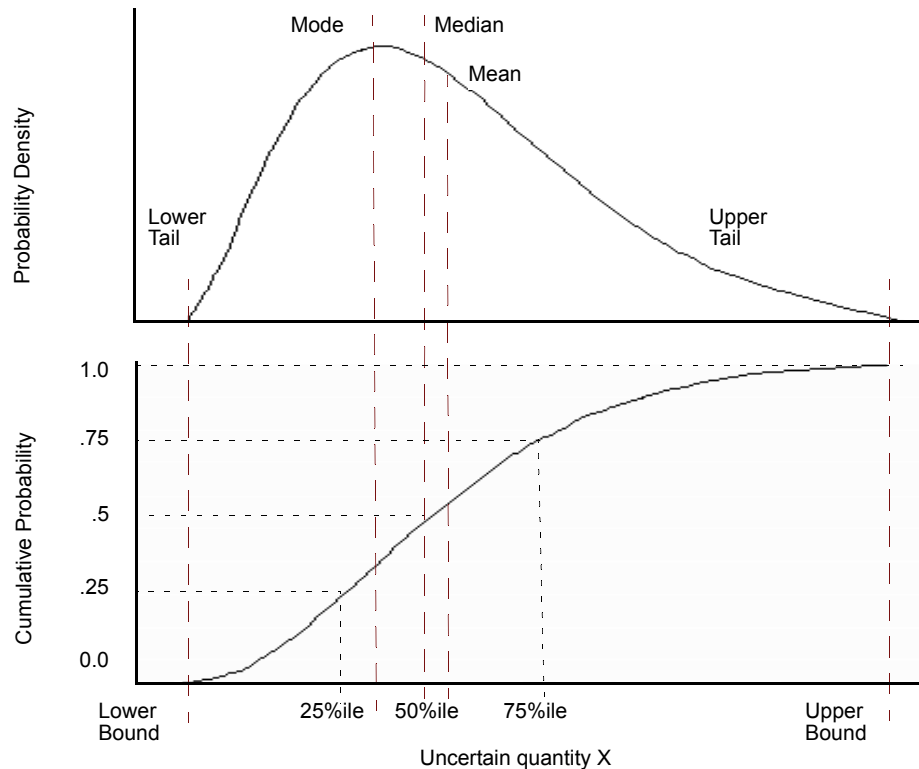
Expressing Uncertainty

This chapter shows you how to:

- Choose a distribution
- Define a variable as a distribution
- Use Analytica's built-in probability distributions



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 “Glossary” on page 421 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.

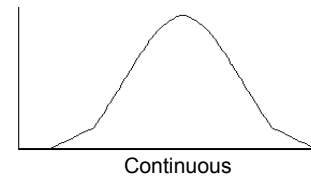
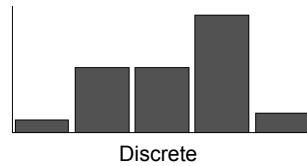
Answer the following questions about the uncertain quantity to select the most appropriate kind of distribution:

- 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.

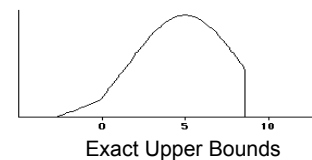
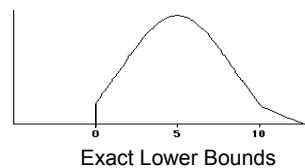
A **continuous** quantity can be represented by a real number, and has infinitely many possible values between any two values in its domain. Examples are the quantity of an air pollutant released during a given period of time, the distance in miles of a residence from a source of air pollution, and the 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.

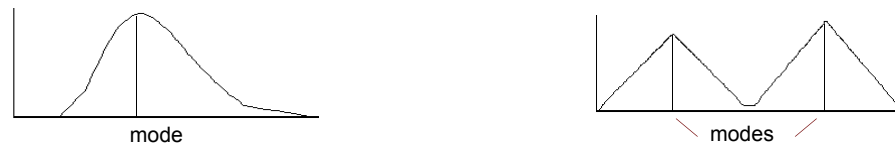
Many standard continuous probability distributions, such as the normal distribution, are unbounded. In other words, there is some probability that a normally distributed quantity

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.

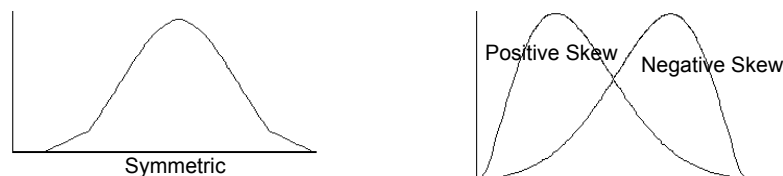


Important questions to ask about a distribution are how many modes it has, and approximately where it, or they, are? Most distributions have a single mode, but some have several and are known as multimodal distributions.

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.

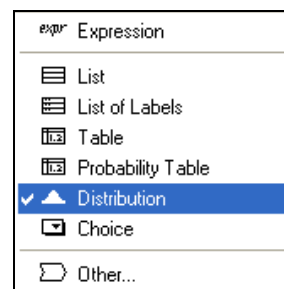
It is generally much faster to assess a standard distribution than a full custom distribution, because standard distributions have fewer parameters, typically from two to four. You should usually start by assigning a simple standard distribution to each uncertain quantity using a quick judgment based 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

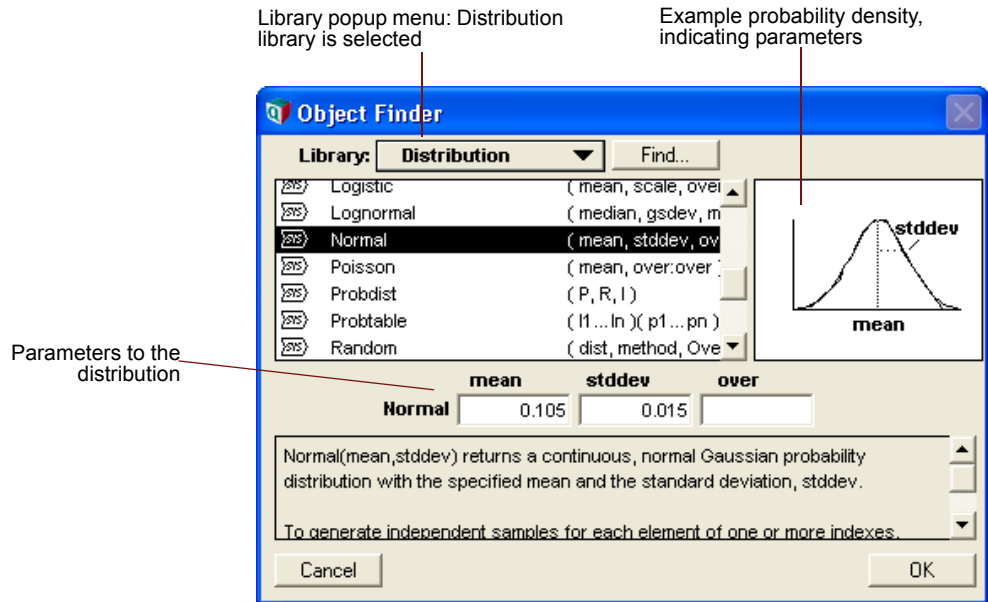
To define a variable as an Analytica probability distribution, first select the variable and open either the variable's **Object** window or the **Attribute** panel of the diagram (see "The Attribute panel" on page 23) with **Definition** selected from the **Attribute** popup menu (see "Creating or editing a definition" on page 116).

To define the distribution:

1. Click the **Expression** popup menu above the definition field and select **Distribution**.

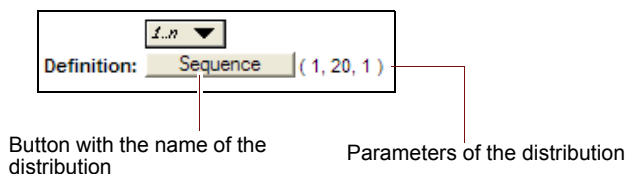


The **Object Finder** opens, showing the Distribution library.



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 **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 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:

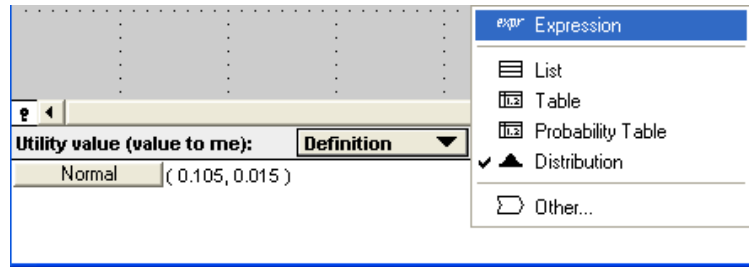
1. Set the cursor in the definition field and type in the distribution name and parameters, for example:

`Normal(.105,0.015)`

2. Press **Alt-Enter** or click the button.

You can also paste a distribution from the Distribution library in the **Definition** menu (see “Using a library” on page 346).

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 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 views” on page 32). 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

Tip 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 232).

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 232).

Tip If you try to apply an array reducing function (see “Array-reducing functions”) 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 280).

Example:

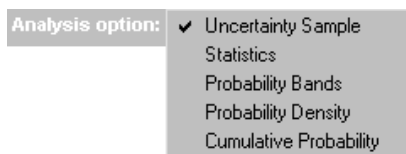
Max(Sample(X),Run) → 0.8892

Uncertainty Setup dialog box

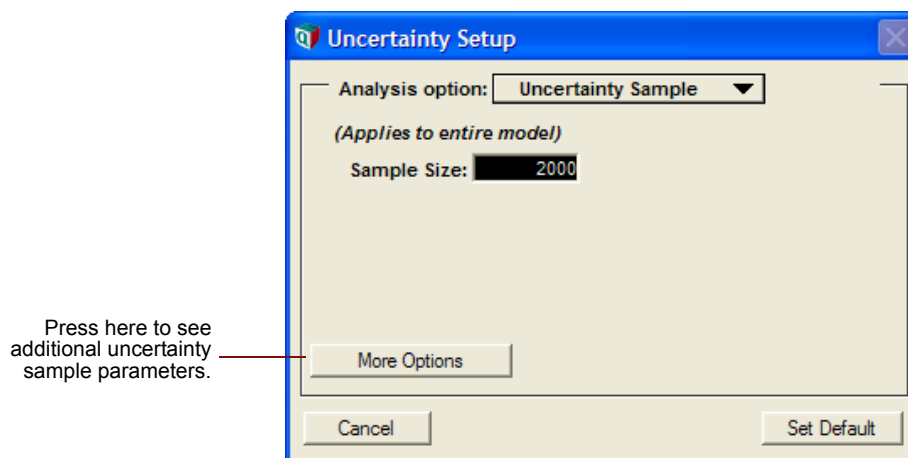
Use the **Uncertainty Setup** dialog box to inspect and change the sample size, sampling method, statistics, probability bands, and samples per plot point for probability distributions. All settings are saved with your model.

To open the **Uncertainty Setup** dialog box, select **Uncertainty Options** from the **Result** menu or *Control+u*. To set values for a specific variable, select the variable before opening the dialog box.

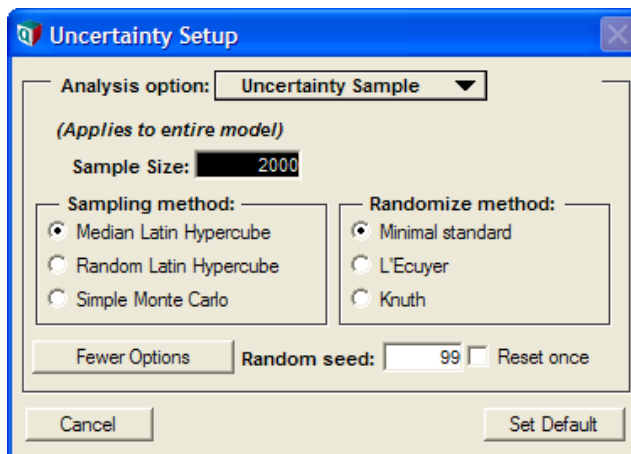
The five options for viewing and changing information in the **Uncertainty Setup** dialog box can be accessed using the **Analysis option** popup menu.



Uncertainty sample To change the sample size or sampling method for the model, select the **Uncertainty Sample** option from the **Analysis option** popup menu.



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” 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, n , 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 $i=1,2,\dots,m$, 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 ,

$$X_i \text{ where } P(x \leq X_i) = u_i \text{ for } i=1,2,\dots,m.$$

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”.

- **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

$$X_i \text{ where } P(x \leq X_i) = (i-0.5)/m, \text{ for } i=1,2,\dots,m.$$

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:

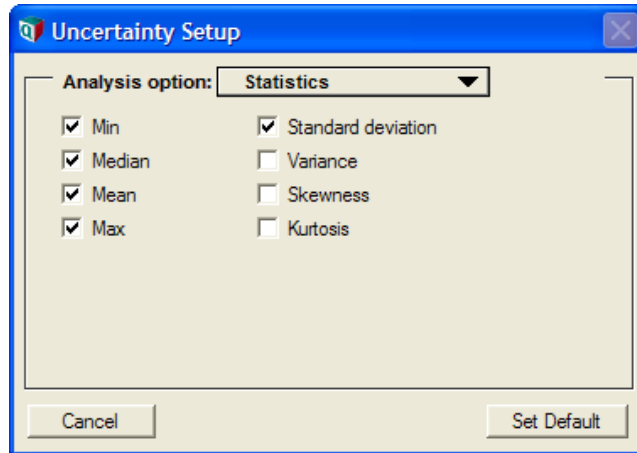
- A model is opened
- The value in this field is changed
- The **Reset once** box is checked, and the **Uncertainty Setup** dialog box is closed by clicking the **Accept** or **Set Default** button.

Reset once

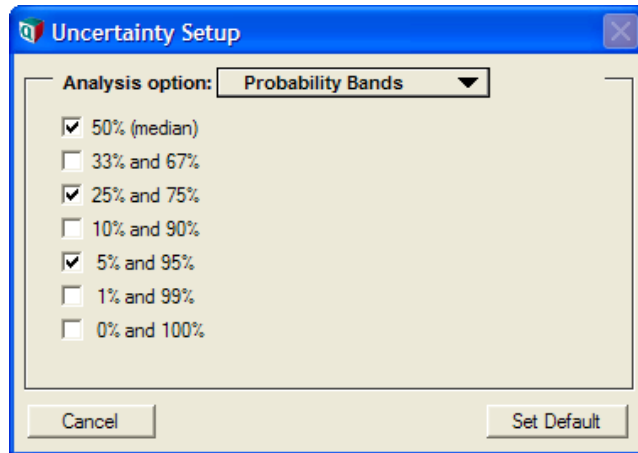
Check the **Reset once** box to produce the exact same series of random numbers.

Statistics option

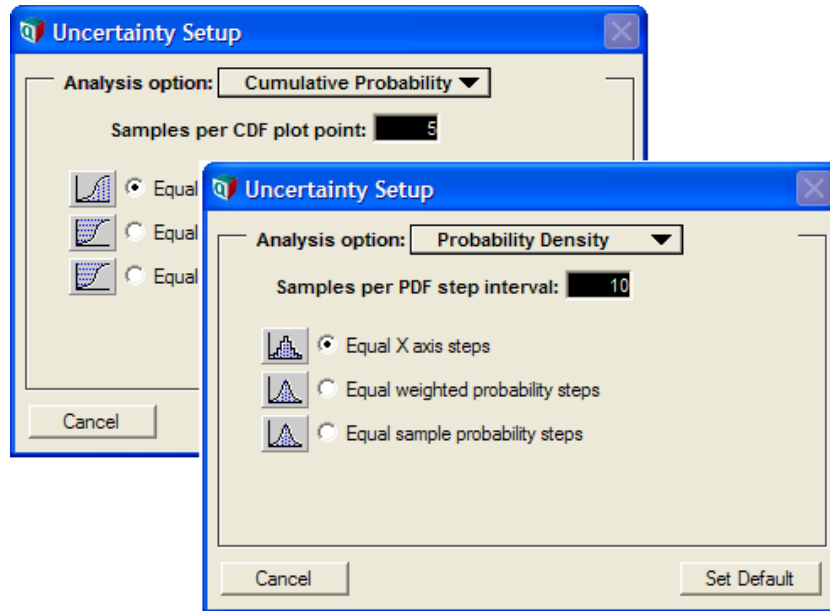
To change the statistics reported when you select **Statistics** as the uncertainty view for a result, select the **Statistics** option from the **Analysis option** popup menu.



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 “Selecting the Sample Size” on page 398).

Equal probability steps

With this option, Analytica uses the sample to estimate a set of $m+1$ fractiles (quantiles), x_p , at equal probability intervals, where $p=0, \alpha, 2\alpha, \dots, 1$, and $\alpha = 1/m$. 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, α . The density between two successive fractiles is plotted at the mid point (on the horizontal axis) of the two fractiles.

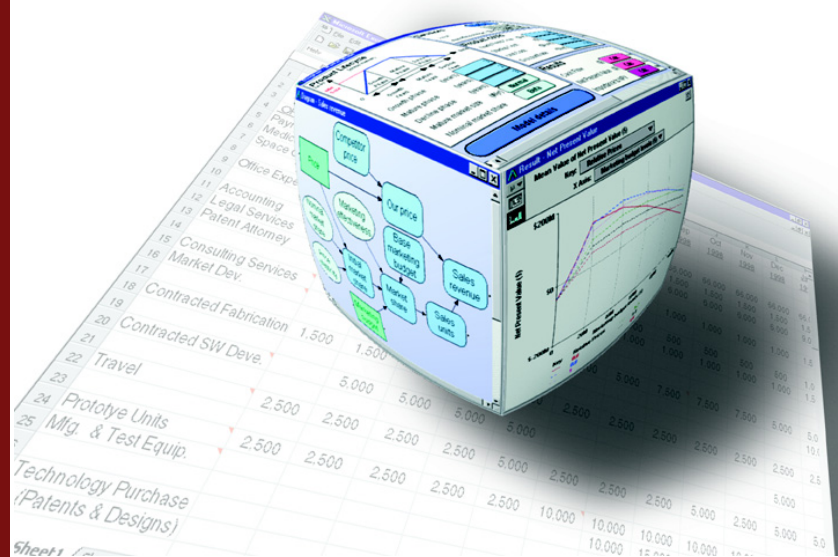
Equal X axis steps

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 bar is estimated as the fraction of the sample values that fall within that x interval.

Chapter 15

Probability Distributions

This chapter describes how to define uncertain quantities using probability distributions, discrete or continuous. You can use standard *parametric* distributions, such as a Normal, Uniform, Bernoulli, or binomial, or custom distributions, where you specify points in tables or arrays. You can also create multivariate distributions over an array of uncertain quantities.



Probability distributions

The built-in Distribution library (available from the **Definition** menu) offers a wide range of distributions for *discrete* and *continuous* variables. (See “Is the quantity discrete or continuous?” and “Glossary” on page 421 for an explanation of this distinction.) Some are standard or *parametric* distributions with just a few parameters, such as **Normal** and **Uniform**, which are continuous, and **Bernoulli** and **Binomial**, which are discrete. Others are *custom* distributions, such as **CumDist**, which lets you specify an array of points on a cumulative probability distribution, and **Probtble** (page 248), which lets you edit a table of probabilities for a discrete variable conditional on other discrete variables.

There are a variety of ways to create arrays of uncertain quantities, or multivariate distributions (see “Multivariate distributions”). You may set parameters to array values, specify an index to the optional **Over** parameter, or use functions from the **Multivariate** library.

Parametric Discrete

- **Bernouli()**
- **Binomial()**
- **Poisson()**
- **Geometric()**
- **Hypergeometric()**
- **Uniform()**

Custom Discrete

- **Probtble()**
- **Determtable()**
- **Chancedist()**

Special Probabilistic

- **Certain()**
- **Shuffle()** page 263
- **Truncate()**
- **Random()**

Parametric Continuous

- **Uniform()**
- **Triangular()**
- **Normal()**
- **Lognormal()**
- **Beta()**
- **Exponential()**
- **Gamma()**
- **Logistic()**
- **StudentT()**
- **Weibull()**
- **Chisquared()**

Custom Continuous

- **Cumdist()**
- **Probdist()**

Multivariate

- **Normal_correl()**
- **Correlate_with()**
- **Dist_reshape()**
- **Correlate_dists()**
- **Gaussian()**
- **Multinormal()**
- **BiNormal()**
- **Dirichlet()**
- **Multinomial()**
- **UniformSpherical()**
- **MultiUniform()**
- **Normal_serial_correl()**
- **Dist_serial_correl()**
- **Normal_additive_gro()**
- **Dist_additive_growth()**
- **Normal_compound_gro()**
- **Dist_compound_growth()**

Parametric discrete distributions

Bernoulli (p)

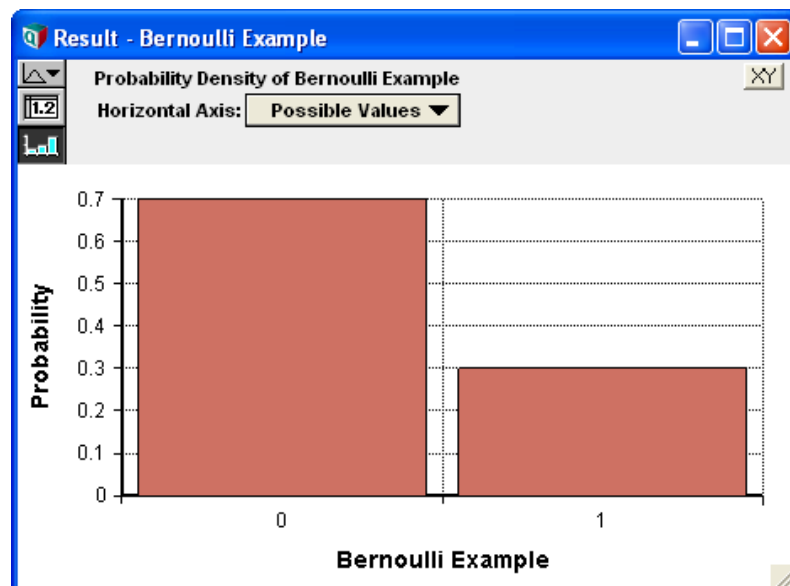
Defines a discrete probability distribution with probability p of result 1 and probability $(1 - p)$ of result 0. It generates a sample containing 0s and 1s, with the proportion of 1s is approximately p . p is a probability between 0 and 1, inclusive, or an array of such probabilities. The Bernoulli distribution is equivalent to:

```
If Uniform(0, 1) < P Then 1 Else 0
```

Library Distribution

Example The domain, **List of numbers**, is [0, 1].

```
Bernoulli_ex: Bernoulli (0.3) →
```



Binomial(n, p)

An event that can be true or false in each trial, such as a coin coming down heads or tails on each toss, with probability p 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 equivalent, if less efficient, way to define a Binomial distribution function would be:

```
Function Binomial2(n, p)
Parameters: (n: Atom; p)
Definition: Index i := 1..n;
Sum(FOR J := I DO Bernoulli(p), i)
```

The parameter **n** is qualified as an **Atom** to ensure that the sequence `1..n` is a valid one-dimensional index value. It allows `Binomial` to array abstract if its parameters **n** or **p** are arrays.

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 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.

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)

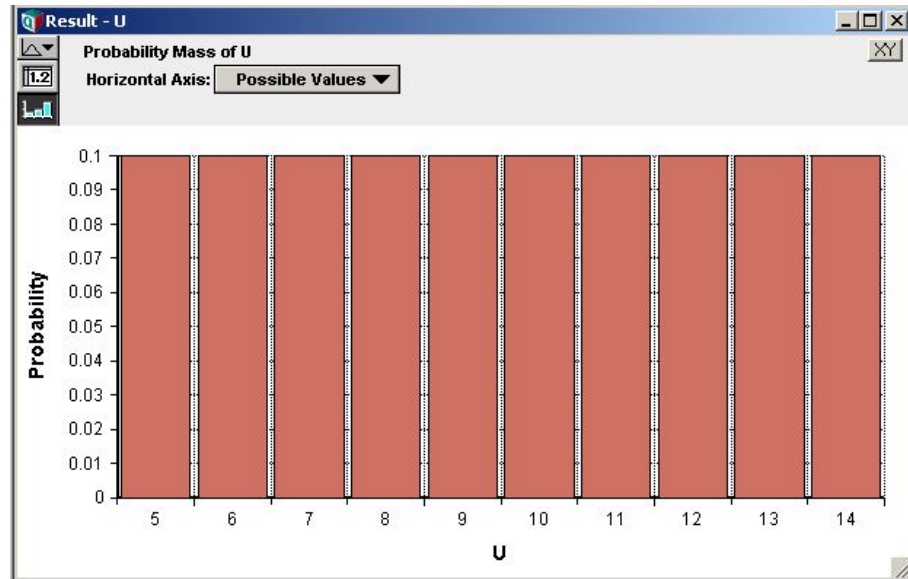
The hypergeometric distribution describes the number of times an event occurs in a fixed number of trials without replacement, e.g., the number of red balls in a sample of **s** balls drawn without replacement from an urn containing **n** balls of which **m** are red. Thus, the parameters are:

- 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.

Uniform(min, max, Integer:True)

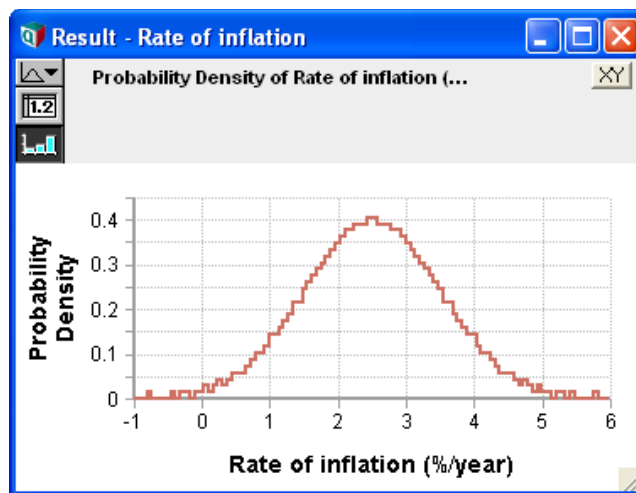
The **Uniform** distribution with the optional integer parameter set to `True` returns discrete distribution over the integers with all integers between and including **min** and **max** having equal probability.

`Uniform(5, 14, Integer:True)` →



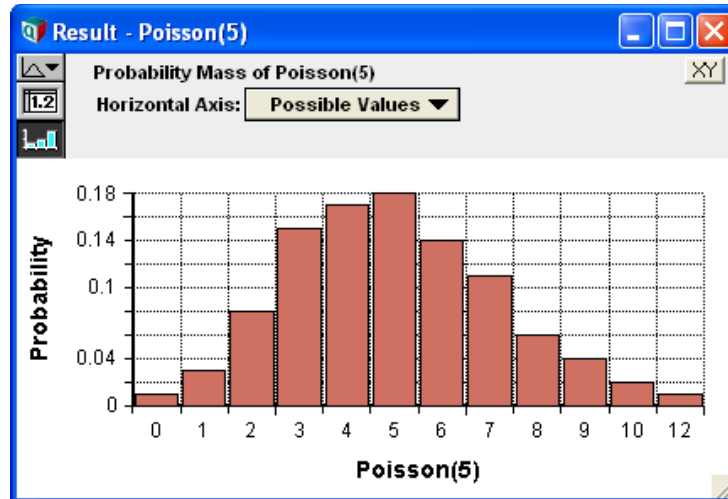
Probability density and mass graphs

When you select the **Probability density** as the Uncertainty view (see “Uncertainty views” on page 32) for a *continuous* variable, it graphs the distribution as a **Probability Density function**. The height of the density shows the relative likelihood the variable will have that value:

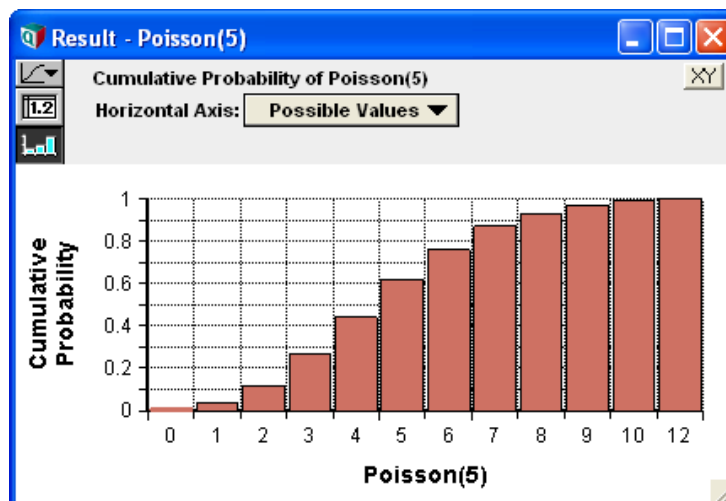


Technically, the probability density of variable x , means the probability per unit increment of x . The units of probability density are the reciprocal of the units of x — if the units of x are dollars, the units of probability density are probability per dollar increment

If you select **Probability density** as the Uncertainty view for a *discrete* variable, it actually graphs the **Probability Mass** function — using a bar graph style to display the *probability* of each discrete value as the height of each bar:



Similarly, if you choose the **cumulative probability** Uncertainty view for a *discrete* variable, it actually displays the **cumulative probability mass** distribution as a bar graph. Each bar shows the cumulative probability that *x* will have that value or any lower value:



Is a distribution discrete or continuous?

Almost always, Analytica can figure out whether a variable is discrete or continuous, and so choose the probability density or probability mass view as appropriate — so you don't need to worry about it. If the values are text, it knows it must be discrete. If the numbers are integers, such as generated by Bernoulli, Poisson, binomial, and other discrete parametric distributions, it also assumes it is discrete.

Infrequently, a discrete distribution may contain numbers that are not integers, which it may not recognize as discrete, for example,

```
Chance Indiscrete := Poisson(4)*0.5
```

In this case, you can make sure it does what you want by specifying the domain attribute of the variable as discrete (or continuous). The next section on the domain attribute explains how.

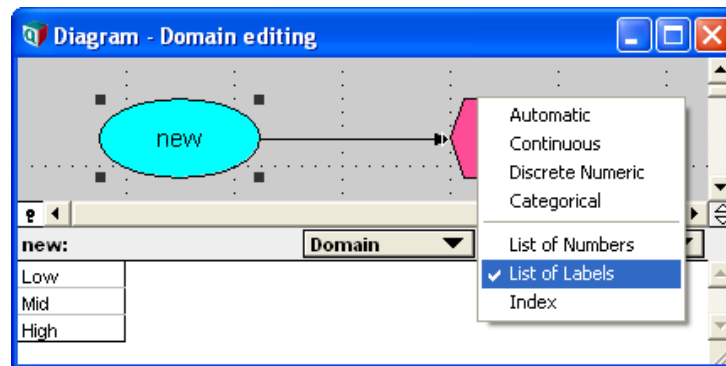
The domain attribute and discrete variables

The **domain** attribute specifies the set of possible values for a variable. You rarely need to view or change a domain attribute explicitly. The most common reason to set the domain is for a variable defined as a custom discrete distribution, especially **ProbTable**. You can do this by editing it directly as an index in the probtable view (see “Probtable(): Probability Tables”), so you can usually ignore the information below. The rare case you will need it is to specify a distribution as discrete, when Analytica would not otherwise figure it out — because it has non-integer numerical value.

By default, the domain type is **Automatic**, meaning Analytica figures it out when it needs to. Usually, this is obvious (see previous section). For a discrete quantity, the domain may be a **list of numbers** or a **list of labels**. If the domain is **continuous**, it means that any number is valid.

Editing the domain You can view and edit the domain like any other attribute of a variable, in the **Attribute** panel:

1. Select the variable.
2. Open the **Attribute** panel, and select **Domain** from the **Attribute** menu.
3. Select the domain type from the popup menu:



The domain type

Automatic: The default, meaning Analytica should figure it out.

Continuous: Any number. All other types are discrete.

Discrete Numeric and **Categorical:** Discrete but its values are unspecified.

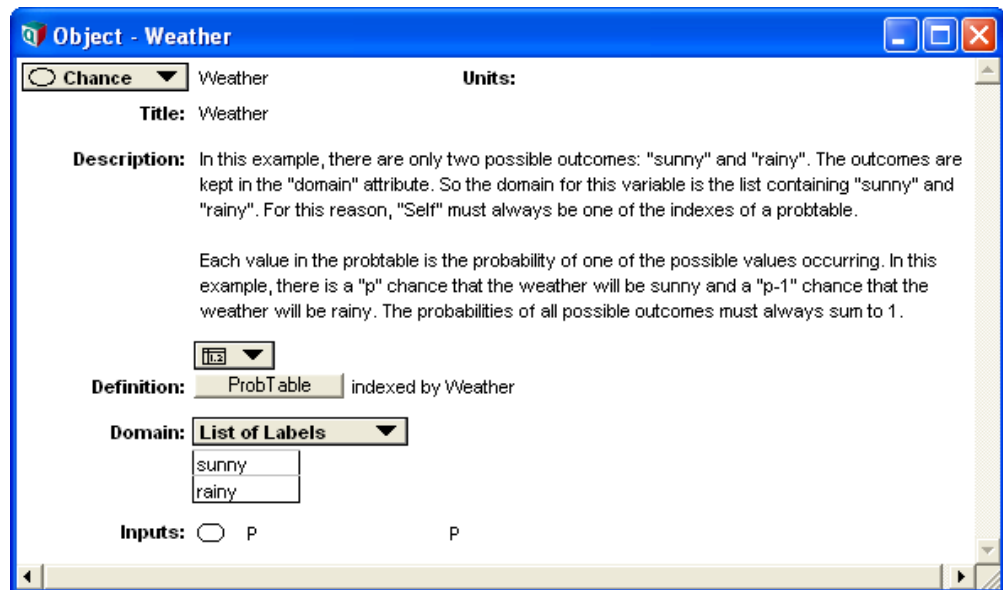
List of Numbers: You specify a list of numbers.

List of Labels: You specify a list of label (text) values, as illustrated.

Index: You enter the name of an index variable, to use its values as the domain, or another variable to copy its domain values.

4. If you choose **List of Numbers** or **List of Labels**, you enter the list values in the usual way (see).

Domain in the Object window You can also view and edit the domain attribute in the **Object** window if you set it to do so in the **Attributes** dialog (see “Managing attributes”):



Tip The domain of a discrete variable should include all its possible values. If not, its probability mass function may sum to less than 1.

Custom discrete probabilities

These functions let you specify a discrete probability distribution using a custom set of values, text (label) values, or numbers.

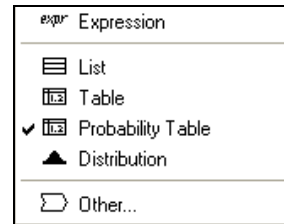
Probtable(): Probability Tables

To describe a probability distribution on a discrete variable whose domain is a list of numbers or list of labels, you use special kind of edit table called a **probability table** (or **probtable**) (see “Arrays and Indexes” on page 151).

Create a probability table

To define a variable using a probability table:

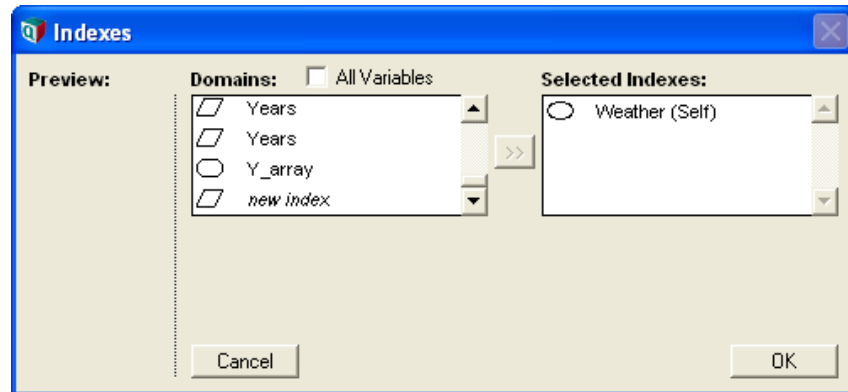
1. Determine the variable’s **domain** — number or labels for its possible values.
2. Select the variable and view its definition attribute in the **Object** window or **Attribute** panel of the **Diagram** window (see “The Attribute panel” on page 23).
3. Press the **Expression** menu above the definition field and select **Probability Table**.



If the variable already has a definition, it confirms that you wish to replace it.

Tip If the definition of a variable is already a probability table, a **ProbTable** button appears in the definition. Click it to open the **Edit Table** window (see “Viewing an array as an edit table” on page 155).

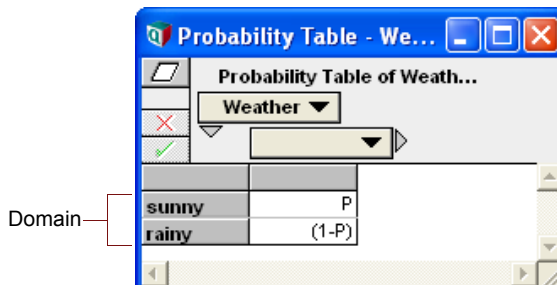
4. The **Indexes** dialog opens to confirm your choices for the indexes of the table. It already includes the selected variable (self) among the selected indexes. Other options are variables with a domain that is a list of numbers or list of labels. Add or remove any other variables that you want to condition this variable on.



Tip `self` is required as an index of a probability table. It refers to the domain (possible values) of this variable.

5. Click the **OK** button. An **Edit Table** window appears.
6. Enter the possible values for the domain in the left column. As in any edit table, press *Enter* or *down-arrow* in the last row to add a row. Select **Insert row** (*Control+i*) or **Delete row** (*Control+k*) from the **Edit** menu. If they are numbers, they must be in increasing order.
7. Enter the probability of each possible outcome in the second column. The probabilities should sum to 1. You may enter literal numbers or expressions.

Example If x is a variable whose value is a probability (between 0 and 1) and the possible weather outcomes are sunny and rainy, you might define a probability table for weather thus:



Expression view of probability table

The **Weather** probability table when viewed as an *expression*, looks like this:

```
Probtable(Self) (P, (1-P))
```

The domain values do not appear in the expression view, and it is not very convenient for defining a probability table. More generally, the *expression* view of a multidimensional probability table looks like this:


```
Probtable(i1, i2, ... in) (p1, p2, p3, ... pm)
```

This example is 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. **m** is the product of the sizes of the indexes **i1, i2, ... in**.

Add a conditioning variable

You may wish to add one or more conditioning variables to a probability table, to create **conditional dependency**. Each discrete conditioning variable adds a dimension to the table. For example, in the **Weather** probability table (see page 247), the probability of rain may depend on the season. So you might have **Season** as a conditioning variable, defined as a list of labels:

```
Variable Season := ['Winter', 'Spring', 'Summer', 'Fall']
```

1. Open the **Edit Table** window by clicking the **ProbTable** button.
2. Click the indexes  button to open the **Indexes** dialog box.
3. Click the **All Variables** check box above the left hand list.
4. Move the desired variable, e.g., **Season**, to add it as an index.
5. Click 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. You need to enter a probability for each cell. The probabilities must sum to one over the domain of the variable (**sunny** and **rainy** in the example), not over the conditioning index(es).

Tip You must have already specified the variables as probability tables, before adding them with the **Indexes** dialog box.

Determtable(): Deterministic conditional table

Determtable() defines the value of a variable as a deterministic (not uncertain) function of one or more discrete variables. It gives a value conditional on the value of one or more discrete variables, often including a probabilistic discrete variable and a discrete decision variable defined as a list. We include it in this section on discrete probability distributions, even though it is not probabilistic, because you will usually use it in con-

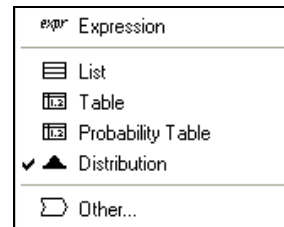
junction with **Protable** and other discrete distributions. It is an editable table, like **Protable**, but with a single (deterministic) value, number, or text, in each cell.

The **Determtable()** function looks like an edit table or a probability table, with an index (dimension) from each discrete variable on which it depends. Unlike **Protable**, it does not need a self index. Its result is probabilistic if any of its conditioning variables are probabilistic.

Creating a determtable

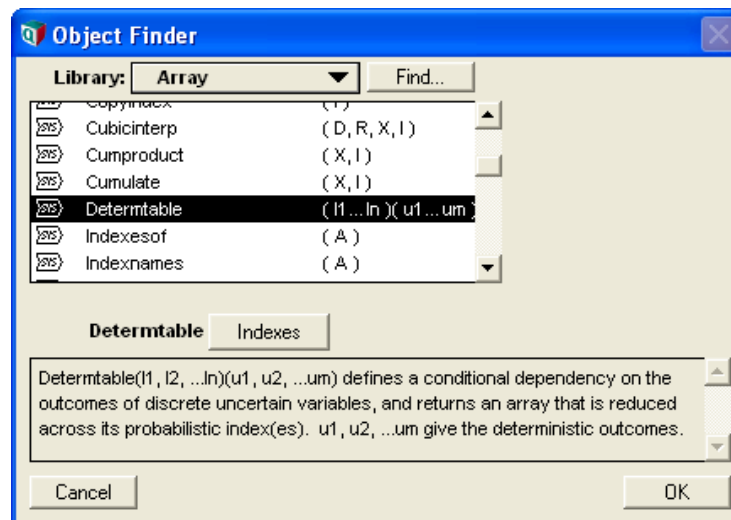
To define a variable as a determtable:

1. Determine the variable's domain — the list of possible outcomes.
2. Press the **Expression** popup menu above the definition field and select **Other**.

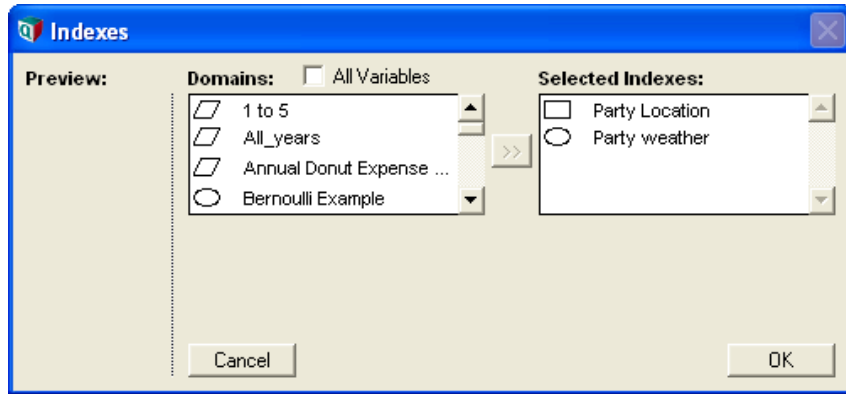


Analytica opens the **Object Finder** dialog box (see “Object Finder dialog” on page 121).

3. Select **Array** from the **Library** popup menu and select **Determtable** from the function list:



4. Click the **Indexes** button to open the **Indexes** dialog, which lets you choose discrete conditioning variables:



5. Click **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 looks like this: You cannot initially create a determtable as an expression.

```
Determtable(i1, i2, ... in) (r1, r2, r3, ... rm)
```

This 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.

Example

In “Create a probability table”, **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**:

	sunny	rainy
outdoors	100	0
porch	90	20
indoors	40	50

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.

Party Location	
outdoors	40
porch	48
indoors	46

Chancedist (p, a, i)

Creates a discrete probability distribution, where **a** is an array of outcome values, numbers or text, 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 **ProbTable()** when:

- The array of outcome **a** is multidimensional, or
- You want to use other variables or expressions to define the outcomes or probability arrays.

Library Distribution

Example **Index_b:**

Red	White	Blue
-----	-------	------

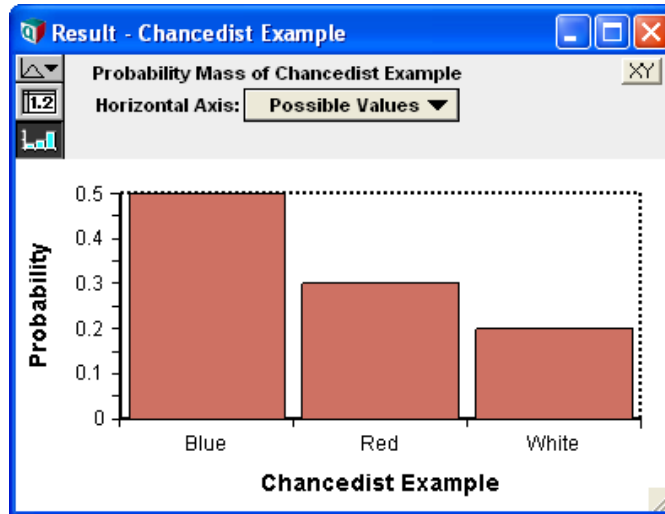
Array_q:

Index_b ►

	Red	White	Blue
	0.3	0.2	0.5

The domain of the variable is a list of labels: ['Red', 'White', 'Blue'].

Chancedist(Array_q, Index_b, Index_b) →



Parametric continuous distributions

Tip To produce the example graphs of distributions below, we used a sample size of 1000, equal sample probability steps, samples per PDF of 10, and we set the graph style to *line*. Even if you use the same options, your graphs may look slightly different due to random variation in the Monte Carlo sampling.

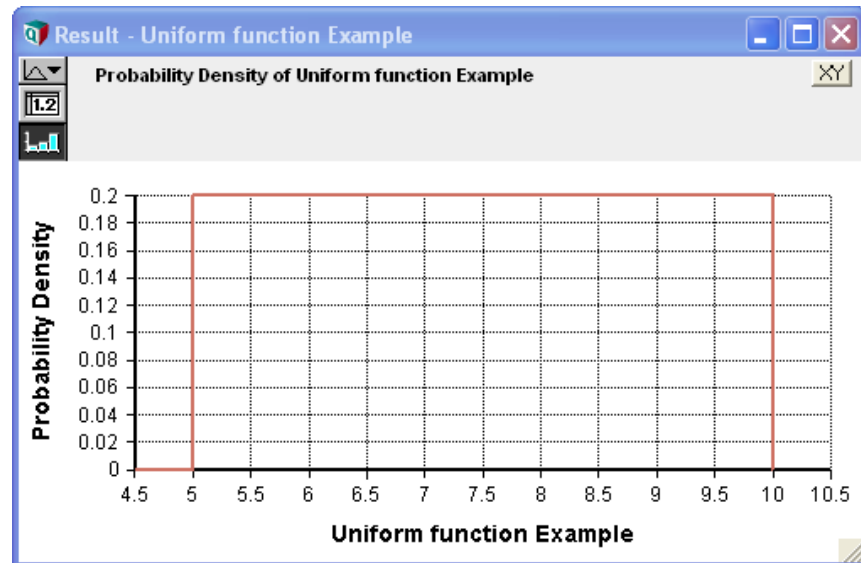
Uniform(min, max)

Creates a uniform distribution between values **min** and **max**. If omitted, they default to 0 and 1. If you specify optional parameter **Integer: True**, it returns a discrete distribution consisting of only the integers between **min** and **max**, each with equal probability. See “Uniform(min, max, Integer:True)”.

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, you know that 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) →`



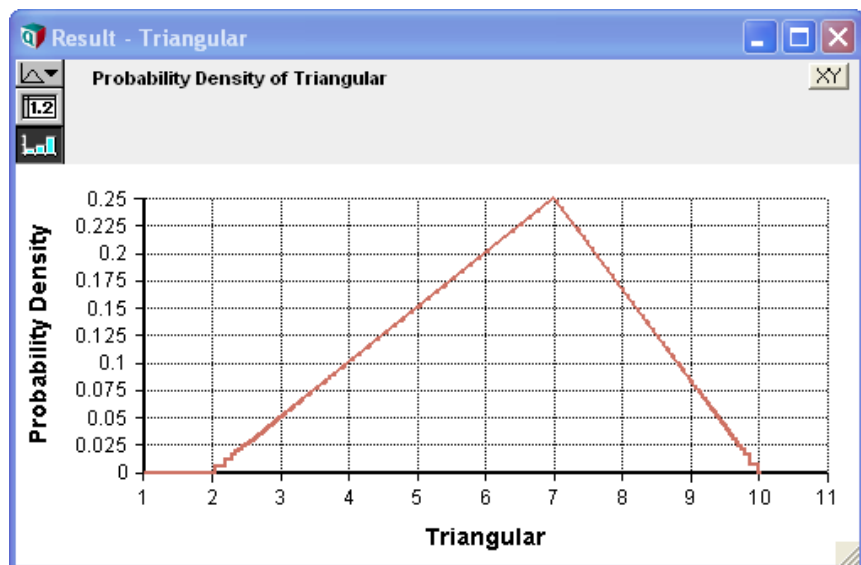
Triangular(min, mode, max)

Creates a triangular distribution, with minimum **min**, most likely value **mode**, and maximum **max**. **min** must 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)` →



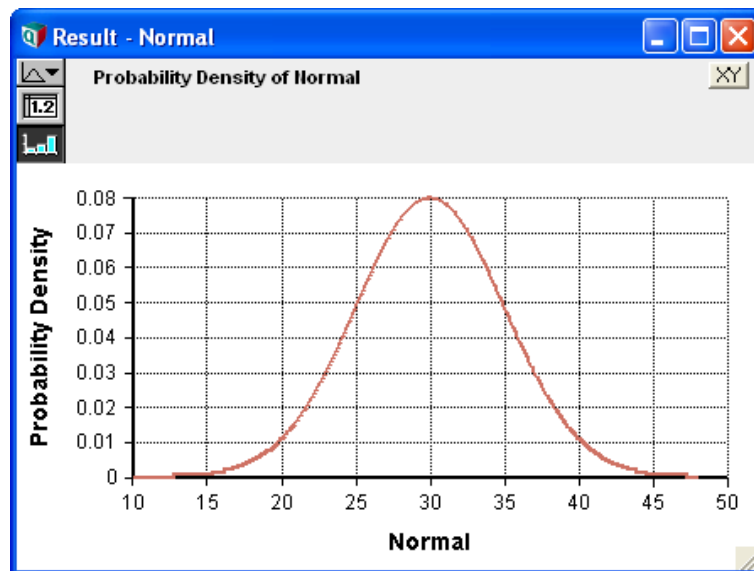
Normal(mean, stddev)

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)` →



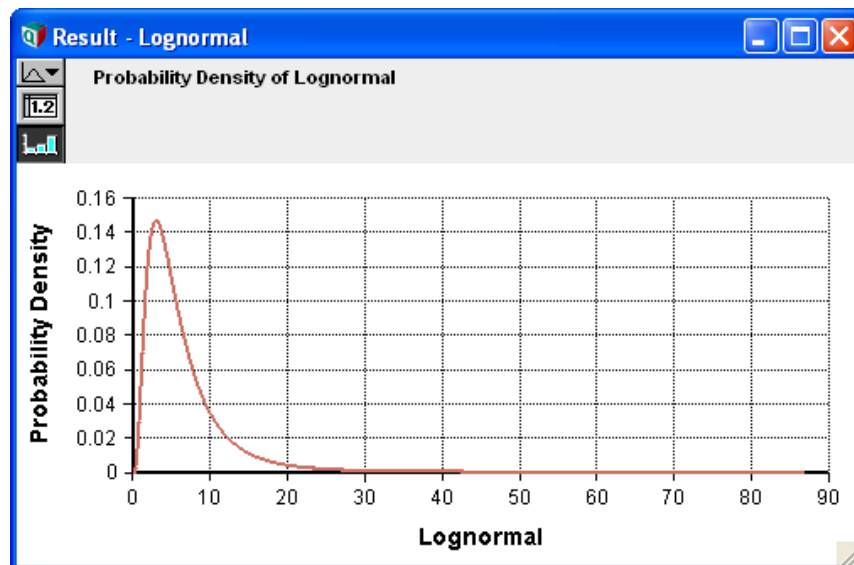
Lognormal(median, gsdev, mean, stddev)

Creates a lognormal distribution. You can specify its **median** and geometric standard deviation **gsdev**, or its **mean** and standard deviation **stddev**, or any two of these four parameters. The geometric standard deviation, **gsdev**, must be 1 or greater. It is sometimes also known as the **uncertainty factor** or **error factor**. The range [**median/gsdev**, **median x gsdev**] encloses about 68% of the probability — just like the range [**mean - stddev**, **mean + stddev**] for a normal distribution with standard deviation **stddev**. **median** and **gsdev** must be positive.

If **x** is lognormal $\ln(\mathbf{x})$ has a normal distribution with mean $\ln(\mathbf{median})$ and standard deviation $\ln(\mathbf{gsdev})$.

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. The lognormal is particularly appropriate if you believe that the uncertain quantity is the product (or ratio) of a large number of independent random variables. The multiplicative version of the central limit theorem says that the product or ratio of many independent variables tends to lognormal — just as their sum tends to a normal distribution.

Library Distribution

Examples `Lognormal(5, 2) →``Lognormal(mean: 6.358, Stddev: 5) →`

Beta (x, y, min, max)

Creates a beta distribution of numbers between 0 and 1 if you omit optional parameters **min** and **max**. **x** and **y** must be positive. If you specify **min** and/or **max**, it shifts and expands the beta distribution so that they form the lower and upper bounds. The mean is:

$$\frac{x}{x+y} \times (max - min) + min$$

When to use Use a beta distribution to represent uncertainty about a continuous quantity bounded by 0 and 1 (0% or 100%) with a single mode. It is particularly useful for modeling an opinion about the fraction (percentage) of a population that has some characteristic. For example, suppose you are trying to estimate the long run frequency of heads, h , for a bent coin about which you know nothing. You could represent your prior opinion about h as a uniform distribution,

`Uniform(0, 1)`

or equivalently,

`Beta(1, 1)`

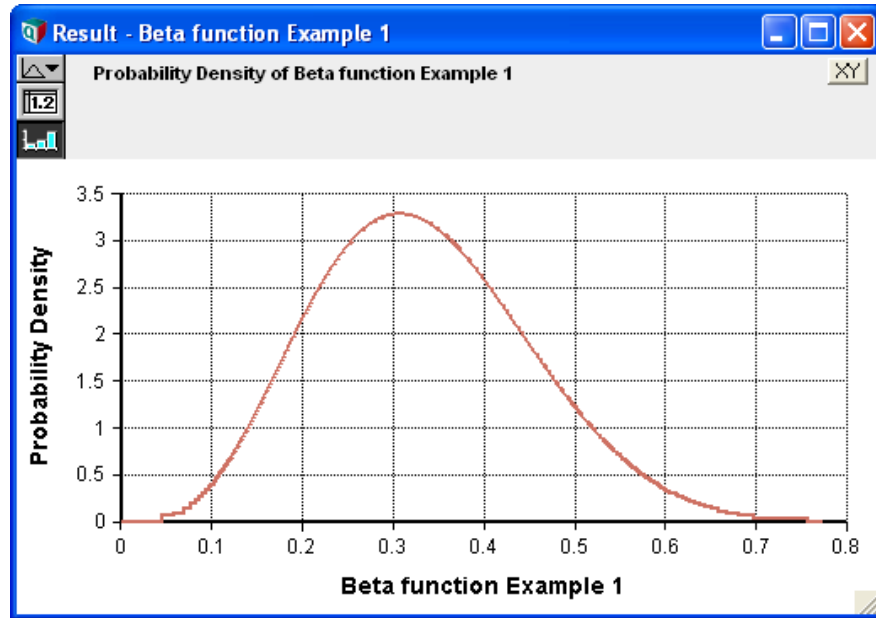
If you observe r heads in n tosses of the coin, your new (posterior) opinion about h , should be:

`Beta(1 + r, 1 + n - r)`

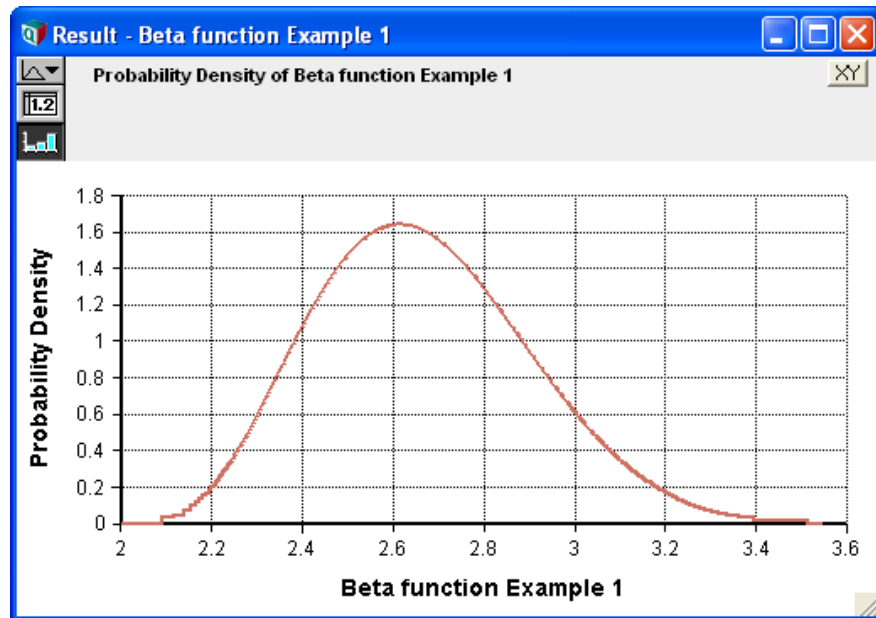
If the uncertain quantity has lower and upper bounds other than 0 and 1, include the lower and upper bounds parameters to obtain a **transformed beta** distribution. The transformed beta is a very flexible distribution for representing a wide variety of bounded quantities.

Library Distribution

Examples Beta (5, 10) →



Beta (5, 10, 2, 4) →



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 $a > b$, the distribution is unimodal with the mode at $(a - 1) \cdot b$. An exponential distribution results when $a = 1$. As $a \rightarrow \infty$, the gamma distribution approaches a normal distribution in shape.

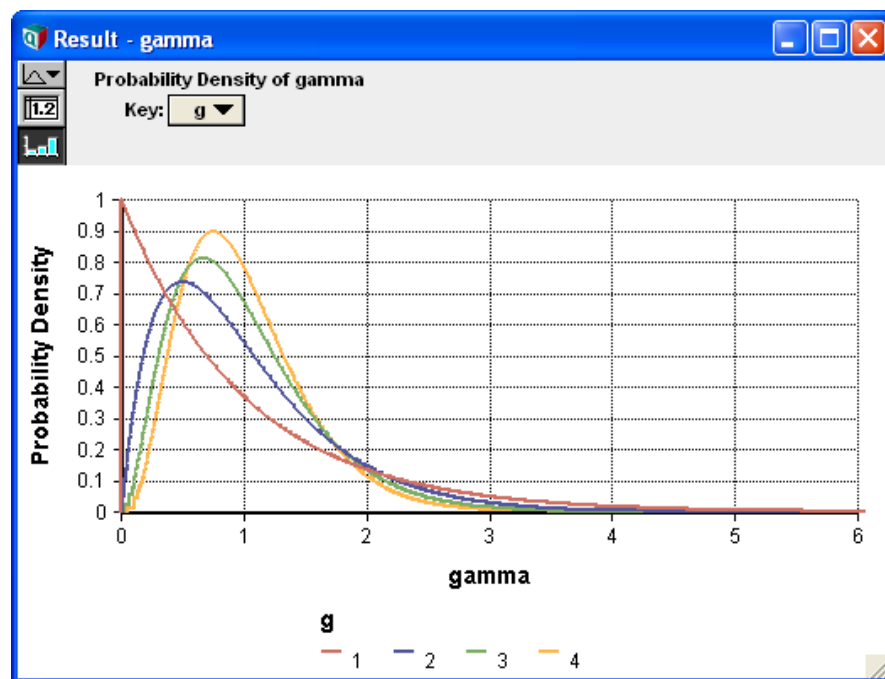
The gamma distribution encodes the time required for **a** events to occur in a Poisson process with mean arrival time of **b**.

Tip Some textbooks use $\text{Rate}=1/\mathbf{b}$, instead of **b**, as the scale parameter.

When to use Use the gamma distribution with $\mathbf{a} > 1$ 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 $\text{mean}=1$:



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:

$$\frac{\pi^2 \times s^2}{3}$$

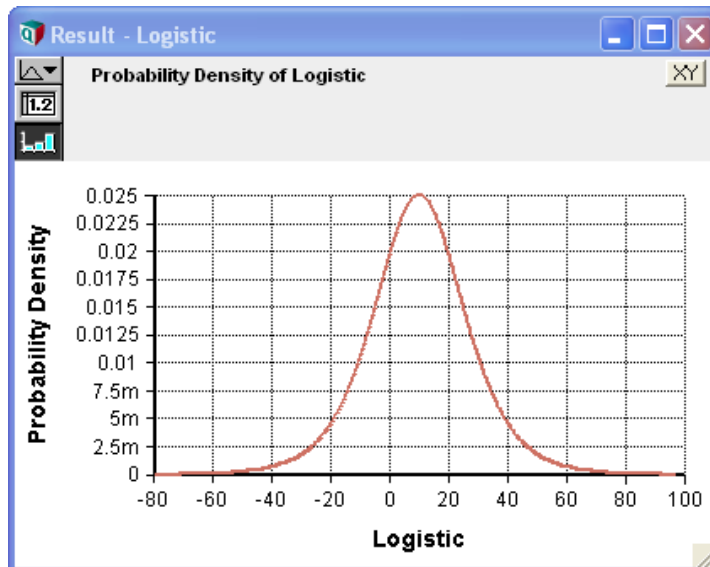
and kurtosis of 6/5 and no skew. The scale parameter, **s**, is optional and defaults to 1.

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

$$\text{Logit}(p(x)) = \text{Ln}(p(x) / (1-p(x))) = x/s - m/s$$

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)`



StudentT(d)

The **StudentT** 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:

$$T = (m - \bar{x}) / (s \text{ Sqrt}(n))$$

where \bar{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 StudentT 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 $x_1 \dots x_n$ measurements are taken to test the hypothesis $m > 0$,

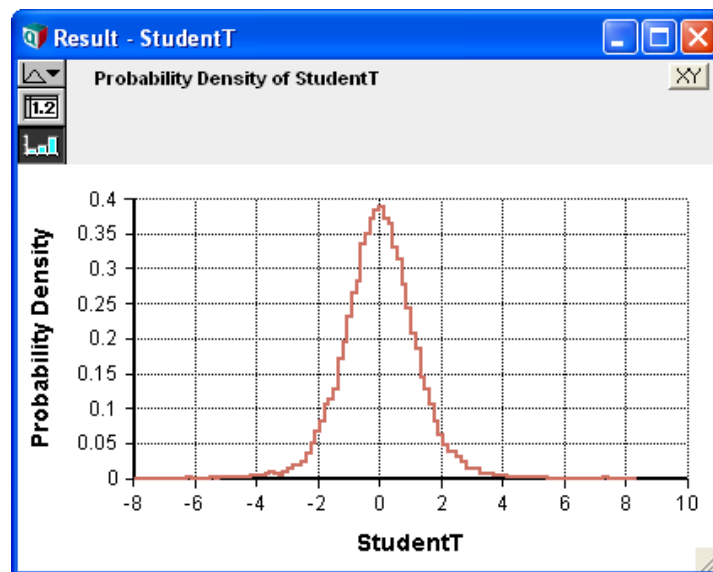
```
GetFract(StudentT(n-1), 0.95)
```

is the acceptance threshold for the T statistic. If T is greater than this fractile, we can reject the null hypothesis (that $m \leq 0$) 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 StudentT 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 StudentT 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 `StudentT(8)`



Weibull(n, s)

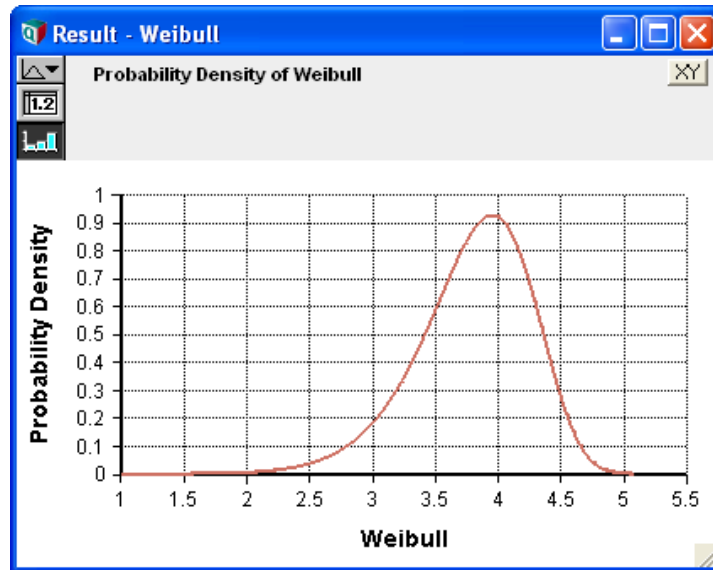
The Weibull distribution has a cumulative density given by:

$$f(x) = 1 - e^{-\left(\frac{t}{s}\right)^n}$$

for $t \geq 0$. It is similar in shape to the gamma distribution, but tends to be less skewed and tail-heavy. It 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)` →



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.

Custom continuous distributions

These functions let you specify a continuous probability distribution by specifying any number of points on its cumulative or density function.

Cumdist(p, r, i)

Specifies a continuous probability distribution as an array of cumulative probabilities, **p**, and an array of corresponding outcome values, **r**. The values of **p** must be non-decreasing, start with 0, and end with 1. The values of **r** must be increasing. Either **r** must be an index of **p**, in which case you can omit **i**, or **p** and **r** must both be indexed by **i**. If **p** or **r** have more than one index, you must specify the common index **i** to link **p** and **r**.

By default, it fits the cumulative distribution using piecewise cubic monotonic interpolation between the specified points, so that the PDF is also continuous. If you set the optional parameter **Smooth** to False, it uses piecewise linear interpolation for the CDF, so that the PDF is piecewise uniform.

Library Distribution

Example

Array_b →

Index_a ▶

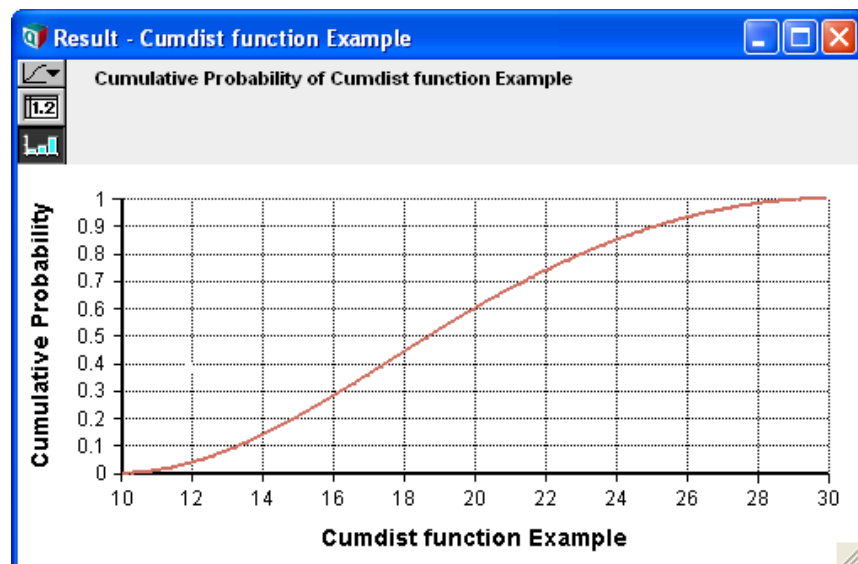
	1	2	3
	0	0.6	1.0

Array_x →

Index_a ▶

	1	2	3
	10	20	30

CumDist(Array_b, Array_x) →



Probdist(p, r, i)

Specifies a continuous probability distribution as an array of probability densities, **p**, for an array of corresponding values, **r**. The values of **r** must be increasing. The probability densities **p** must be non-negative. It normalizes the densities so that the total probability integrates to 1.

Usually the first and last values of **p** should be 0. If not, it assumes zero at $2x_1 - r_1$ (or $2x_n - r_{n-1}$).

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 index **i** to link **p** and **r**.

It produces the density function using linear interpolation between the points on the density function (quadratic on CDF).

Library Distribution

Array_p →

Index_a ►

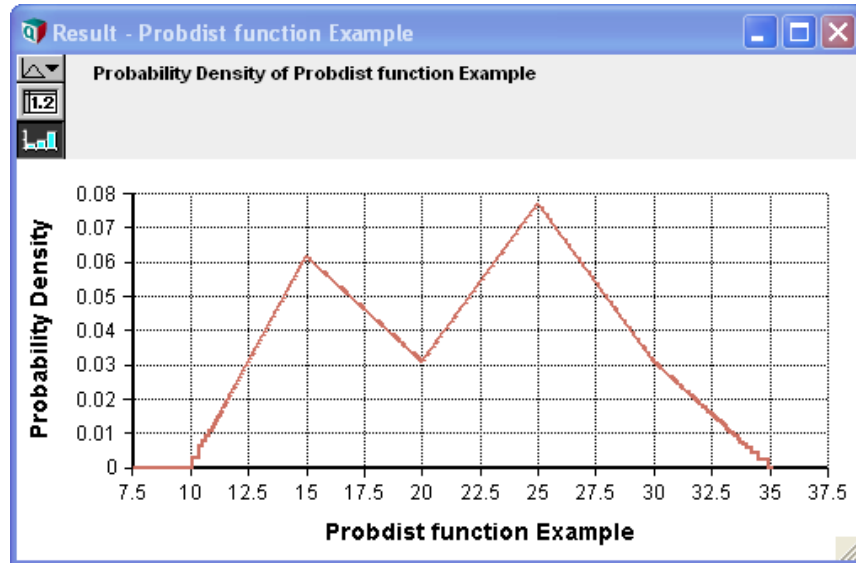
	1	2	3	4	5	6
	0	0.4	0.2	0.5	0.2	0

Array_r →

Index_a ►

	1	2	3	4	5	6
	10	15	20	25	30	35

Probdist(Array_p, Array_r) →



Special probabilistic functions

Certain(u)

Returns the mid (deterministic) value of **u** even if **u** is uncertain and evaluated in a prob (probabilistic) context. It is not strictly a probability distribution. It is sometimes useful in browse mode, when you want to replace an existing probability distribution defined for an input (see “Using input nodes”) with a non-probabilistic value.

Library Distribution

Shuffle(a , i)

Shuffle returns a random reordering (permutation) of the values in array **a** over index **i**. If you omit **i**, it evaluates **a** in prob mode, and shuffles the resulting sample over **Run**. You can use it to generate an independent random sample from an existing probability distribution **a**.

If **a** contains dimensions other than **i**, it shuffles each slice over those other dimensions independently over **i**. If you want to shuffle the slices of a multidimensional array over index **i**, without shuffling the values within each slice, use this method:

```
A[@I = Shuffle(@I, I)]
```

This shuffles **a** over index **i**, without shuffling each slice over its other indexes.

Truncate(u, min, max)

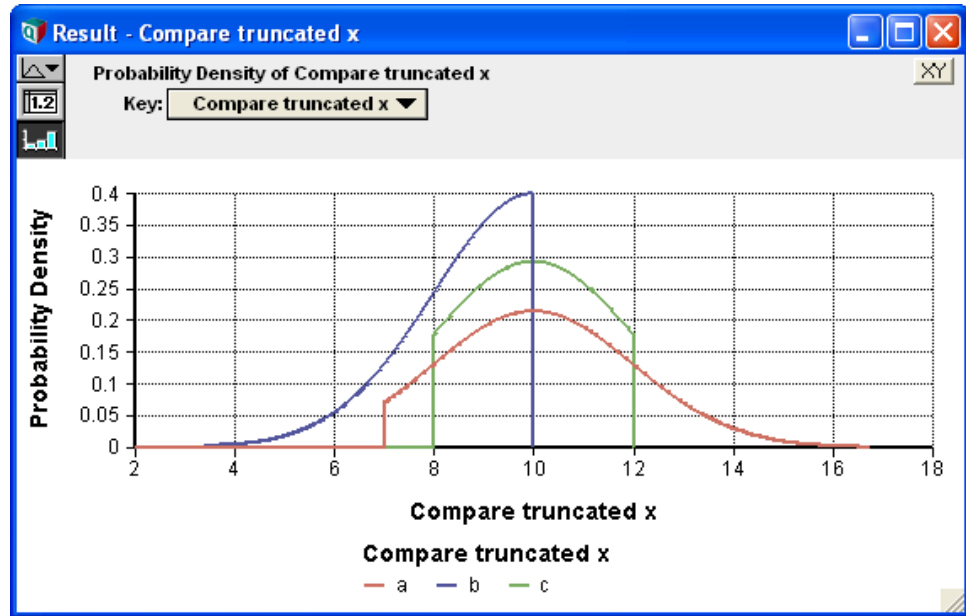
Truncates an uncertain quantity **u** so that it has no values below **min** or above **max**. You must specify one or both **min** and **max**.

It does not discard sample values below **min** or above **max**: It generates a new sample that has approximately same probability distribution as **u** between **min** and **max**, and no values outside them. The values of the result sample have the same rank order as the input **u**, so the result retains the same rank-correlation that **u** had with any predecessor.

It gives an error if **u** is not uncertain, or if **min** is greater than **max**. It gives a warning if no sample values of **u** are in the range **min** to **max**. In mid mode, it returns an estimate of the median of the truncated distribution. Unlike other distribution functions, even in mid mode, it evaluates its parameter **u** (and therefore any of its predecessors) in prob mode. It always evaluates **min** and **max** in mid mode.

Examples We define a normal distribution, **x**, and variables **A**, **B**, and **C** that truncate **x** below, above, and on both sides. Then we define a variable to compare **A**, **B**, and **C** and display its result in the probability density view:

```
Chance X := Normal(10, 2)
Chance A := Truncate(X, 7)
Chance B := Truncate(X, , 10)
Chance C := Truncate(X, 8, 12)
Variable Compare_truncated_x := [A, B, C]
```



Library Distribution

Random(expr)

Generates a single value randomly sampled from **expr**, which, if given, must be a call to a probability distribution with all needed parameters, for example:

```
Random(Uniform(-100,100))
```

returns a single real-valued random number uniformly selected between -100 and 100. If you omit parameter **expr**, it generates one sample from the uniform distribution 0 to 1, for example:

```
Random(Uniform(-100,100)) → 74.4213148
Random() → 0.265569265
```

Random is not a true distribution function, since it generates only a single value from the distribution, whether in mid or probab context. It generates each single sample using Monte Carlo, not Latin hypercube sampling, no matter what the global setting in the uncertainty setup. It is often useful when you need a random number generator stream, such as for rejection sampling, Metropolis-Hastings simulation, and so on.

Random has these parameters, all optional:

Parameters **dist**: If specified, must be a call to a distribution function that supports single-sample generation (see below). Defaults to **Uniform(0,1)**.

Method: Selects the random number generator: 0=default, 1=Minimal standard, 2=L'Ecuyer, 3=Knuth.

Over: A convenient way to list index(es) so that the result is an array of independent random numbers with this index or indexes. For example:

```
Random(Over: I)
```

returns an array of independent uniform random numbers between 0 and 1 indexed by **i**. It is equivalent to:

```
Random(Uniform(0, 1, Over: I))
```

Supported distributions

Random supports *all* built-in probability distribution functions *with the exception of Fractiles, ProbDist, and Truncate*. It supports Bernoulli, Beta, Binomial, Certain, ChiSquared, CumDist, Exponential, Gamma, Geometric, HyperGeometric, Logistic, LogNormal, Normal, Poisson, StudentT, Triangular, Uniform, Weibull. It supports these distributions in the Distribution Variations library: Beta_m_sd, Chancedist, Erlang, Gamma_m_sd, InverseGaussian, Lorenzian, NegBinomial, Pareto, Pert, Rayleigh, Smooth_Fractile, and Wald, and these distributions from the Multivariate Distributions library: BiNormal, Dirichlet, Dist_additive_growth, Dist_compound_growth, Dist_serial_correl, Gaussian, Multinomial, MultiNormal, MultiUniform, Normal_additive_gro, Normal_compound_gro, Normal_correl, Normal_serial_correl, UniformSpherical, Wishart, and InvertedWishart.

User-defined functions can be used as a parameter to **Random**, if they are given an optional parameter declared as:

```
singleSampleMethod: Optional Atom Number
```

If the parameter is provided, the distribution function must return a single random variate from the distribution indicated by the other parameters. The value specifies the random number generator to use: 0=default, 1=Minimal standard, 2=L'Ecuyer, 3=Knuth.

Multivariate distributions

A multivariate distribution is a distribution over an array of quantities — or, equivalently, an array of distributions. Analytica's Intelligent Array features make it relatively easy to generate multivariate distributions. There are three main ways: To create an array of identical independent distributions, use the **Over** parameter. To create an array of independent distributions with different parameters, pass array(s) of parameter values to the function. To create an array of dependent distributions, use a function from the Multivariate Distributions library, which lets you specify a dependence as a correlation, correlation matrix, or covariance matrix. See the following sections for details.

Over indexes as parameters to probability distributions

If you want to generate an array of identical, independent distributions, the simplest method is to specify the index(es) in the **Over** parameter, for example:

```
Normal(10, 2, Over: K)
```

generates an array of independent normal distributions, each with mean 10 and standard deviation 2, over index K . All parametric distributions accept **over**: as an optional parameter. **over** allows multiple indexes if you want to create a multidimensional array of identically distributed quantities. For example, this will generate a three-dimensional array of independent, identically distributed uniform distributions:

```
Uniform(0, 10, Over: I, J, K)
```

Probability distributions with array parameters

Probability distribution functions fully support Intelligent Arrays. If a parameter is an array, the function will generate an array of independent distributions over any index(es) of the array. For example,

```

Index K := ['A', 'B', 'C']
Variable Xmean := Table(K)(10, 11, 12)
Variable X := Normal(Xmean, 2)

```

`x` will be an array of normal distributions over index `κ`, each with the corresponding mean. If you define a normal distribution with two parameters (mean and standard deviation) with the same Index(es) — in this case, `Xmean` and `Ysd` are both indexed by `κ`:

```

Variable Ysd := Table(K)(2, 3, 4)
Variable Y := Normal(Xmean, Ysdeviation)

```

it generates an array of normal distributions over index `κ`, each with corresponding mean and standard deviation. More generally, the result is an array with the union of the indexes of all its parameters — just the same as all other functions and operations that support Intelligent Arrays.

The custom probability distributions, including `ProbTable`, `ProbDist`, and `CumDist`, expect their parameters to be arrays of probabilities, probability densities, or values, with a common index. In this case, the common index is used in generating the random sample and does not appear in the result. But, if those array parameters have any *other* indexes, those indexes will also appear in the result, following the usual rules of Intelligent Arrays.

Multivariate Distribution library

This library offers a variety of functions for generating probability distributions that are dependent or correlated. It is distributed with Analytica. To add this library to your model see “Adding library to a model” on page 331.

Many of these functions specify dependence among distributions using a **rank correlation** number or matrix, also known as the Spearman correlation. Unlike the Pearson or product-moment correlation, rank correlation is a non-parametric measure of correlation. It is equivalent to the Pearson correlation on the ranks of the same. It does not assume that the relationship is linear, and applies to ordinal as well as interval-scale variables. It is therefore a more robust statistic. For example, it is a more stable way to estimate the relationship between two random samples when one or both has a long tail — such as a lognormal distribution. In such cases, Pearson correlation may be misleadingly large (or small) when an extreme sample in the tail of one sample does (or does not) correspond with an extreme value in the other sample.

The methods provided to generate general multivariate distributions with specified rank correlation, first generate multivariate normal (Gaussian) distributions with specified rank correlation, and then transform them to the desired marginal distributions. The rank correlations are not changed by such transformation.

The method for generating the correlated distribution (based on Iman & Conover) works for median and random Latin Hypercube as well as simple Monte Carlo simulation methods. The rank-correlations of the results are approximately, but not exactly, equal to the specified rank-correlations. The accuracy of the approximation increases with the sample size.

Create one distribution dependent on another

Normal_correl(m, s, r, y)

Generates a normal distribution with mean **m**, standard deviation **s**, and correlation **r** with uncertain quantity **y**. In mid mode, it returns **m**. If **y** is not normally distributed, the result will also not be normal, and the correlation will be approximate. It generalizes appropriately if any of the parameters are arrays. The result array will have the union of the indexes of the parameters.

Correlate_with(s, ref, rc)

Reorders the samples of **s** so that the result has the identical values to **s**, and a rank correlation close to **rc** with the reference sample, **ref**.

Example: To generate a lognormal distribution with a 0.8 rank correlation with **Z**, use:

```
Correlate_with(LogNormal(2, 3), Z, 0.8)
```

Note: If you have a non-default **SampleWeighting** of points, the weighted rank correlation may differ from **rc**.

Dist_reshape(x, newdist)

Reshapes the probability distribution of uncertain quantity **x**, so that it has the same marginal probability distribution (i.e., same set of sample values) as **newdist**, but retains the same ranks as **x** over **Run**. Thus:

```
Rank(Sample(x), Run)
= Rank(Sample(Dist_reshape(x, y)), Run)
```

In a Mid context, it simply returns **Mid(newdist)**, with any indexes of **x**.

The result retains any rank correlations that **x** may have with other predecessor variables. So, the rank-order correlation between a third variable **z** and **x** is the same as the rank-order correlation between **z** and a reshaped version of **x**, like this:

```
RankCorrel(x, z) = RankCorrel(Dist_reshape(x, y), z)
```

The operation may optionally be applied along an index **r** other than **Run**.

An array of distributions with correlation or covariance matrix

Correlate_dists(x, rcm, m, i, j)

Given an array **x** indexed by **i** of uncertain quantities, it reorders the samples so as to match the desired rank correlation matrix, **rcm** between the **x[i]** as closely as possible. **rcm** is indexed by **i** and **j**, which must be the same length. It must be positive definite, and the diagonal should be all ones. The result has the same marginal distributions as **x[i]**, and rank correlations close to those specified in **rcm**. In mid mode, it returns **Mid(x)**.

Gaussian(m, cvm, i, j)

Generates a multivariate Gaussian (i.e., normal) distribution with mean vector, **m**, and covariance matrix, **cvm**. **m** is indexed by **i**. **cvm** must be a symmetric and positive-definite matrix, indexed by **i** and **j**, which must be the same length. It is similar to **Multinormal()** except that it takes a covariance matrix instead of a rank correlation matrix.

Multinormal(m, s, cm, i, j)

Generates a multivariate normal (or Gaussian) distribution with mean **m**, standard deviation **s**, and correlation matrix **cm**. **m** and **s** may be scalar or indexed by **i**. **cm** must be a symmetric, positive-definite matrix, indexed by **i** and **j**, which must be the same length. It is similar to **Gaussian**, except that it takes a correlation matrix instead of a covariance matrix.

BiNormal(m, s, i, c)

A 2-D Normal (or bivariate Gaussian) distribution with means **m**, standard deviations **s** (>0) and correlation **c** between the two variables. The index **i** must have exactly two elements. **s** must be indexed by **i**.

Other parametric multivariate distributions**Dirichlet(alpha, i)**

A Dirichlet distribution with parameters **alpha>0** indexed by **i**. Each sample of a Dirichlet distribution produces a random vector indexed by **i** whose elements sum to 1. It is commonly used to represent second order probability information.

The Dirichlet distribution has a density given by:

$$k * \text{Product}(x^{(\text{alpha}-1)}, i)$$

where **k** is a normalization factor equal to:

$$\text{GammaFn}(\text{Sum}(\text{alpha}, i)) / \text{Sum}(\text{GammaFn}(\text{alpha}), i)$$

The **alpha** parameters can be interpreted as observation counts. The mean is given by the relative values of **alpha** (normalized to 1), but the variance narrows as the alphas get larger, just as your confidence in a distribution would narrow as you get more samples.

The Dirichlet lends itself to easy Bayesian updating, if you have a prior of **alpha = 0**, and you have **N** observations.

Multinomial(n, theta, i)

Returns the multinomial distribution, a generalization of the binomial distribution to **n** possible outcomes. For example, if you were to roll a fair die **n** times, the outcome would be the number of times each of the six numbers appears. **theta** would be the probability of each outcome, where **Sum(theta, i)=1**, and index **i** is the list of possible outcomes. If **theta** doesn't sum to 1, it is normalized.

Each sample is a vector indexed by **i** indicating the number of times the corresponding outcome (die number) occurred during that sample point. Each sample has the property

$$\text{Sum}(\text{result}, i) = n$$

UniformSpherical(i, r)

Generates points uniformly on a sphere (or circle or hypersphere). Each sample generated is indexed by **i**, so if **i** has three elements, the points lie on a sphere.

The mid value is a bit strange here since there isn't really a median that lies on the sphere. Obviously the center of the sphere is the middle value, but that isn't in the allowed range. So, it returns an arbitrary point on the sphere.

MultiUniform(cm, i, j, lb, ub)

The multi-variate uniform distribution.

Generates vector samples (indexed by **i**) such that each component has a uniform marginal distribution, and each component has the pair-wise correlation matrix **cm**, indexed by **i** and **j**, which must have the same number of elements. **cm** needs to be symmetric and must obey a certain semidefinite condition, namely that the transformed matrix **[2*sin(30*cov)]** is positive semidefinite. (In most cases, this roughly the same as **cm** being positive semidefinite.) **lb** and **ub** can be used to specify upper and lower bounds, either for all components, or individually if these bounds are indexed by **i**. If **lb** and **ub** are omitted, each component has marginal **Uniform(0, 1)**.

Note: **cm** is the true sample correlation, not rank correlation.

The transformation is based on:

* Falk, M. (1999), "A simple approach to the generation of uniformly distributed random variables with prescribed correlations," *Comm. in Stats - Simulation and Computation* 28: 785-791.

Arrays with serial correlation

These six functions each generate an array of distributions over an index **t** such that each distribution has a specified serial correlation with the preceding element over **t**. They are especially useful for modeling dynamic processes or Markov processes over time, where the value at each time step depends probabilistically on the value at the preceding time. **Normal_serial_correl()** and **Dist_serial_correl()** generate arrays of serially correlated distributions that are normal and arbitrary, respectively.

Normal_additive_gro() and **Dist_additive_growth()** produce arrays with uncertain additive growth with serial correlation. **Normal_compound_gro()** and **Dist_compound_growth()** produce arrays with uncertain compound growth with serial correlation.

Normal_serial_correl(m, s, r, t)

Generates an array of normal distributions over index **t** with mean **m**, standard deviation **s**, and serial correlation **r** between successive values over index **t**. You can give each distribution a different mean and/or standard deviation if **m** and/or **s** are arrays indexed by **t**. If **r** is indexed by **t**, **r[t=k]** specifies the correlation between **result[t=k]** and **result[t=k-1]**. (Then it ignores the first correlation, **r[@t=1]**.)

Dist_serial_correl(x, r, t)

Generates an array **y** over time index **t** where each **y[t]** has a marginal distribution identical to **x**, and serial rank correlation of **rc** with **y[t-1]**. If **x** is indexed by **t**, each **y[t]** has

the same marginal distribution as $\mathbf{x}[\mathbf{t}]$, but with samples reordered to have the specified rank correlation \mathbf{r} between successive values. If \mathbf{r} is indexed by \mathbf{t} , $\mathbf{r}[\mathbf{@t=k}]$ specifies the rank correlation between $\mathbf{y}[\mathbf{@t=k}]$ and $\mathbf{y}[\mathbf{@t=k-1}]$. Then the first correlation, $\mathbf{r}[\mathbf{@t=1}]$, is ignored.

Normal_additive_gro(x, m, s, r, t)

Generates an array of values over index \mathbf{t} , with the first equal to \mathbf{x} , and successive values adding an uncertain growth, normally distributed with mean \mathbf{m} and standard deviation \mathbf{s} . If we denote the result by \mathbf{g} , \mathbf{r} specifies a serial correlation between $\mathbf{g}[\mathbf{@t = k}]$ and $\mathbf{g}[\mathbf{@t=k-1}]$. \mathbf{x} , \mathbf{m} , \mathbf{s} , and \mathbf{r} each may be indexed by \mathbf{t} if you want them to vary over \mathbf{t} .

Dist_additive_growth(x, g, rc, t)

Generates an array of values over index \mathbf{t} , with the first equal to \mathbf{x} , and successive values adding an uncertain growth \mathbf{g} , and serial correlation \mathbf{rc} between $\mathbf{g}[\mathbf{@t = k}]$ and $\mathbf{g}[\mathbf{@t=k-1}]$. \mathbf{x} , \mathbf{g} , and \mathbf{rc} each may be indexed by \mathbf{t} if you want them to vary over \mathbf{t} .

Normal_compound_gro(x, m, s, r, t)

Generates an array of values over index \mathbf{t} , with the first equal to \mathbf{x} , and successive values multiplied by compound growth $1+\mathbf{g}$, where \mathbf{g} is normally distributed with mean \mathbf{m} and standard deviation \mathbf{s} . It applies serial correlation \mathbf{r} between $\mathbf{g}[\mathbf{@t = k}]$ and $\mathbf{g}[\mathbf{@t=k-1}]$. \mathbf{x} , \mathbf{g} , and \mathbf{rc} each may be indexed by \mathbf{t} if you want them to vary over \mathbf{t} .

Dist_compound_growth(x, g, rc, t)

Generates an array of values over index \mathbf{t} , with the first equal to \mathbf{x} , and successive values multiplying by an uncertain compound growth \mathbf{g} , and serial rank correlation \mathbf{rc} between $\mathbf{g}[\mathbf{@t = k}]$ and $\mathbf{g}[\mathbf{@t=k-1}]$. \mathbf{x} , \mathbf{g} , and \mathbf{rc} each may be indexed by \mathbf{t} if you want them to vary over \mathbf{t} .

Uncertainty over regression coefficients

For a description of `RegressionDist()`, `RegressionNoise()`, and `RegressionFitProb()`, see “Uncertainty in regression results”.

Importance weighting

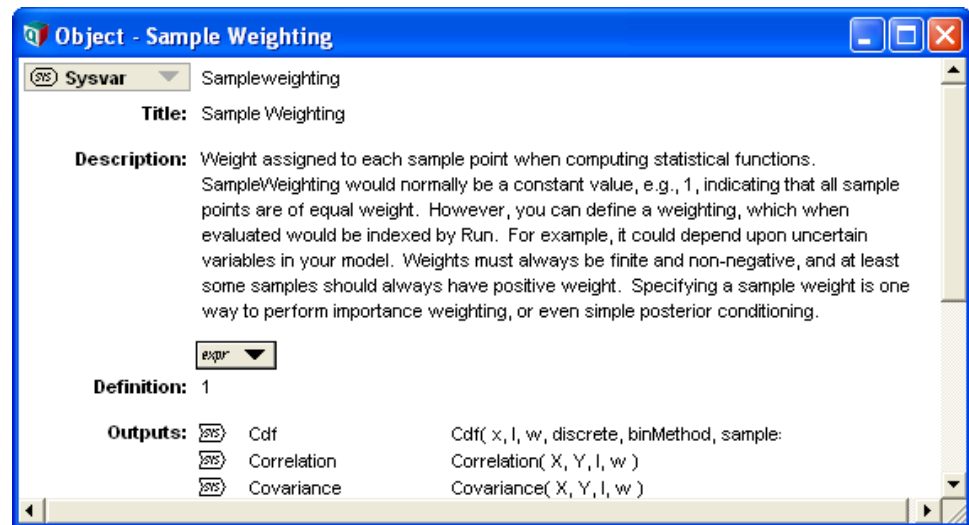
Importance weighting is a powerful enhancement to Monte Carlo and Latin hypercube simulation that lets you get more useful information from fewer samples; it is especially valuable for risky situations with a small probability of an extremely good or bad outcome. By default, all simulation samples are equally likely. With importance weighting, you set `SampleWeighting` to generate more samples in the most important areas. Thus, you can get more detail where it matters and less where it matters less. Results showing probability distributions with uncertainty views and statistical functions reweight sample values using `SampleWeighting` so that the results are unbiased.

You can also modify `SampleWeighting` interactively to reflect different input distributions and so rapidly see the effects the effects on results without having to rerun the

simulation. In the default mode, it uses equal weights, so you don't have to worry about importance sampling unless you want to use it.

SampleWeighting To set up importance weighting, you set weights to each sample point in the built-in variable **SampleWeighting**. Here is how to open its **Object** window:

1. De-select all nodes, e.g., by clicking in the background of the diagram.
2. From the **Definition** menu, select **System Variables**, and then **SampleWeighting**. Its **Object** window opens:



Initially, its definition is 1, meaning it has an equal weight of 1 for every sample. (1 is equivalent to an array of 1s, e.g., `Array(Run, 1)`). For importance weighting, you assign a different weighting array indexed by **Run**. It automatically normalizes the weighting to sum to one, so you need only supply relative weights.

Suppose you have a distribution on variable **X**, with density function **f(x)**, which has a small critical region in **cr(x)** — in which **x** causes a large loss or gain. To generate the distribution on **X**, we use a mixture of **f(x)** and **cr(x)** with probability **p** for **cr(x)** and **(1-p)** for **f(x)**. Then use the `sampleweighting` function to adjust the results back to what they should be is:

$$f(x) / ((p f(x) + (1 - p) cr(x))) \quad (3)$$

For example, suppose you are selecting the design **Capacity** in Megawatts for an electrical power generation system for a critical facility to meet an uncertain **Demand** in Megawatts which has a lognormal distribution:

```
Chance Demand := Lognormal(100, 1.5)
Decision Capacity := 240
Probability(Demand) → 0.015
```

In other words, the probability of failing to meet demand is about 1.5%, according to the probabilistic simulation of the lognormal distribution. Suppose the operator receives **Price** of 20 dollars per Megawatt-hour delivered, but must pay **Penalty** of 200 dollars per megawatt-hour of demand that it fails to supply to its customers:

```
Variable Price := 100
Variable Penalty := 1000
```

```

Variable Revenue := IF Demand <= Capacity THEN Price*Demand
                  ELSE Price*Capacity - (Demand - Capacity)*Penalty
Mean (Revenue) → $2309

```

The estimated mean revenue of \$2309 is imprecise because there is a small (1.5%) probability of a large penalty (\$200 per Mwh that it cannot supply), and only a few sample points will be in this region. You can use Importance sampling to increase the number of samples in the critical region, where `Demand > Capacity`.

```

Chance Excess_demand := Truncate(Demand, 150)
Variable Mix_prob := 0.6
Variable Weighted_demand := If Bernoulli(Mix_prob)
                            THEN Excess_demand ELSE Demand
SampleWeighting := Density(Demand) /
                  ((1 - Mix_prob)*Density(Demand) +
                   Mix_prob*Density(Excess_demand))

```

Thus, we compute a `weighted_demand` as a mixture between the original distribution on `Demand` and the distribution in the critical region, `Excess_demand`. We assign weights to `SampleWeighting`, using the **Object** window for `SampleWeighting` opened as described above. See the Analytica Wiki at <http://www.lumina.com/wiki> for more.

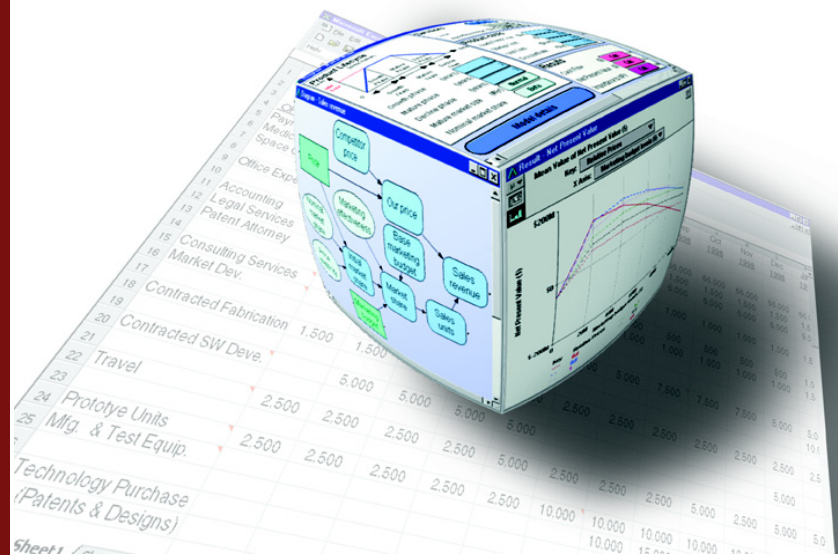
For more on weighted statistics and conditional statistics, see “Weighted statistics and w parameter”.

Chapter 16

Statistics, Sensitivity, and Uncertainty Analysis

This chapter describes:

- Statistical functions that compute statistics, such as mean, variance, or correlation over a probabilistic value (or for arrays with other indexes).
- Tornado charts and importance analysis to see how to apportion credit or blame for the uncertainty in an output to its uncertain inputs.
- XY plots and scatter plots to visualize the effect of an input on an output.
- Functions that show the sensitivity of a variable to one or more variables that affect it, including Whatif and Tornado analysis.
- Functions to perform regression analysis.



Statistical functions

Statistical functions compute a statistic from a probability distribution. More precisely, they estimate the statistic from a random sample of values representing a probabilistic value. Common examples are **Mean**, **Variance**, **Correlation**, and **Getfract** (which returns a fractile or percentile). The uncertainty view options available in the **Result** window (see “Uncertainty views”) use these functions.

Statistical functions force prob mode evaluation

Unlike other functions, statistical function usually force their main parameter(s) to be evaluated in prob mode (probabilistically) and they return a nonprobabilistic value — whether they are evaluated in a mid mode or prob mode. For example:

```
Chance X := Normal(0, 1)
Variable X90 := Getfract(X, .9)
X90 → 1.259
```

Evaluating variable **x90** causes variable **x** to be evaluated in prob mode, so that **Getfract(X, 90%)** can estimate the 90th percentile (0.9 fractile) of the distribution for **x**. **x90** itself has only a mid value, and no probabilistic value. The exception is the **Mid(x)** function that forces **x** to be evaluated in mid mode, no matter the evaluation context.

Statistics from non-probabilistic arrays

The default usage of statistical functions is over a probability distribution, represented as a random sample indexed by **Run**. You can also use these functions to compute statistics over an array with a different index by specifying that index explicitly. This is often useful for computing statistics from data tables — including if you want to fit a probability distribution to a set of data. For example, suppose **Data** is an array of imported measurements:

```
Index K := 1..1000
Variable Data:= Table(K)(123.4, 252.9, 221.4, ...)
Variable Xfitted := Normal(Mean(Data,K), Sdeviation(Data,K))
```

xfitted is a normal distribution fitted to **Data** with the same mean and standard deviation.

Tip

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 how to select a sample size).

Notation in formulas

The formulas used to define statistics use this 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, “Selecting the Sample Size”).

Statistics and text-valued distributions

Most statistical functions require their parameters to be numerical. A few statistical functions, those that only requiring ordinal (ordered) values, also work on distributions with text values (whose domain is a list of labels), namely: **Frequency** (use **Frequency(X, x)**), **Mid**, **Min**, **Max**, **Probability_bands**, and **Sample**. These functions assume the values are ordered as specified in the domain list of labels, e.g. Low, Mid, High.

Example model The examples in this section use the following variables:

```
Variable Alt_fuel_price := Normal(1.25, 0.1)
Variable Fuel_price := Normal(1.19, 0.1)
Variable Skfuel_price := Beta(4,2,1,1.5)
```

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

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

Variance(x)

Returns an estimate of the variance of **x** if **x** is probabilistic. If **x** is non-probabilistic, returns 0.

Variance() 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

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() 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

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:

$$\left(\frac{1}{m} \sum_{i=1}^m \left[\frac{x_i - \bar{x}}{\sigma} \right]^4 \right) - 3$$

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`

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.

```
Index Fract := [0.1,0.9]
Getfract(Fuel_price, Fract) →
Fract ►
```

	0.10	0.90
	1.06	1.32

Library Statistical

Example `Kurtosis(Skfuel_prices) → -0.48`

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").

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

Covariance(x, y)

Returns an estimate of the covariance of uncertain variables **x** and **y**. If **x** or **y** are non-probabilistic, it returns 0. The covariance is a measure of the degree to which **x** and **y** both tend to be in the upper (or lower) end of their ranges at the same time. Specifically, it is defined as:

$$\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$$

Library Statistical

Suppose you have an array **X** of uncertain quantities indexed by **I**,

```
Index I := 1..5
Variable X := Array(I, [...])
```

you can compute the covariance matrix of each element of **X** against each other's element (over **I**), thus:

```
INDEX J := CopyIndex(I)
Covariance(X, X[I=J])
```

We create index **J** as a copy of index **I** and then create a copy of **x** that replaces **I** by **J** so that the covariance is computed for each slice of **x** over **I** against each slice over **J**. The result is the covariance matrix indexed by **I** and **J**. Each diagonal element contains the variance of the variable, since **variance(x) = covariance(x, x)**. You can use this same method to generate a correlation matrix using the **Correlation()** or **Rank_correl()** functions described below.

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`

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”).

Library Statistical

Example With `sampleSize = 100`:
`Rankcorrel(Fuel_price, Alt_fuel_price) → .02`

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”). If you want the relative frequency rather than the count of each value, divide the result by `sampleSize`.

Library Statistical

Example (continuous) `Index 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

Mid(x)

Returns the mid value of **x**. Unlike other statistical functions, **Mid()** 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`

Sample(x)

Forces **x** to be evaluated 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, it just returns its mid value. The system variable **sampleSize** specifies the size of this sample. You can set **sampleSize** in the **Uncertainty Setup** dialog box (see “Uncertainty Setup dialog box”).

Library Statistical

When to use Use when you want to force probabilistic evaluation.

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

Statistics(x)

Returns an array of statistics of **x**. Select the statistics to display in the **Uncertainty Setup** dialog box, Statistics option (see “Uncertainty Setup dialog box”).

Library Statistical

Example `Statistics(Fuel_price) →`
`Statistics ▶`

	Min	Median	Mean	Max	Std. Dev.
	0.93	1.19	1.19	1.45	0.10

PDF(X) and CDF(X)

These functions generate histograms from a sample **X**. They are similar to the methods used to generate the probability density function (PDF) and cumulative probability distribution function (CDF) as uncertainty views in a result window as graph or table. But, as functions, they return the resulting histogram as an arrays available for further processing, display, or export. For example,

```
PDF (X)
CDF (X)
```

evaluate **x** in prob mode, and returns an array of points on the density or cumulative distribution respectively.

You can also use **PDF** and **CDF** to generate a histograms (direct or cumulative) of data that is not uncertain, but indexed by something other than **Run**. For example, to generate a histogram of **x** over index **J**, specify the index explicitly:

```
PDF (Y, J)
```

If it decides that **X** is discrete rather than continuous, **PDF** generates a probability mass distribution and **CDF** generates a cumulative mass distribution, with a probability for each discrete value of **X**. It uses the same method as the uncertainty views in results to decide if **X** is discrete — if it has text values, if it has many repeated numerical values, or if **X** has a Domain attribute that is discrete (see #xref Continuous or Discrete). Alternatively, you may control the result by setting the optional parameter **discrete** as true or false. For example:

```
Variable X := Poisson(20)
PDF(X, Discrete: True)
```

generates a discrete histogram over **x**. If **x** contains text values, i.e., categorical data, you may want to control the order of the categories, e.g., ["Low", "Medium", "High"]. You can do this by specifying the Domain attribute of **x** as a **List of Labels** with these values, or as an **Index**, referring to an Index using them. Alternatively, you can provide PDF or CDF with the optional **Domain** parameter provided as the list of labels. If **x** is an expression rather than a variable, this is your only choice.

PDF and **CDF** have one required parameter:

X The sample data points, indexed by **i**.

and several optional parameters:

i The index over which they generate the histogram. By default this is **Run** (i.e., a Monte Carlo sample) but you can also specify another index to generate a histogram over another dimension.

w The sample weights. Can be used to weight each sample point differently. Defaults to system variable **SampleWeights**.

discrete Set true or false to force discrete or continuous treatment. By default, it guesses, usually correctly.

binMethod Selects the histogramming method used. Otherwise it uses the system default set in the **Uncertainty Setup** dialog from the **Result** menu. Options are:
 0 "equal-X": Equal steps along the X axis (values of X).
 1 "equal-sample-P": Equal numbers of sample values in each step.
 2 "equal-weighted-P": Equal sum of weights of samples, weighted by **w**.

- samplesPerStep** An integer specifying the number of samples per bin. Otherwise, it uses the default **samplesize** set in the **Uncertainty Setup** dialog from the **Result** menu.
- domain** A list of numbers or labels, or the identifier of a variable whose Domain attribute should be used to specify the sequence of possible values for discrete distribution. If omitted, it uses the domain from the sample values.

Weighted statistics and w parameter

Normally, each statistical function gives an equal weight to each sample value in its parameters. You can use the optional parameter **w** for any statistical function to specify unequal weights for its samples. This lets you estimate conditional statistics. For example:

```
Mean(X, w: X>0)
```

computes the mean of **x** for those samples of **x** that are positive. In this case, the weight vector contains only zeros and ones. The expression **x>0** gives a weight of 1 (True) for each sample that satisfies the relationship and 0 (False) to those that do not.

By default, this method works over uncertain samples, indexed by **Run**. You can also use it to compute weighted statistics over other indexes. For example, if **v** is an array indexed by **J**, you could compute:

```
Mean(Y, I, W : Y>0)
```

If you set the system variable **SampleWeighting** to something other than 1 (see “Importance weighting”, all statistical functions use **SampleWeighting** as the default weights, unless you specify parameter **w** with some other weighting array. So, when using importance weighting, all statistics (and uncertainty views) automatically use the correct weighting.

Importance analysis

In a model with uncertain variables, you may want to know 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. 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.

The importance analysis features in Analytica can help you quickly learn which inputs contribute the most uncertainty to the output.

What is importance? This analysis uses as a metric of the "importance" of each uncertain input to a selected output, the absolute rank-order correlation between each input sample and the output sample. 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, such as used in the Tornado analysis, it averages over the entire joint probability distribution. Therefore, it works well even for models where there are strong interactions, where the sensitivity to one input depends on the value of another.

Create an importance variable

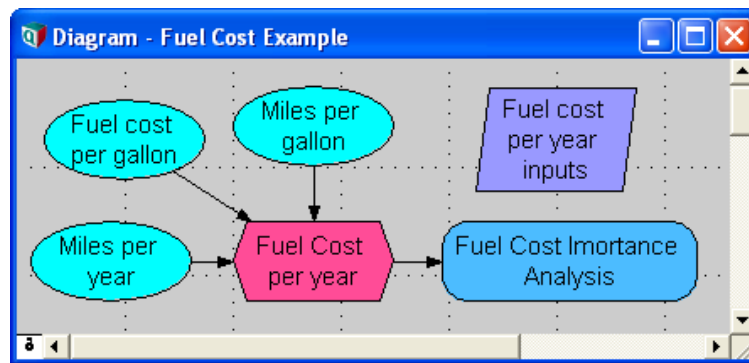
1. Be sure you are in edit mode, viewing a **Diagram** window. Select an output variable, σ , that depends on two or more uncertain inputs — possibly, an objective.
2. Select **Make Importance** from the **Object** menu.

If the selected output is σ , it creates an index σ **Inputs**, a list of the uncertain inputs, and a general variable, σ **Importance**, containing the importance of those inputs to the output.

Example

```
Variable Miles_per_year := Triangular(1,12K,30K)
Variable Fuelcost_per_gallon := Lognormal(3)
Variable Miles_per_gallon := Normal(33,2)
Variable Fuel_cost_per_year :=
(Fuel_cost_per_gallon*Miles_per_year)/Miles_per_gallon
```

After you select **Fuel_cost_per_year** and then **Make Importance** from the **Object** menu, the diagram contains two new variables.



Fuel_cost_per_year 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.

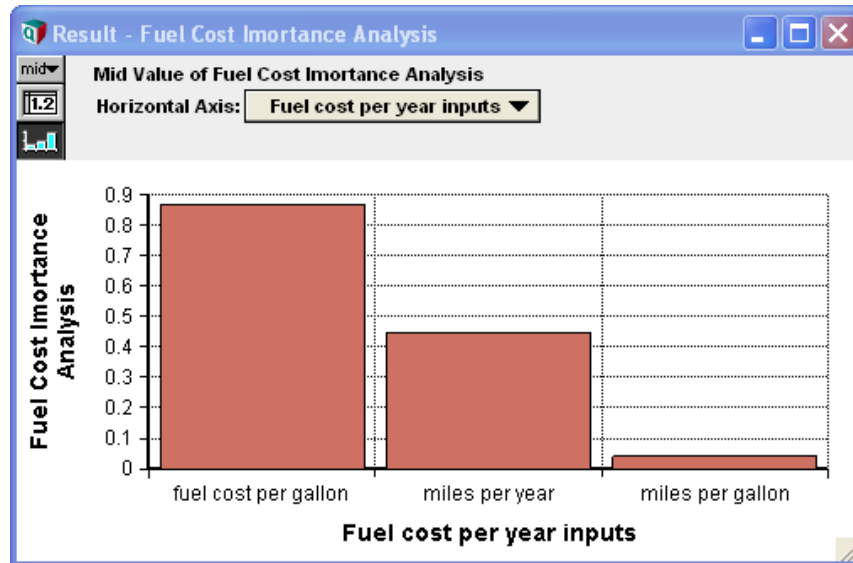
Edit Table of Fuel cost per year inputs	
Fuel cost per year inputs	
fuel cost per gallon	Fuel_cost_per_gallon
miles per year	Miles_per_year1
miles per gallon	Miles_per_gallon1

Fuel cost per year inputs evaluates to a set of probability distributions, one for each chance variable.

Fuel cost per year Importance is defined as

```
Abs(Rankcorrel(Fuel_cost_per_year_inputs, Fuel_cost_per_year))
```

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, `Fuelcost_per_gallon` contributes considerably more uncertainty to `Fuel_cost_per_year` than `Miles_per_gallon`.

Tip Importance, like every other statistical measure, is estimated from the random sample. The estimates may vary slightly from one computation 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 identify two or three that are the primary contributors to the uncertainty in the output. For greater precision, use a larger sample size.

Updating inputs to importance analysis

If you create an importance analysis variable for σ , and later add or remove uncertain variables that affect σ , the uncertainty analysis is not automatically updated to reflect those changes. You may update the analysis either by:

- Select σ and then select **Make Importance** from the **Object** menu. It will automatically update the importance analysis to reflect any new or removed uncertain inputs.
- Draw an arrow from any new uncertain input into index σ **inputs**. It will add the new variable as an uncertain input. Similarly, you can remove a variable from σ **inputs** by redrawing an arrow from that variable into σ **inputs**.

Sensitivity analysis functions

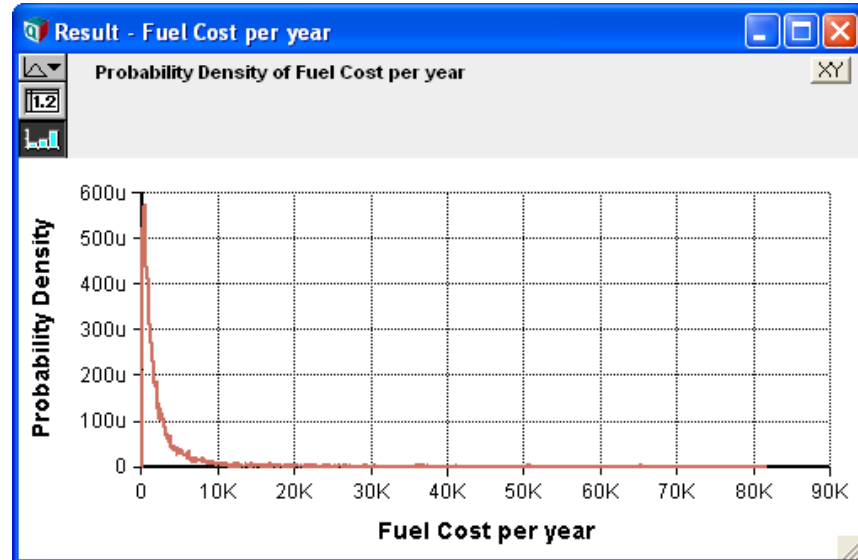
Sensitivity analysis enables you to examine the effect of a change in the value of an input variable on the values of its output variables. They do not require their parameters to be uncertain.

Examples The examples in this section refer to the following variables:

<code>gasPrice</code>	<code>Normal(1.3, .3)</code>	Cost of gasoline per gallon within market fluctuations
<code>mpy:</code>	<code>12K</code>	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
```

Tip When you evaluate `DyDx()` in 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:

$$\text{Elasticity}(y, x) = \text{Dydx}(y, x) * (x/y)$$

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.

Tip 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.

Whatif(e, v, vNew)

Returns the value of expression **e** when variable **v** is set to the value of **vNew**. **v** must be a variable. It lets you explore the effect of a change to a value without changing it permanently. It restores the original definition of **v** after evaluating **Whatif()** expression, so that there is no permanent change (and so causes no side effects).

Library Special

```
Example      Fuelcost → 557.1
                Whatif(Fuelcost, Mpy, 14K) → 650
```

WhatifAll(e, vList, vNew)

Like **Whatif**, but it lets you examine a set of changes to a list of variables, **vList**. It returns the mid value of **e** when each of variables in **vList** is assigned the value in **x** one at a time, with the remaining variables remaining at their nominal values. The result is indexed by **vList**. If **vNew** is indexed by **vList**, it assigns the corresponding value of **vNew** to each variable, letting you assign a different value to each variable in **vList**. **WhatifAll()** is useful for performing *ceteris paribus* style sensitivity analysis, which varies one variable at a time, leaving the others at their initial value, such as in Tornado charts (see next section for an example).

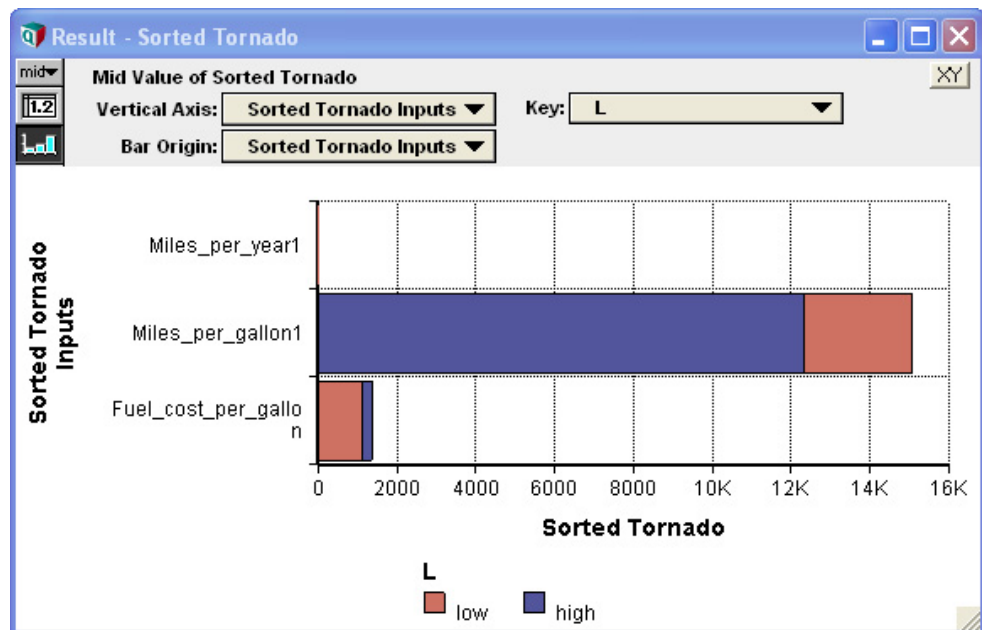
Suppose Z is a function of A , B , and C , and we wish to examine the effect on Z when each input is varied, one at a time, by 10% from its nominal value. Define:

```
Variable Z := 10*A + B^2 + 5*C
Index L := [90%,110%]
Variable V := [A, B,C]
MyTornado := WhatIfAll(Z, V, L*V )
```

Library Special

Tornado charts

A tornado diagram is a common tool used to depict the sensitivity of a result to changes in selected variables. It shows the effect on the output of varying each input variable at a time, keeping all the other input variables at their initial (nominal) values. Typically, you choose a "low" and a "high" value for each input. 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.



Create a tornado analysis

To perform a tornado analysis, you must:

1. Select the result or output variable to perform the analysis on.
2. Select the input variables which may affect the output.
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.*

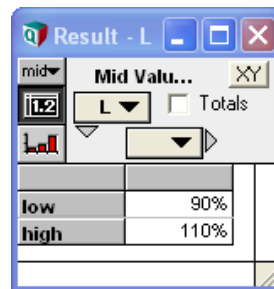
There are several options for selecting low and high values, including:

- 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_Analysis` 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 Tornadoes, or select a single sort order based on whatever criteria fits your needs, realizing that not all tornadoes 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


It's usually best to graph a tornado switching X and Y axis, so that the names of the input variables are listed down the vertical axis, and the effect on the output along the horizontal axis:

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**.

X-Y plots

When evaluating a variable, you can specify another variable to view it against, for **Mid**, **Mean**, **Statistics**, **Probability Bands**, and **Sample**.

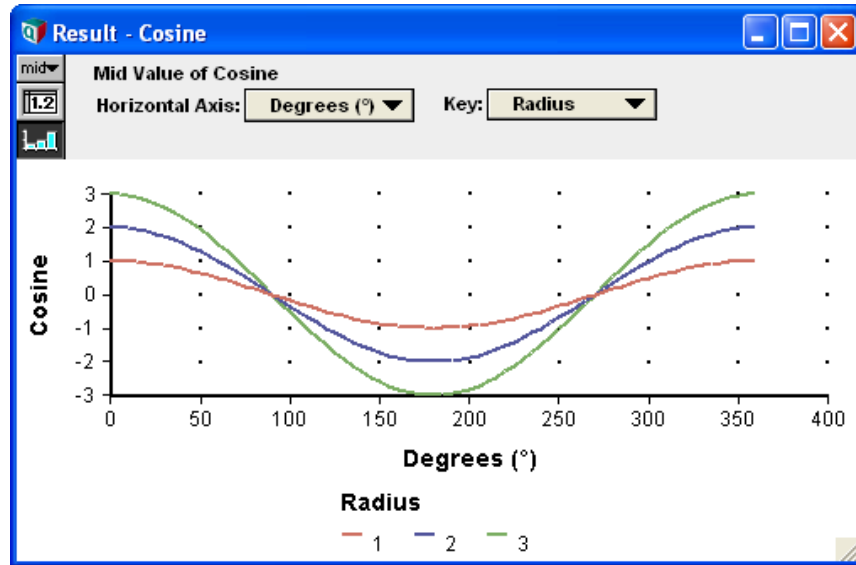
To graph one variable against another:

1. Open a **Result** window for the **y-** (vertical axis) variable.
2. Click 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

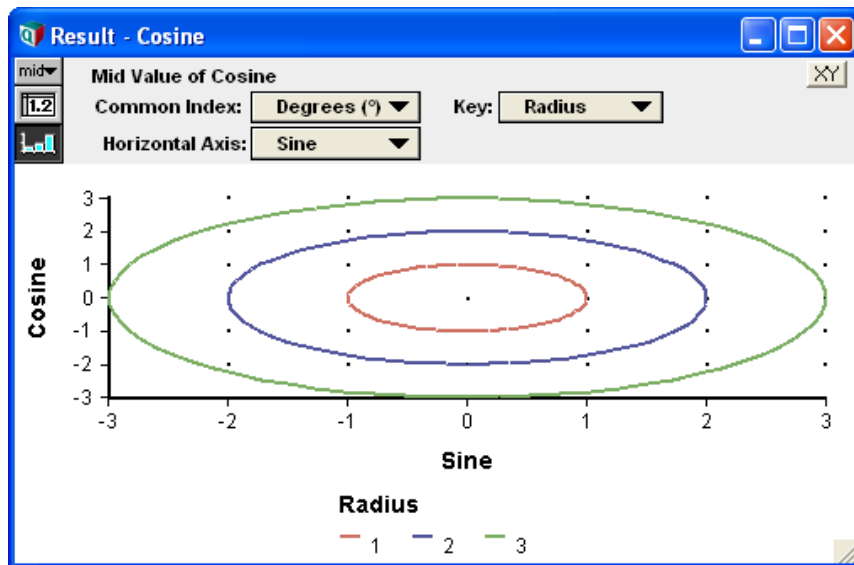
The two variables in an **XY** window must share at least one index, and all indexes of **x** must also be indexes of **y**. The popup menu in the index selection area becomes **Common Index** — only indexes of both **x** and **y** may be selected.

```
Variable Degrees := Sequence(0,360,10)
Variable SinX := Sin(Degrees)
```

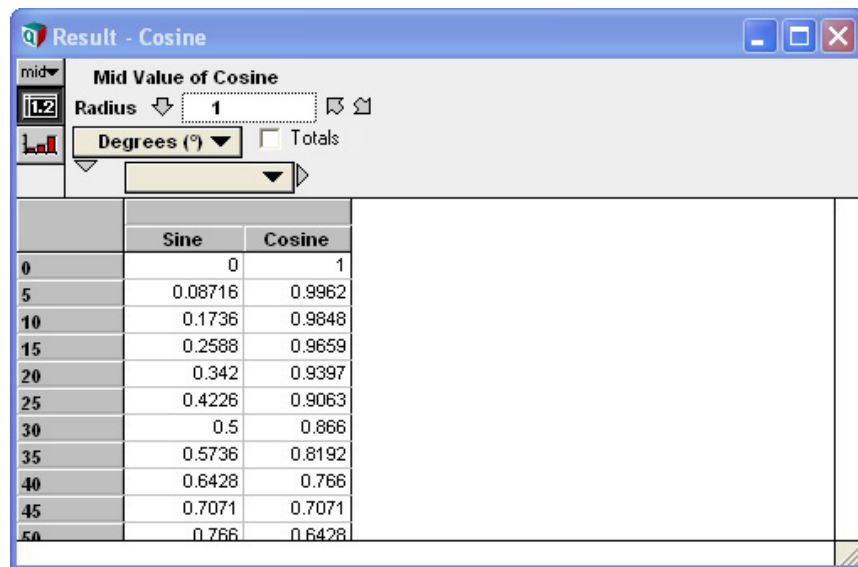
Cosine: Cos(Degrees) →



Click the **XY** button. In the **Object Finder** dialog under **Current Module** select the variable **sine** to display:



Click the **Table View** button to display:



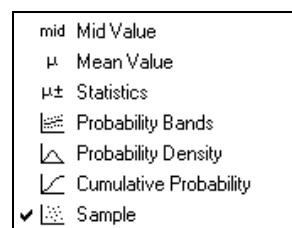
To return to the graph or table of **Cosine** vs. **Degrees**, click in the **XY** check box.

Scatter plots

A **scatter plot** graphs the samples of two probabilistic variables against each other, and provides 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 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.

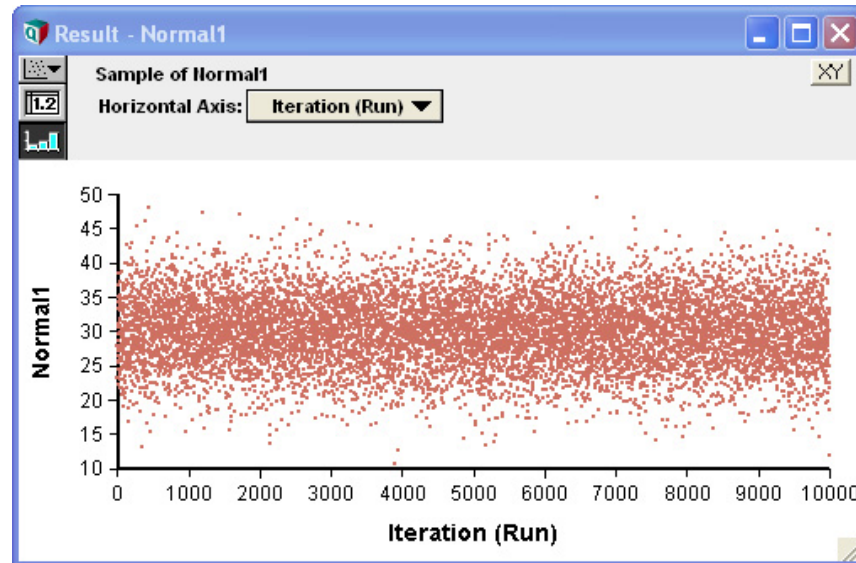
If the variables are independent, the scatter plot points will fall randomly on the graph. If the variables are totally dependent, the scatter plot points will fall along a single line. The strength of the relationship is indicated by the degree to which the points are close to a line. If the line is straight, the relationship is linear; if the line is curved, the relationship is nonlinear.

You can superimpose several scatter plots of \mathbf{y} in an array of uncertain quantities depending on \mathbf{x} . The different quantities will be represented by differently colored dots or symbols.

Example \mathbf{x} : `Uniform(1, 2)`

\mathbf{y} : `Normal(1.0, 3)`

The resulting scatter plot, of two independent variables, is:



Regression analysis

Regression is a widely used statistical method to estimate the effects of a set of inputs (independent variables) on an output (the dependent variable). It is a powerful method to estimate the sensitivity of the output to a set of uncertain inputs. Like the rank-correlation used in importance analysis (see “Importance analysis”), it is a global measure of sensitivity in that it averages the sensitivity over the joint distribution of the inputs, unlike Tornado analysis that is local, meaning it varies each variable one at a time, leaving all others fixed at a nominal value.

Regression() is in the built-in Statistics library, and works with all editions of Analytica. The logit and probit regression functions are in an add-in library, **Generalized Regression.ana**, and require Analytica Optimizer.

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()** can 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:

```
Index K := ['m', 'b']
```

Next, define **B** as a table indexed by **k**:

```
Variable 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:

```
Variable 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**.

Logistic_Regression(Y, B, I, K)

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

This function requires Analytica Optimizer.

The **Logistic_regression** function returns the best-fit coefficients, **c**, for a model of the form:

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

given a data set basis **B**, with each sample classified as **y_i**, having a classification of 0 or 1.

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

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

Probit_regression(Y, B, I, K)

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

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

Given a set of data points, indexed by **I**, with each point classified as **0,1** in the **Y** parameter, and a set of basis terms, **B**, containing the dependent variables (where the vector of dependent variables is indexed by **K**), the **Probit_regression** function finds and returns the set of coefficients for the probit model:

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

where Φ is the inverse cumulative normal distribution function.

The basis, **B**, is a function of the dependent variables in your data. Each element along **K** of the basis vector may be an arbitrary, even non-linear, combination of the data in your data set. However, the number of terms in the basis should be kept small relative to the number of data point in your data set.

Library Generalized Regression.ana

This function requires Analytica Optimizer.

Uncertainty in regression results

These functions help estimate the uncertainty in the results from a regression analysis, including uncertainty in the regression coefficients and the noise. Together they are useful for generating a probability distribution that represents the uncertainty in the predictions from a regression model. When applying regression to make projections into the future based on historical data, there may be additional sources of uncertainty because the future may be different from the past. These functions estimate uncertainty due to noise and imperfect fit to the historical data. You may wish to add further uncertainty for projections into the future to reflect these additional differences.

RegressionDist(y, b, i, k)

RegressionDist estimates the uncertainty in linear regression coefficients, returning probability distributions on them. Suppose you have data where **Y** was produced as:

$$Y = \text{Sum}(C*B, K) + \text{Normal}(0, S)$$

S is the measurement noise. You have the data **B[I,K]** and **Y[I]**. You might or might not know the measurement noise **S**. So you perform a linear regression to obtain an estimate of **C**. Because your estimate is obtained from a finite amount of data, your estimate of **C** is itself uncertain. This function returns the coefficients **C** as a distribution (i.e., in sample mode, it returns a sampling of coefficients indexed by **Run** and **K**), reflecting the uncertainty in the estimation of these parameters.

Library Multivariate Distributions.ana

Examples If you know the noise level **S** in advance, then you can use historical data as a starting point for building a predictive model of **Y**, as follows:

```
{ Your model of the dependent variables: }
Variable Y := your historical dependent data, indexed by I
Variable B := your historical independent data, indexed by I,K
Variable X := { indexed by K. Maybe others. Possibly uncertain }
Variable S := { the known noise level }
Chance C := RegressionDist(Y,B,I,K)
Variable Predicted_Y := Sum(C*X,K) + Normal(0,S)
```

If you don't know the noise level, then you need to estimate it. You'll need it for the normal term of **Predicted_Y** anyway, and you'll need to do a regression to find it. So you can pass these optional parameters into **RegressionDist**. The last three lines above become:

```
Variable E_C := Regression(Y,B,I,K)
Variable S := RegressionNoise( Y,B,I,K,E_C )
Chance C := RegressionDist(Y,B,I,K,E_C)
Variable Predicted_Y := Sum(C*X,K) + Normal(0,S)
```

If you use **RegressionNoise** to compute **S**, you should use **Mid(RegressionNoise(...))** for the **S** parameter. However, when computing **S** for your prediction, don't use **RegressionNoise** in context. Better is if you don't know the measurement noise in advance, don't supply it as a parameter.

RegressionFitProb(y, b, i, k, c)

When you've obtained regression coefficients **C** (indexed by **K**) by calling the **Regression** function, this function returns the probability that a fit this poor would occur by chance, given the assumption that the data was generated by a process of the form:

$$Y = \text{Sum}(C*B, K) + \text{Normal}(0, S)$$

If this result is very close to zero, it probably indicates that the assumption of linearity is bad. If it is very close to one, then it validates the assumption of linearity.

Library Multivariate Distributions.ana

This is not a distribution function — it does not return a sample when evaluated in sample mode. However, it does complement the multivariate **RegressionDist** function also included in this library.

Example To use, first call the **Regression** function, then you must either know the measurement knows a priori, or obtain it using the **RegressionNoise** function.

```
Var E_C := Regression(Y,B,I,K);
Var S := RegressionNoise(Y,B,I,K,C);
Var PrThisPoor := RegressionFitProb(Y,B,I,K,E_C,S)
```

RegressionNoise(y, b, i, k, c)

When you have data, **Y[I]** and **B[I,K]**, generated from an underlying model with unknown coefficients **C[k]** and **S** of the form:

$$Y = \text{Sum}(C * B, I) + \text{Normal}(0, S)$$

This function computes an estimate for **S** by assuming that the sample noise is the same for each point in the data set.

When using in conjunction with **RegressionDist**, it is most efficient to provide the optional parameter **C** to both routines, where **C** is the expected value of the regression coefficients, obtained from calling **Regression(Y,B,I,K)**. Doing so avoids an unnecessary call to the built-in **Regression** function.

Library Multivariate Distributions.ana

These functions express uncertainty in the coefficients of a linear regression. If you are using results from a linear regression, you can use these functions to estimate uncertainty in predictive distributions.

These uncertainties reflect only the degree to which the regression model fits the observations to which it was fit. They do not reflect any possible systematic differences between the past process that generated those observations and the process generating the results being predicted, usually in the future. In this way, they are lower bounds on the true uncertainty.

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.

Tip Read Chapter 11, “Arrays and Indexes,” before using these features.

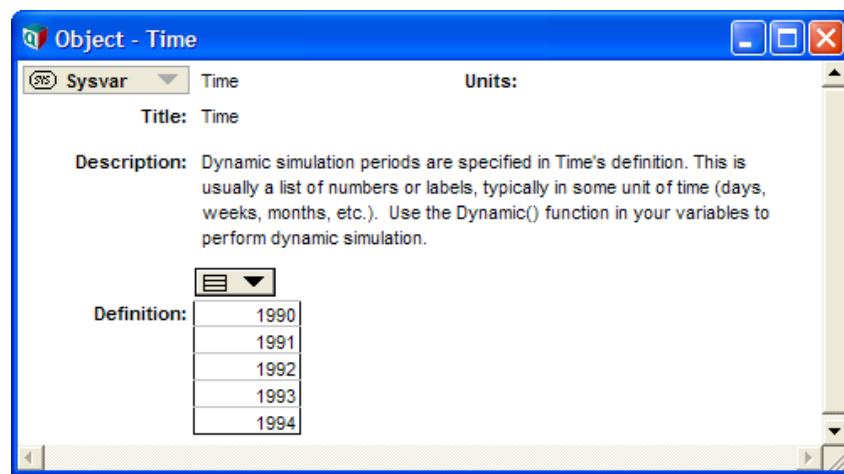
The term *dynamic* is used in this chapter to refer to the **Dynamic()** function.

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.

Tip 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.

Tip 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.

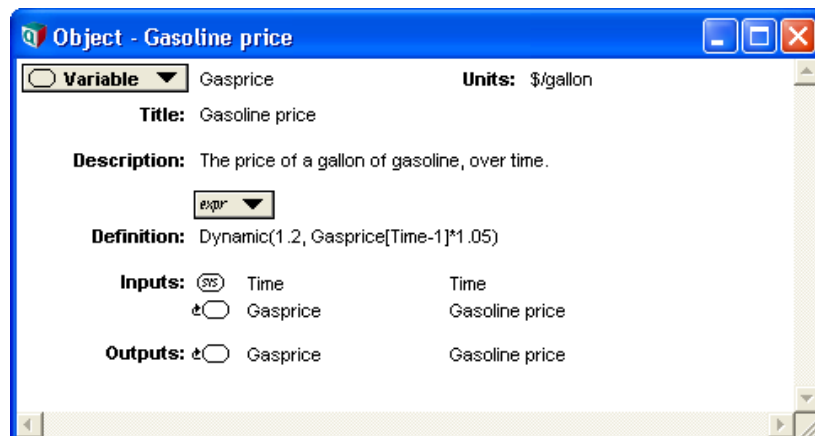
When a dynamic variable refers to itself, it appears in its own list of inputs and outputs, with a symbol for cyclic dependency: .

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 the **Result** button and viewing the mid value as a table displays the following results:

Time	Mid Value of Gasoline price (\$/gallon)
1990	1.2
1991	1.26
1992	1.323
1993	1.389
1994	1.459

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

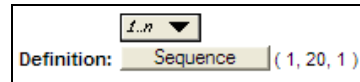
There are three ways to define the **Time** index, each of which has different advantages:

- 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” and “Creating an array with an edit table”). The numeric values for **Time** can be used in other expressions.

Example



Time as a list (numeric)

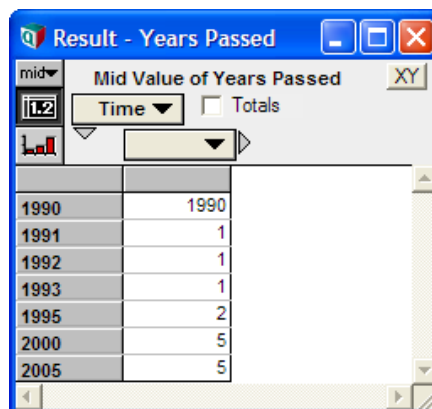
When **Time** is defined as a numeric list, it will usually consist of increasing numbers. The intervals between entries can be unequal, and the values for **Time** can be used in other expressions.

Example **Time:**

1990
1991
1992
1993
1995
2000
2005

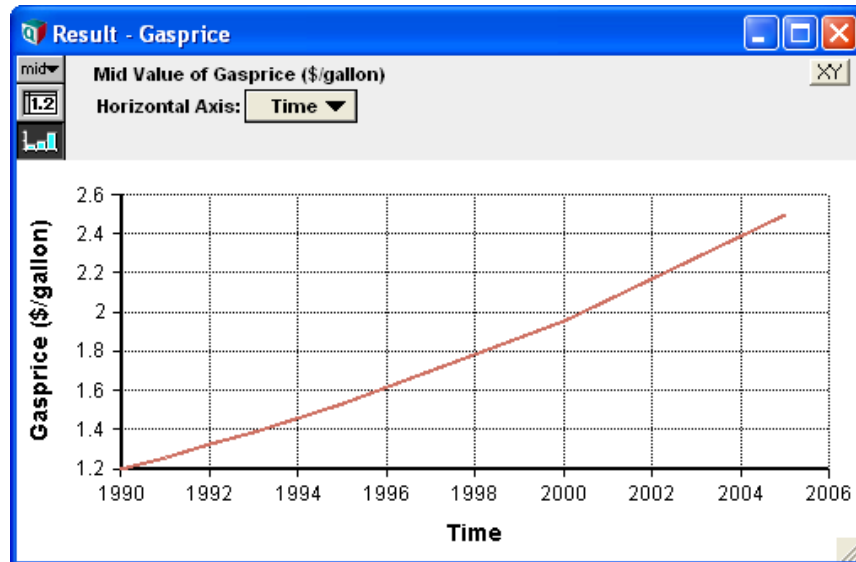
When you use time periods that differ by a value other than 1, typing **(Time-1)** won't provide the value of the previous time period. You can use the syntax **x[Time-1]** if you want to utilize a variable indexed by **Time**, but if you want to perform an operation that depends on the difference in time between the current time period and the last one, you must first create a node that uncumulates the **Time** index:

YearsPassed: Uncumulate(Time)



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 **Dynamic()** definition 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)

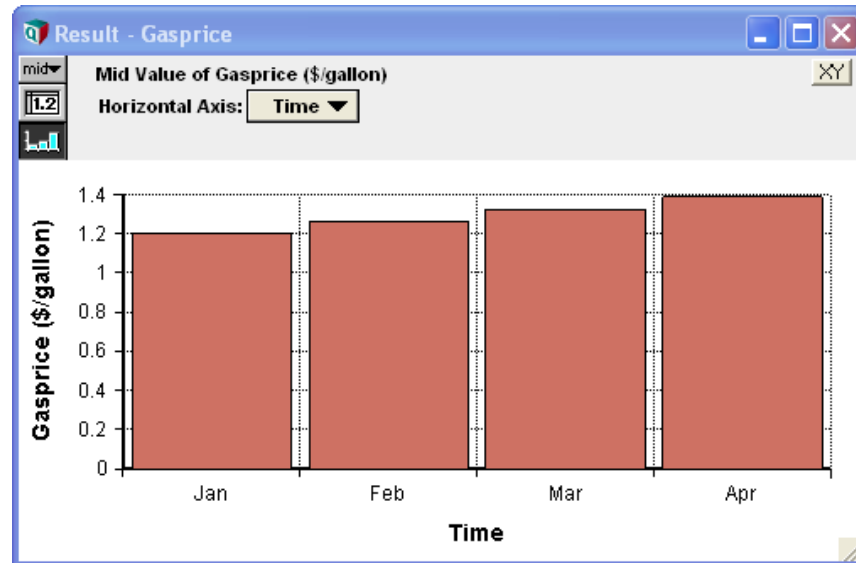
When **time** is defined as a list of labels, **time** values cannot be used in other expressions as numbers.

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 “An alias is like its original”).

When the **time** node displays on a diagram, arrows from **time** to all dynamic variables display by default.

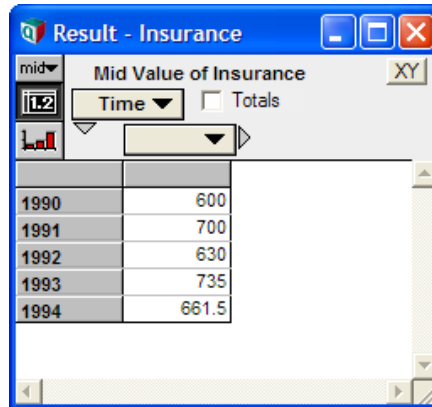
Initial values for Dynamic

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:

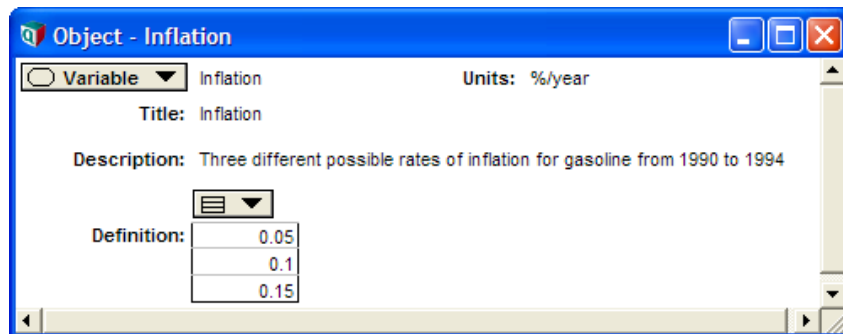
Dynamic(600, 700, Insurance[Time - 2] * 1.05) →



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”), 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

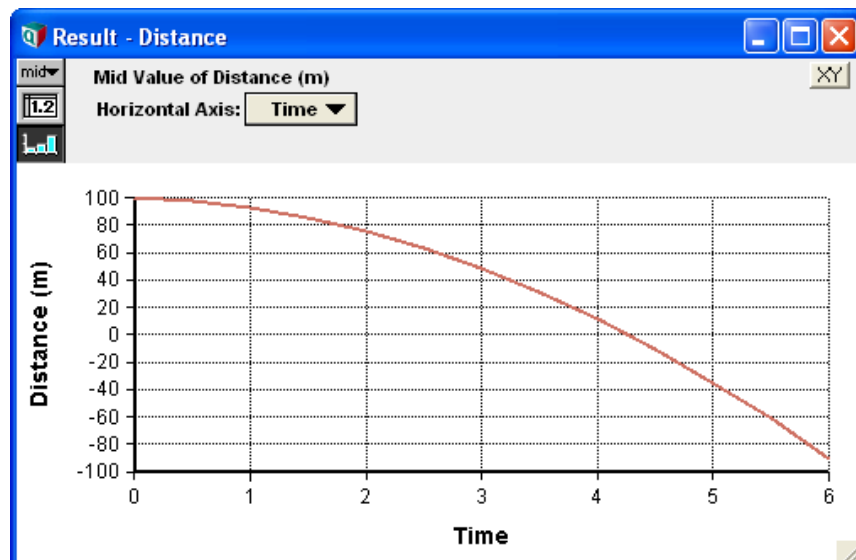
Time: Sequence(0, 6, Dt)

Velocity:

Dynamic(0, Self[Time-1] + Acceleration * Dt)

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”).
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 identifier 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 an atom 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.

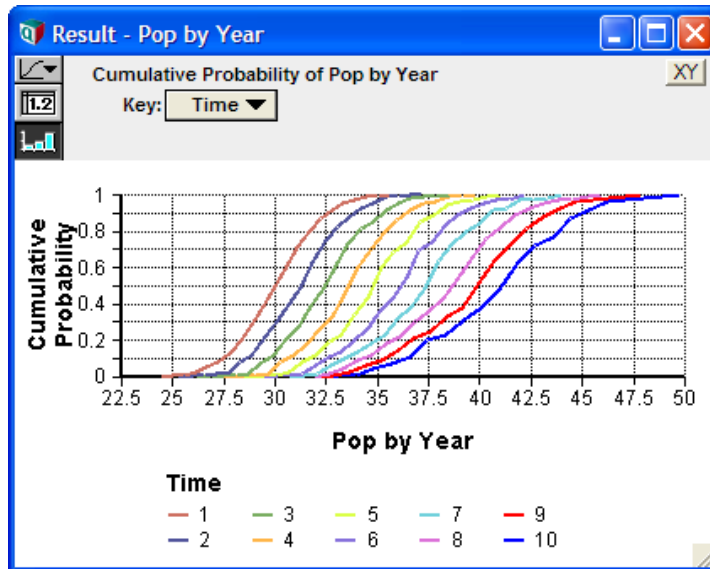
If you attempt to operate over the **Time** dimension from within a dynamic loop, Analytica issues the warning: "Encountered application of an array function over the Time index from within a dynamic loop. The semantics of this operation may be different than you expect."

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:

```
Variable Population := Normal(30, 2)
Variable Birthrate := Normal(1.2, .3)
Time := 1 ..10
Variable 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)
```


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 (*Control+v*).

Tip

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 Character Suffixes 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 (*Control+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 (*Control+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").

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** (*Control+c*) menu to copy a PICT representation of the graph.

Using OLE to link results to other applications

Object Linking and Embedding (OLE) is a widely used Microsoft technology that enables objects in two applications to be hotlinked, 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 the Analytica model is automatically 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 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.

Tip The external application must support OLE-linking of tab-delimited text 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

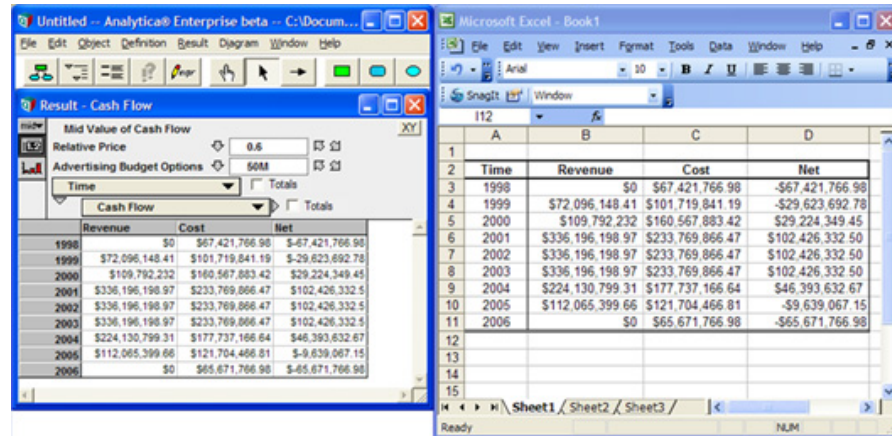
	Revenue	Cost	Net
1998	\$0	\$67,421,766.98	\$-67,421,766.98
1999	\$72,096,148.41	\$101,719,841.19	\$-29,623,692.78
2000	\$109,792,232	\$160,567,883.42	\$29,224,349.45
2001	\$336,196,198.97	\$233,769,866.47	\$102,426,332.5
2002	\$336,196,198.97	\$233,769,866.47	\$102,426,332.5
2003	\$336,196,198.97	\$233,769,866.47	\$102,426,332.5
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	\$65,671,766.98	\$-65,671,766.98

This example itemizes 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 your 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 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.

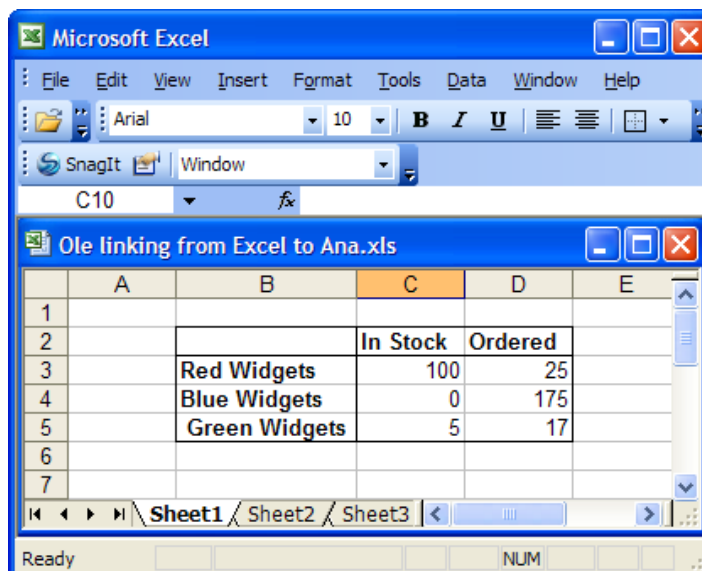
Linking procedure Steps for creating a linked edit table in Analytica with data from an Excel spreadsheet:

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

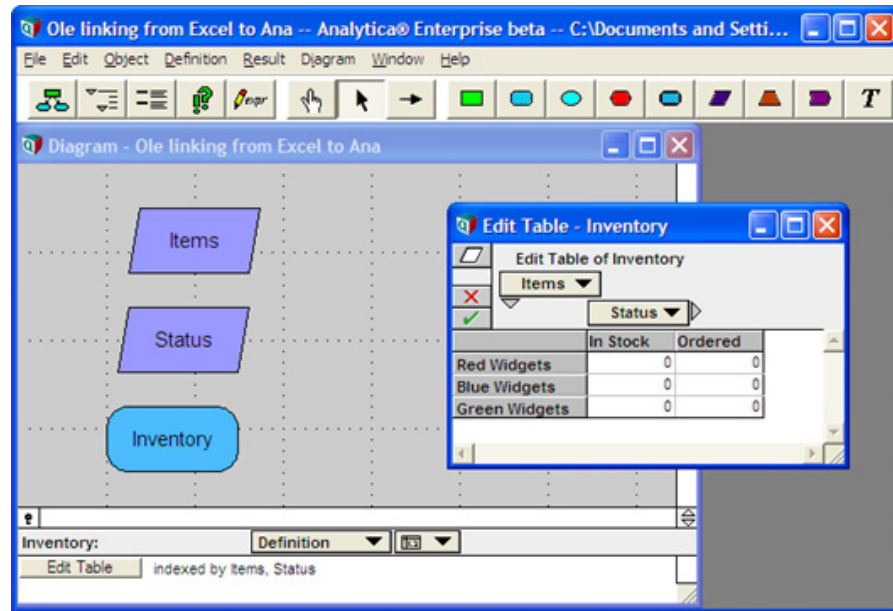
This section itemizes detailed steps for linking a table from Excel into Analytica by creating a node with a "Linked Table" definition. Specifically, suppose you desire to link the Excel table displayed in the following figure into Analytica.



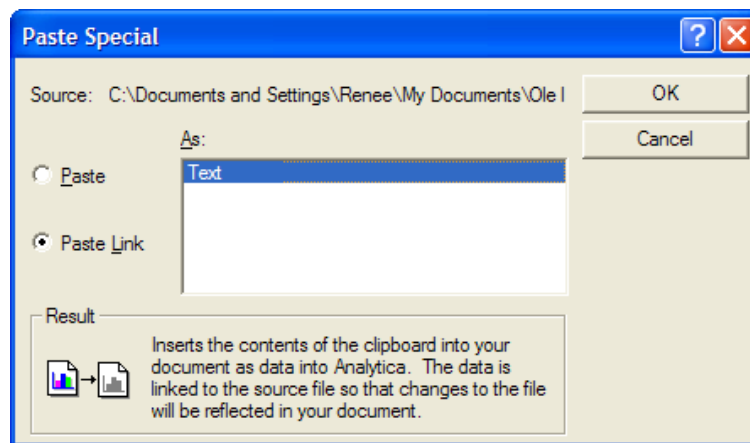
Start by creating two indexes in Analytica to store the row and column headers. Title the first index *Items* and the second *Status*. Select the node *Items* and then click the **Show definition** button on the toolbar (this is the button with the pencil icon) or right mouse menu. In the **Attribute** panel or **Object** window that appears, click 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.

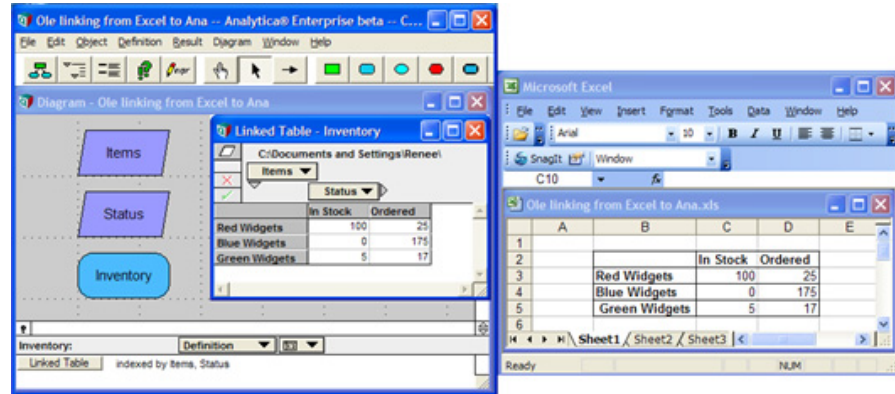
Since you've finished creating the indexes, you're ready to start on the node that will contain the linked table. Create a variable node in Analytica and title it *Inventory*. With this node selected, click the **Show definition** button on the toolbar. In the **Attribute** panel or **Object** window that appears, click 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) and click anywhere in the edit table grid. 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 application windows so that you can see the source table in Excel and the linked table in Analytica, you can readily demonstrate that the link is activated. Change the value for *Green Widgets Ordered* from 2 to say 17. The corresponding value in Analytica's linked table will change accordingly.



Tip 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 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.

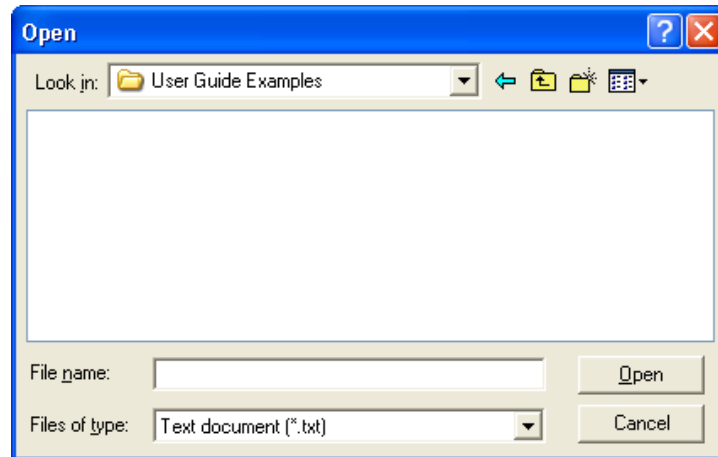
Activating the other application

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 319). 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.

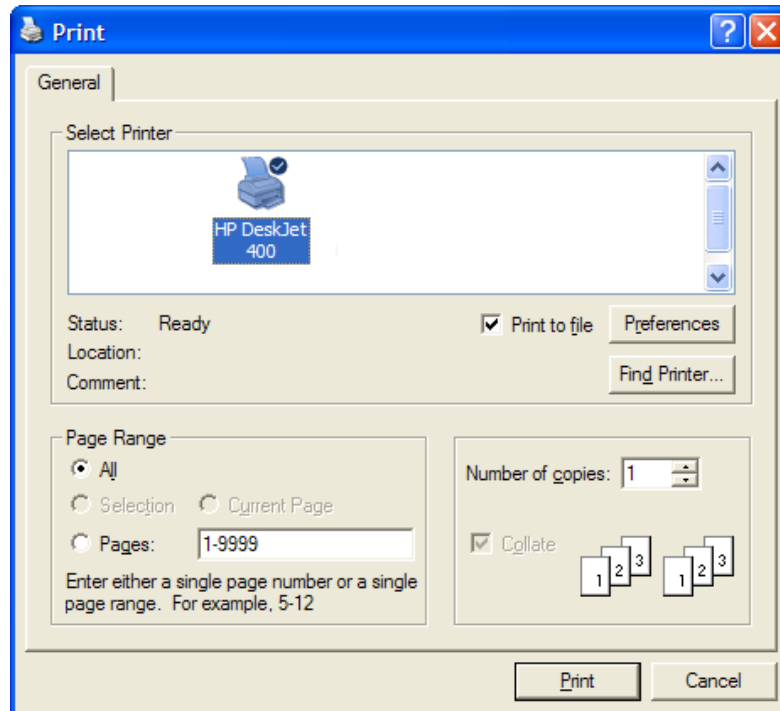
Exporting To export a variable’s definition or result table to a text file, first be certain that the text file is closed.

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.

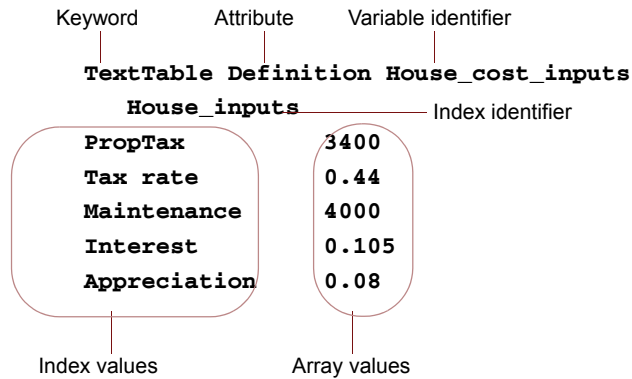
- **TextTable** is a keyword.
- **Attribute** is the name of the attribute into which the data is to be pasted (usually definition).
- **Variable identifier** is the identifier of the variable node into which the data is to be pasted.
- **Index identifier** is the identifier of the index for this variable. 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

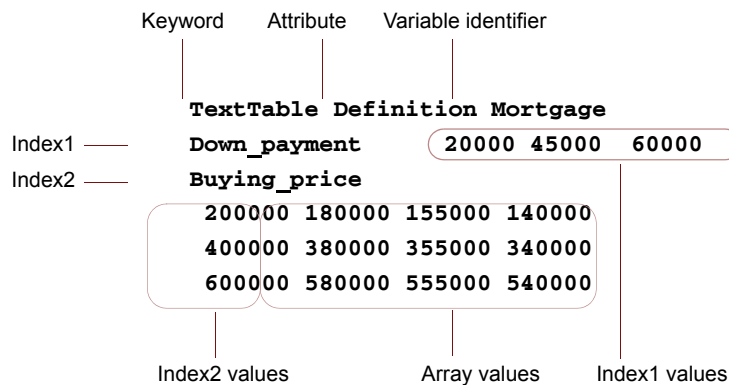


Two-dimensional array

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

	Keyword	Attribute	Variable identifier	
	TextTable	Definition	Net_diff	
Index1	Buying_price	200000		Index1 Value1
Index2	Years_owned	5 10 15		Index2 values
Index3	Down_payment			
	20000	10112	12160	13525
Index3 values	45000	10093	12158	13540
	60000	10073	12157	13555
	Buying_price	400000		
Index1	Years_owned	5 10 15		Index1 Value2
	Down_payment			
	20000	10180.	14201.	16867.
	45000	10160.	14199.	16882.
	65000	10141.	14198.	16897.
	Buying_price	60000		
Index1	Years owned	5 10 15		Index1 Value3
	Down_payment			
	20000	10248	16242	20209
	45000	10228	16241	20224
	60000	10208	16239	20239

Number format

Numerical data can be imported in any format recognized by Analytica (see "Number formats").

Numerical data will be exported in the format set for the table, with these exceptions:

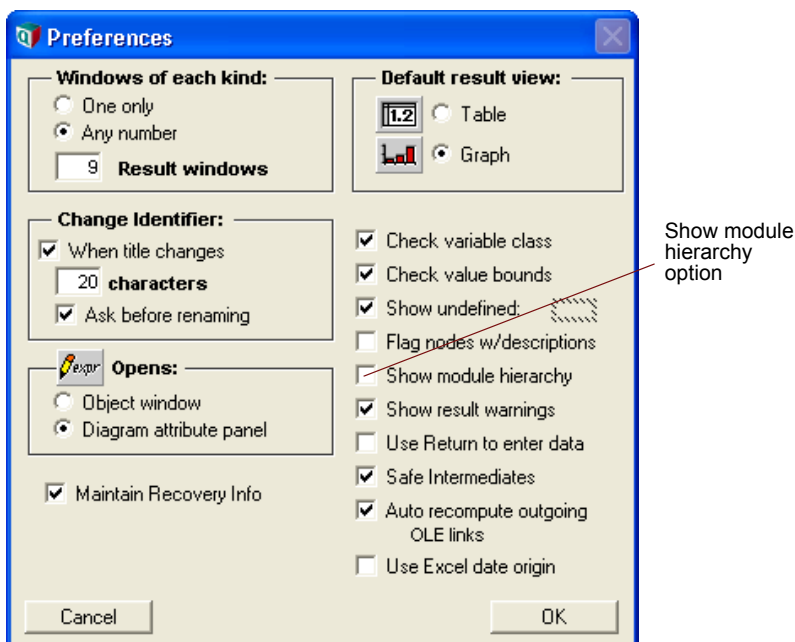
- Suffix format numbers will be exported in scientific exponential format.
- Fixed decimal point numbers of more than 9 digits will be exported in scientific exponential format.
- If a date format begins with the day of the week, e.g., "Saturday, January 1, 2000", the weekday is suppressed: "January 1, 2000".

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.



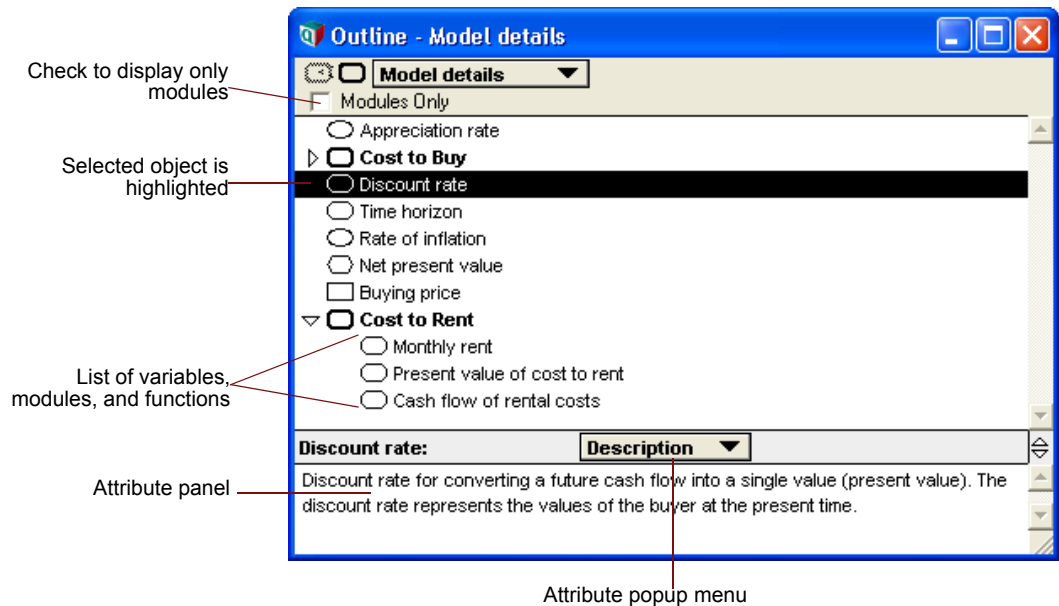
If you check the *Show module hierarchy* box, the top of the active **Diagram** window displays one or more module node shapes to indicate its hierarchy depth.




Indicates that this module has a parent in the model

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 the **Outline** button in the toolbar .

The **Outline** window highlights the entry for the selected module or variable.

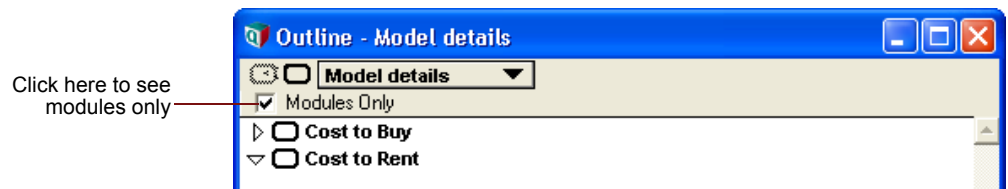
Opening details from an outline

To display a module's **Diagram** window, double-click its entry in the outline.

To display a variable's **Object** window, double-click 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 (\triangleright or \triangleleft) to let you display or hide the module's contents.

- \triangleright Indicates that the module's contents *are not* shown in the **Outline** window. Click this icon to display the module's contents.
- \triangleleft Indicates that the module's contents are shown as an indented list. Click this icon to hide the module's contents.

Viewing and editing attributes

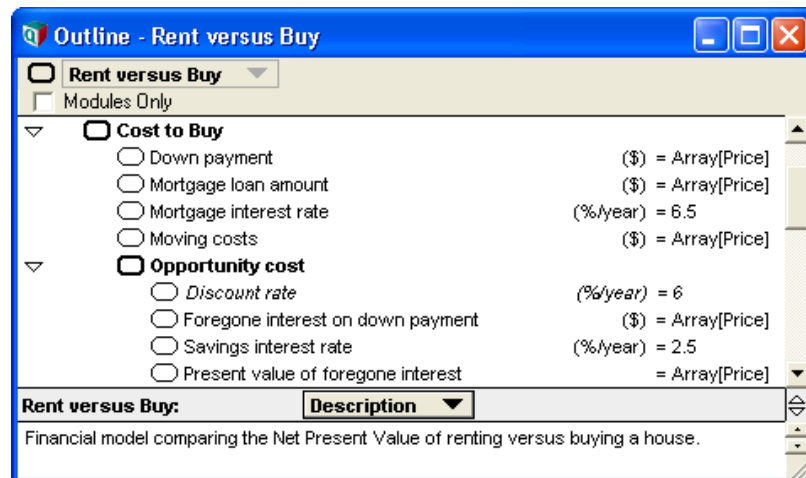
The **Attribute** panel at the bottom of the **Outline** window works just like the **Attribute** panel available at the bottom of a **Diagram** window (see "Diagram window" on page 17).

To view the attributes of a listed node:

1. Select the node by clicking it.
2. Choose the attribute to examine from the **Attribute** popup menu (see “Creating or editing a definition” on page 116).

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 an atom will display in the window (see “Showing values in the Object window” on page 24).

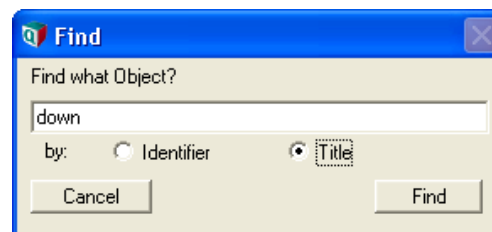


Finding variables

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** (*Control+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 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** (*Control+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** (*Control+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	*	*	*
<i>Created</i>		*	
Definition	*		*
Description	*	*	*
Domain			÷
<i>File Info</i>		*	
Help	÷	÷	÷
Identifier	*	*	*
<i>Inputs</i>	÷		÷
<i>Last Saved</i>		*	
<i>Outputs</i>	÷		÷
Parameters	*		
<i>Probvalue</i>			÷
Title	*	*	*
Units	*		*
<i>Value</i>			÷
User-created (up to 5)	÷	÷	÷

Key:

plain = modifiable by user

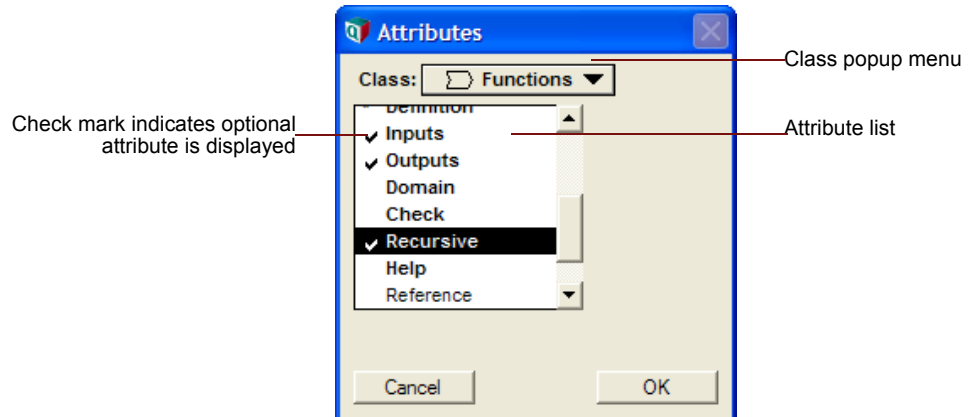
* = always displayed

italic = set by Analytica √ = optionally displayed

For descriptions of the attributes, see “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 displayed optionally.

Displaying optional attributes To display an optional attribute in the **Object** window and **Attribute** panel, click it once to select it, then click it again to show a check mark.

To hide an optional attribute, click it once to select it, then click 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 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 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 *Probvalue*, 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
```

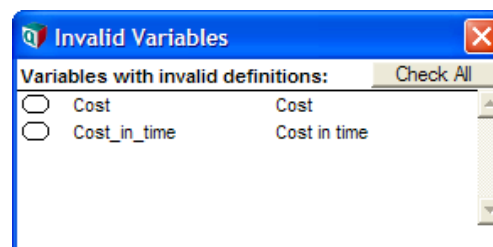
Library Special


Example Units of Time → 'Years'

Tip 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 a variable to open its **Object** window. From the **Object** window, you can edit the definition, or click 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 20, “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

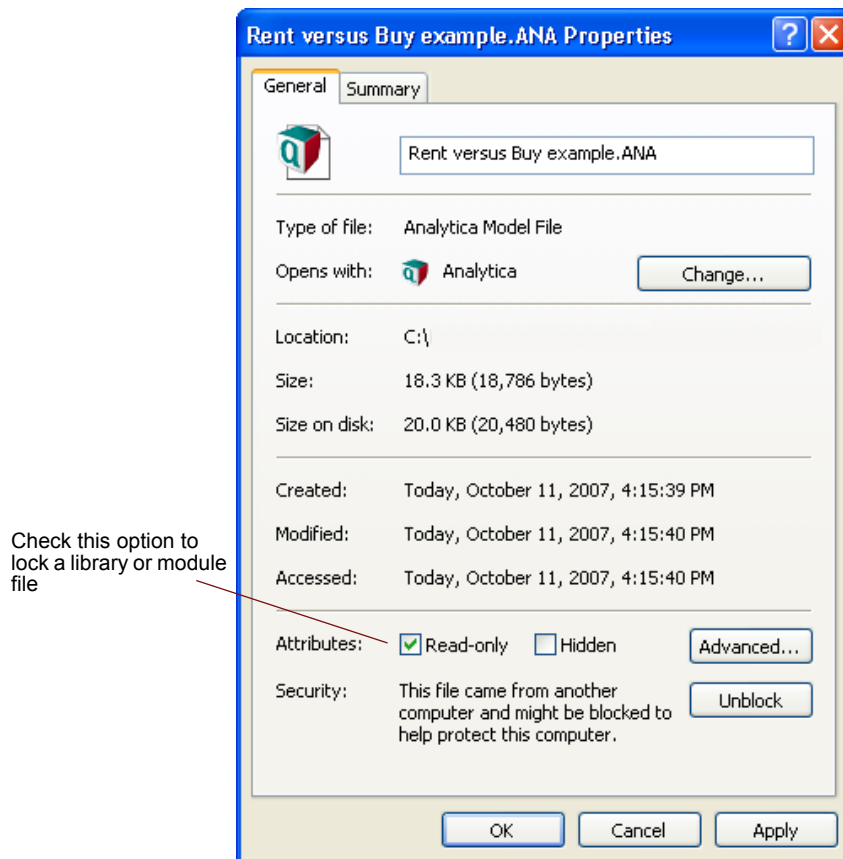
To create a filed module or library:

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 “To change the class of an object” on page 58).
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”). 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”). 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.

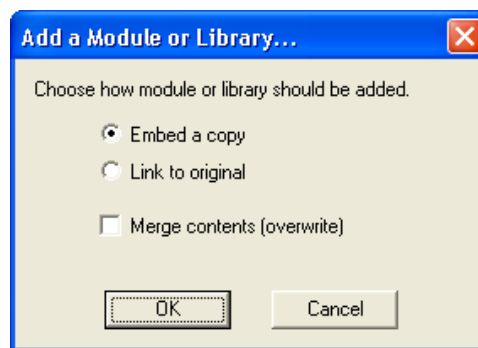
The **Save As** and **Save A Copy In** commands save only the active (topmost window's) model, filed module or filed library.

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 (*Control+I*). 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:



Tip 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”) 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 variable 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”.

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”), 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”) 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”) 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)”.

Created attributes When you combine models with created attributes, the maximum number of defined attributes is five (see “Managing attributes”).

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

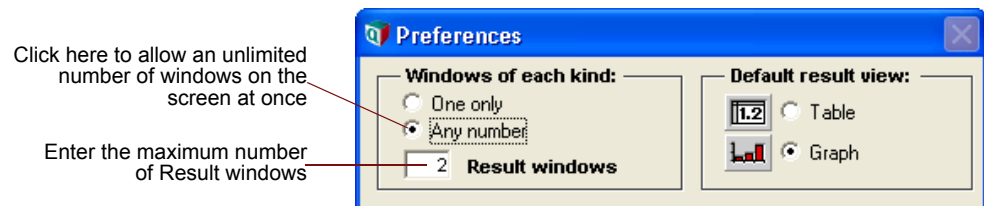
- Two **Result** windows

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 (*Control*) while you click or double-click to open the new window.
- Use the **Preferences** dialog box (see “Preferences dialog”) 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 $\mathbf{A}[\mathbf{I}=\mathbf{J}]$, or equivalently $\mathbf{subscript}(\mathbf{A}, \mathbf{I}, \mathbf{J})$, is now approximately linear in the size of \mathbf{J} , rather than proportional to the product of the sizes of \mathbf{I} and \mathbf{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”.

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**.

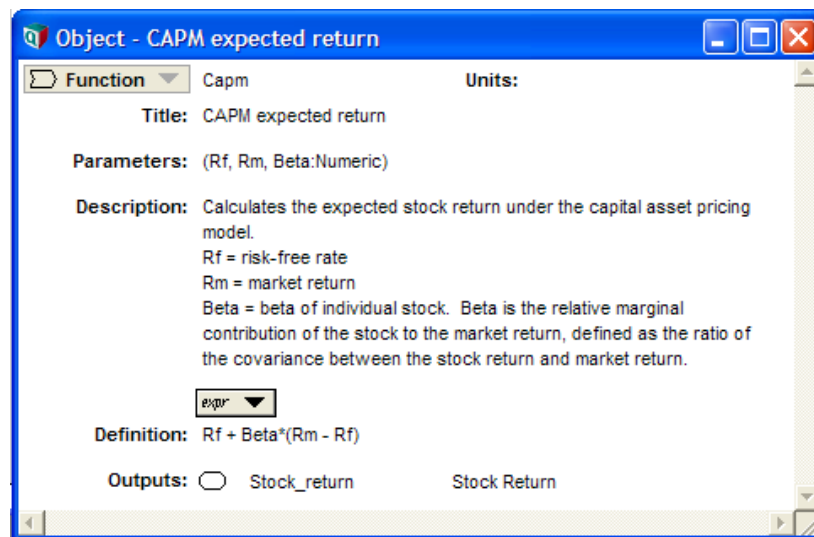
A **library** is a collection of functions grouped in a library file to extend Analytica's built-in functions for use for particular types of application. more than one model. Analytica is distributed with an initial set of libraries, available in the **Libraries** folder inside the Analytica folder 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 numbers.

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 parameters and some are optional. See "Optional parameters" for how to qualify parameters as optional.

You can mix positional and named parameters, as in:

```
Ful(1, 2, D:4, C:3)
```

But, you may not list a positional parameter after a named parameter:

```
Ful(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.

Tip 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.
2. Drag the **Function node** icon from the node palette into the diagram area.

3. Title the node, and double-click 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.
- 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 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: Number Array[I,J]; I, J: Index; C; D: Optional Atom Text)
```

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, **Index**. **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**, **Atom**, 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.

ContextSample Causes the qualified parameter to be evaluated in prob mode if any of the other parameters to the function are **Run**. If not, it evaluates in context mode — i.e., prob or mid following the context in which the function is called.

This qualifier is used for the main parameter of most built-in statistical functions. For example, **Mean** has these parameters:

```
Mean(x: ContextSample[i]; i: Index = Run)
```

Thus, **Mean(X, Run)** evaluates **x** in prob mode. So does **Mean(X)**, because the index **i** defaults to **Run**. But, **Mean(X, J)** evaluates **x** in mid mode, because **J** is not **Run**.

When the parameter declaration contains more than one dimension, prob mode is used if *any* of the indexes is **Run**.

Mid Evaluates the parameter deterministically, or in mid mode, using the mid (usually median) of any explicit probability distribution.

Prob 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. For example:

```
( A : Prob [ In1, Run ] )
```

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. For example:

```
( A : Sample[ In1, Run ] )
```

Index 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.

Variable 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`.

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 `Number Atom`.

Atom 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 `Atom` to qualify parameters of every function, just in case. Functions with `Atom` 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.

Array 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: Array[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:

```
Array [ In1, In2, ... ]
```

Zero or more indexes can be specified between the square brackets. A zero-index declaration means that the value will be an atom when the function body is evaluated, and in this case the keyword `Atom` may be used. `Array` is also optional (e.g., one could write `Number [In1]` rather than `Number Array [In1]`).

Each index identifier listed inside the brackets may be either a global index variable or another parameter explicitly qualified as an `Index`. For example the `Parameters` attribute:

```
(A: [Time, J]; J: Index)
```


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 accepts 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: 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:

Number	A number, including +INF , -INF , or NaN .
Positive	A number greater than zero, or INF .
Nonnegative	Zero, or a number greater than zero or INF .
Text	A text value.
Reference	A reference to a value, created with the \ operator.
Unevaluated	

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 Text [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 an atom (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 [I]
```

Optional parameters

Any parameter may be optional if declared as optional by including the keyword `Optional`, 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 a middle parameter is 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 variable 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 variable and Index parameters. Some built-in functions may crash if passed an unspecified `Index` or `Variable` 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.
```



Deprecated synonyms for parameter qualifiers

Most parameter qualifiers have several synonyms. For example, Atomic, AtomType, and AtomicType are synonyms for Atom. We recommend that you use only the words listed above. If you encounter other synonyms in older models, consult the Analytica wiki "Deprecated qualifiers" to see what they mean (<http://lumina.com/wiki/>).

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 "To change the class of an object" on page 58):

- 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 "To change the class of an object" on page 58).
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.

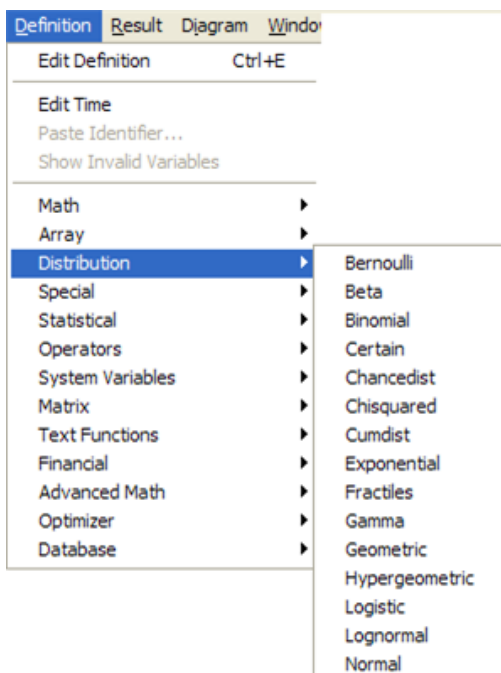
Adding a filed library to a model

Add a filed library to a model using the **Add Module** dialog box (see "Adding a module or library").

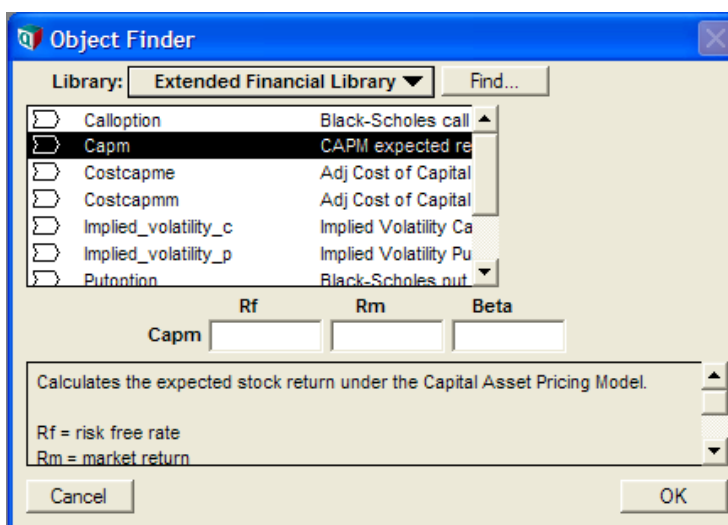
Using a library

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.



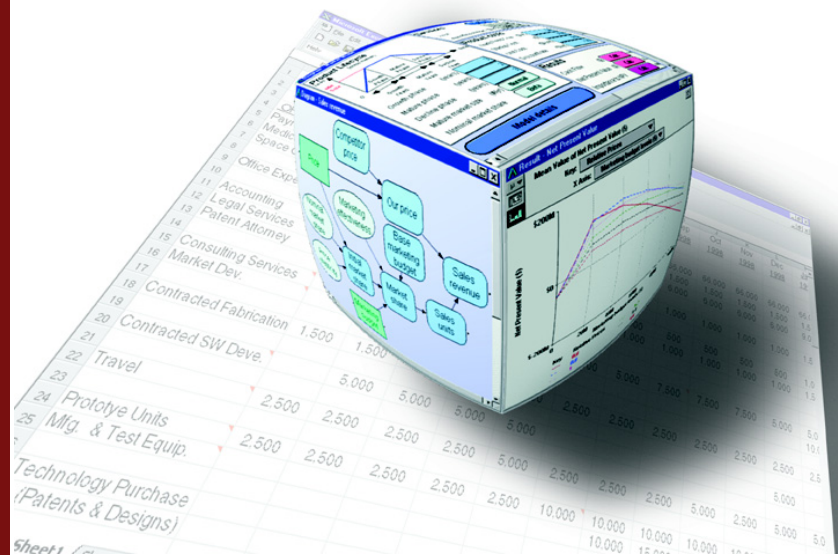
Example Compare the way the `capm()` function is displayed in the **Object** window (page 345) to the way it is displayed in the **Object Finder**:



Chapter 21

Procedural Programming

This chapter shows you how to use the procedural features of the Analytica modeling language.



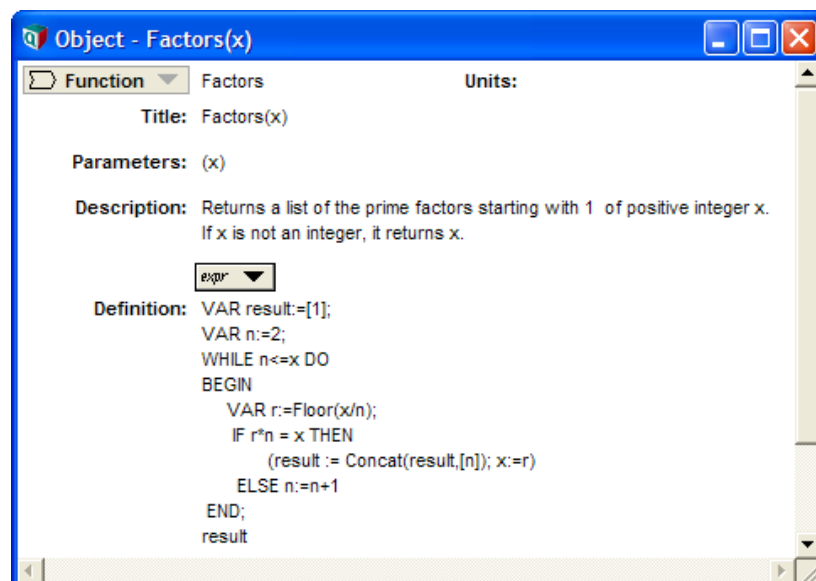
A **procedural program** is sequence of instructions to a computer. Each instruction tells the computer what to do, or in what sequence to execute the instructions. Most Analytica models are **non-procedural** — that is, they consist of an unsequenced set of definitions of variables. Each definition is a simple expression that contain functions, operators, constants, and other variables, but no procedural constructs controlling the sequence of execution. In this way, Analytica is like a 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 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 features:

1. **VAR x := e** construct defines a local variable **x**, and sets an initial value **e**. See page 351 for more.
2. You can group several expressions (statements) into a definition by separating them by ';' (semicolons). Expressions can be on the same line or successive lines. See page 350.
3. **While test Do body** construct tests condition **test**, and, if True, evaluates **body**, and repeats until condition **test** is False. See page 355.
4. **Begin e1 ; e2; ... End** groups several expressions separated by ';'s — in this case as the body of a While loop. See page 350.
5. **(e1 ; e2; ...)** is another way to group expressions — in this case, as the action to be taken in the Then case. See page 350.
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 351.
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 350
BEGIN e1 ; e2; ... ei END	A group of expressions to be evaluated in sequence.	page 350
(e1 ; e2; ... ei)	Another way to group expressions.	page 350
m .. n	Generates a list of successive integers from m to n.	page 360
Var x := e	Define local variable x and assign initial value e .	page 351
Index i := e	Define local index i and assign initial value e .	page 358
x := e	Assigns value from evaluating e to local variable x . Returns value e .	page 351
While Test Do Body	While Test is True, evaluate Body and repeat. Returns last value of Body .	page 355
{ comments } /* comments */	Curly brackets { } and /* */ are alternative ways to enclose comments to be ignored by the parser.	page 349
'text' "text"	You may use single or double quotes to enclose a literal text value, but they must match.	page 142
For x := a DO e	Assigns to loop variable x , successive atoms from array a and repeats evaluation expression e for each value of x . Returns an array of values of e with the same indexes as a .	page 363
For x[i, j...] := a DO e	Same, but it assigns to x successive subarrays of a , each indexed by the indices, [i, j ...] .	page 363
\ e	Creates a reference to the value of expression e .	page 364
\ [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 366
# r	Returns the value referred to by reference r .	page 364

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.

Declaring local variables and assigning to them

Defining a local variable: Var v := e

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 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.

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 278) when data is dimensioned by an index other than the system index *Run*.

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 primary 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 parameter. This will change only the parameter and will not affect any global variables used as actual parameters in the call to the function.

Tip Usually, *you cannot assign to a global variable* — that is, to a variable created as a diagram node. You can assign only to a local variable, declared in this definition using **Var** or **Index**, in the **current context** — that is, at the same or enclosing level in this definition. In a function definition, you may also assign to a parameter. This prevents **side effects** — i.e., where evaluating a global variable or function changes a global variable, other than one that mentions this variable or function in its definition. Analytica's lack of side effects makes models *much* easier to write, understand, and debug than normal computer languages that allow side effects: You can tell how a variable is computed just by looking at its definition, without having to worry about parts of the model not mentioned in the definition. There is an exception to this rule of no assignments to globals: You may assign to globals in button scripts or functions called from button scripts. See "Creating buttons and scripts" for details.

Assigning to a slice of a local variable

Slice assignment means assigning a value into an element or slice of an array contained by a local variable, for example:

```
x[I = n] := e
```

x must be a local variable, **i** is an index (local or global), **n** is a *single* value of **i**, and **e** is any expression. If **x** was not array or was an array not indexed by **i**, the slice assignment adds **i** as a dimension of **x**.

You can write some algorithms much more easily and efficiently using slice assignment. For example:

```
Function Fibonacci_series(f1, f2, n: Number Atom) :=
  INDEX m := 1..n;
  VAR result := 0;
  result[m = 1] := f1;
  result[m = 2] := f2;
  FOR I := 3..n DO result[m = I] := result[m = I - 1] + result[m = I - 2];
  result
```

In the first slice assignment:

```
result[m = 1] := f1;
```

result was not previously indexed by **m**. So the assignment adds the index **m** to **result**, making it into an array with value **f1** for **m=1** and its original value, 0, for all other values of **m**.

More generally, in a slice assignment:

```
x[I = n] := e
```

If \mathbf{x} was already indexed by \mathbf{i} , it sets $\mathbf{x}[\mathbf{i}=\mathbf{n}]$ to the value of \mathbf{e} . For other values of \mathbf{i} , \mathbf{v} retains its previous value. If \mathbf{x} was not already indexed by \mathbf{i} , the assignment adds \mathbf{i} as a dimension of \mathbf{x} , and sets the slice $\mathbf{x}[\mathbf{i}=\mathbf{n}]$ to \mathbf{e} . All other slices of \mathbf{x} over \mathbf{i} retain their previous values. If \mathbf{x} was indexed by other indexes, say \mathbf{j} , the result is indexed by \mathbf{i} and \mathbf{j} . The assigned slice $\mathbf{x}[\mathbf{i}=\mathbf{n}]$ has the value \mathbf{e} for all values of the other index(es) \mathbf{j} . Again, slices for other values of \mathbf{i} retain their original values of \mathbf{x} .

You may index by position as well as name in a slice assignment, for example:

```
 $\mathbf{x}[\text{@}\mathbf{i} = 2] := \mathbf{e}$ 
```

assigns the value of \mathbf{e} as the second slice of \mathbf{x} over index \mathbf{i} .

Slice assignment, e.g., $\mathbf{x}[\mathbf{i} = \mathbf{A}] := \mathbf{e}$, has three limitations:

- \mathbf{x} must be a local variable.
- \mathbf{n} must be an atom, not an array.
- You may use only one index. For example, you may not use an expression like $\mathbf{x}[\mathbf{i} = \mathbf{A}, \mathbf{j}=\mathbf{B}] := \mathbf{e}$, with two index expressions. If \mathbf{x} has two (or more) dimensions, you can create and assign a slice (e.g., a row) to \mathbf{x} .

For and While loops and recursion

Tip Analytica's Intelligent Array features means that you rarely need explicit iteration using FOR loops to repeat operations over each dimensions of an array, often used in conventional computer language. If you find yourself using FOR loops a lot in Analytica, this may be a sign that you are not using the Intelligent Arrays effectively. If so, please (re)read the section on Intelligent Arrays (see).

For $i := a$ Do $expr$

The **For** loop successively assigns the next atom from array \mathbf{a} to local index \mathbf{i} , and evaluates expression \mathbf{expr} . \mathbf{expr} may refer to \mathbf{i} , for example to slice out a particular element of an array. \mathbf{a} may be a list of values defined by $\mathbf{m}..n$ or **Sequence**(\mathbf{m} , \mathbf{n} , \mathbf{dx}) or it may be a multidimensional array. Normally, it evaluates the body \mathbf{expr} once for each atom in \mathbf{a} .

The result of the **For** is an array with all the indexes of \mathbf{a} containing the values of each evaluation of \mathbf{expr} . If any or all evaluations of \mathbf{expr} have any additional index(es), they will also be indexes of the result.

Usually, the Intelligent Array features take care of iterating over indexes of arrays without the need for explicit looping. **For** is sometimes useful in these specialized cases:

- To avoid selected evaluations of \mathbf{expr} that may be invalid or out of range, and can be prevented by nesting an **if-Then-Else** inside a **For**.
- To apply an Analytica function that requires an atom 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.

See below for an example of each of these three cases.

Library Special

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. To 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 an atom, 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 **Atom**, see page 340). 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, SplitText(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 $200 \times 300 \times 10 = 600$ kilobytes of memory at a time.

For the special case of a dot product (see “Dot product of two matrices”), for an expression of the form `Sum(A*B, I)`, it performs a similar transformation internally.

While (Test) Do Body

While evaluates **Body** repeatedly as long as **Test** $\neq 0$. 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, **While** will continue to loop indefi-

nately. If you suspect that may be happening, type *Control+.* (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 361 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.

Tip You can usually write convergence algorithms more cleanly using **While**. One difference is that **While** requires its stopping condition **Test** to be an atom, 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 361 to ensure the expression fully supports array abstraction.

Recursive functions

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:

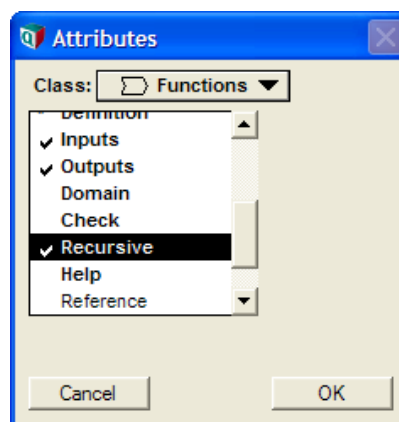
```
Function Factorial2(n: Positive Atom)
Definition: IF n > 1 THEN N*Factorial2(n-1) ELSE 1
```

If its parameter, **n**, is greater than 1, **Factorial2** calls itself if with the actual parameter value **n-1**. Otherwise, it simply returns 1. Like any normal recursive function, it has a termination condition under which the recursion stops — when **n** <= 1.

Tip The built-in function **Factorial** does the same, and is fully abstractable, to boot. We define **Factorial2** here as a simple example to demonstrate key ideas.

Normally, if you try to use a function in its own definition, it will complain about a cyclic dependency loop. To enable recursion, you must display and set the **Recursive** attribute:

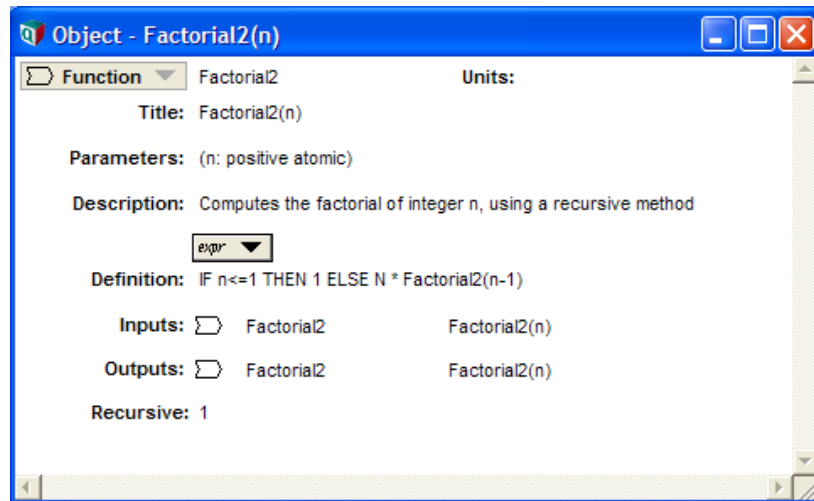
1. Select the **Attributes** dialog from the **Object** menu:



2. Select **Functions** from the **Class** menu in this dialog.
3. Scroll down the list of attributes and click **Recursive** *twice*, so that it shows , meaning that the recursive attribute will be displayed for each function in its **Object** window and the **Attribute** panel.
4. Check **OK** to close **Attributes** dialog.

For each function for which you wish to enable recursion:

5. Open the **Object Window** for the function by double-clicking its node (or select the node and click the **Object** button)
6. Type **1** into its **Recursive** field, thus:



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 Prime_factors(x, y: Positive Atom)
Definition:
  Var n := Floor(x/y);
  IF n < y THEN [x]
  ELSE IF x = n*y THEN Concat([y], Factors(n, y))
  ELSE Prime_factors(x, y+1)
```

`Factors(60, 2) → [2, 2, 3, 5]`

In essence, `Prime_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.

Tip To prevent accidental infinite recursion, it will stop and give a warning if the stack reaches a depth of 256 function calls.

Local indexes

You can declare a local index in the definition of a variable or function. It is possible that the value of the variable or value returned by the function is an array using this index. This is handy because it lets you define a variable or function that creates an array without relying on an externally defined index.

The construct, `Index i := indexExpr` defines an index local to the definition in which it is used. The expression `indexExpr` 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 := 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 →

0	1
1	2
2	4
3	8
4	16
5	32

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 := 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., $\mathbf{A} \times \mathbf{A}'$, where \mathbf{A}' is the transpose of \mathbf{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: Array; i, j: Index)  
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 := Var XX := MatSqr(X, Rows, Cols);  
Sum(XX * Y[I=XX.I2], XX.I2)
```

Ensuring array abstraction

The vast majority of the elements of the Analytica language (operators, functions, and control constructs) fully support Intelligent Arrays — that is, they can handle operands or parameters that are arrays with any number of indexes, and generate a result with

the appropriate dimensions. Thus, most models automatically obtain the benefits of array abstraction with no special care.

There are just a few elements that do *not* inherently enable Intelligent Arrays — i.e., support *array abstraction*. They fall into these main types:

- Functions whose parameters must be atoms (not arrays), including **Sequence**, **m..n**, **SplitText**. See page 360.
- Functions whose parameter must be a vector (an array with just one index), such as **CopyIndex**, **SortIndex**, **Subset**, **Unique**, and **Concat** when called with two parameters.
- The **While** loop, which requires its termination condition to be an atom.
- **If b Then c Else d**, when condition **b** is an array, and **c** or **d** may give an evaluation error.
- Functions with an optional index parameter that is omitted, such as **Sum(x)**, **Product**, **Max**, **Min**, **Average**, **Argmax**, **SubIndex**, **ChanceDist**, **CumDist**, and **ProbDist**. See page 363.

When using these constructs, you must take special care to ensure that your model is fully array-abstractable. Here we explain how to do this for each of these five types.

Functions expecting atomic parameters

Consider this example:

```
Variable N := 1..3
Variable B := 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)`, cannot work if `N` is an array, because it would have to create a nonrectangular array containing slices with 1, 2, and 3 elements. Analytica does not allow nonrectangular arrays, and so requires the parameters of **Sequence** to be atoms (single elements).

Most functions and expressions that, like **Sequence**, are used to generate the definition of an index require atomic (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 Factorial2
Parameters: (n)
Definition: Product(1..n)
```

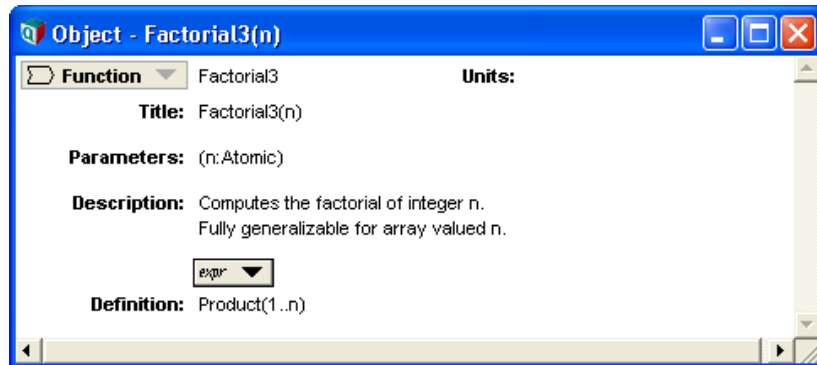
It works if `n` is an atom, but not if it is an array, because `1..n` requires atom operands. However, in this version, using a **For** loop, will work fine:

```
Function Factorial3
Parameters: (n)
Definition: FOR m := n DO Product(1..m)
```

The **For** loop repeats with the loop variable `m` set to each atom of `n`, and evaluates the body `Product(1..m)` for each value. Because `m` is guaranteed to be an atom, this works fine. The **For** loop reassembles the result of each evaluation of `Product(1..m)` to create an array with all the same dimensions as `n`.

Atom parameters and array abstraction

```
Function Factorial3
Parameters: (n: Atom)
Definition: Product(1..n)
```



```
Index K := 1 .. 6
```

```
Factorial3(K) →
```

1	2	3	4	5	6
1	2	6	24	120	720

Notice that **Atom** does not require the actual parameter x to be an atom when the function is called. If x is an array, as in this case, it repeatedly evaluates the function **Factorial3(n)** with n set to each atom of array x . It then reassembles the results back into an array with the same indexes as parameter x , like the **For** loop above. This scheme works fine even if you qualify several parameters of the function as **Atom**.

In some cases, a function may require a parameter to be a vector (have only one index), or have multiple dimensions with specified indexes. You may use “Array qualifiers” on page 342 to specify this. With this approach, you can ensure your function will array abstract when new dimensions are added to your model, or if parameters are probabilistic.

While and array abstraction

The **While b Do e** construct requires its termination condition b to evaluate to be an atom — that is, a single Boolean value, True (1) or False (0). Otherwise, it would be ambiguous about whether to continue. Again, **Atom** is useful to ensure that a function using a **While** loop array abstracts, as it was for the **Sequence** function. Here’s a way to write a Factorial function using a While loop:

```
Function Factorial4
Parameters: (n: Atom)
Definition:
  VAR fact := 1; VAR a := 1;
  WHILE a < n DO (a := a + 1; fact := fact * a)
```

In this example, the `Atom` qualifier assures that `n` and hence the `While` termination condition `a < n` is an atom during each evaluation of `Factorial4`.

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]
```

For the construct IF *a* THEN *b* ELSE *c*, *a* is an array of truth values, as in this case, so it evaluates both *b* and *c*. It returns the corresponding elements of *b* or *c*, according to the value of condition *a* for each index value. Thus, it still ends up evaluating `sqr(x)` even for negative values of *x*. In this case, it returns 0 for those values, rather than NAN, and so it generates no error message.

A similar problem remains with text processing functions that require a parameter to be a text value. 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 arises 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.
```

`TextReplace` doesn't like the first value of *z*, which is a number, where it's expecting 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)
```

It still doesn't work because the `IF` construct still applies `ReplaceText` to all elements of *t*. Now, let's add the parameter qualifier `Atom` to *t*:

```
Function RemoveCommas(t)
Parameters: (t: Atom)
Definition: IF IsText(t)
THEN Evaluate(TextReplace(t, ',', '')) ELSE t
RemoveCommas(Z) →
```

1,000	1000
'10,000'	10000
'100,000'	100000

This works fine because the **Atom** qualifier means that **RemoveCommas** breaks its parameter **t** down into atomic elements before evaluating the function. During each evaluation of **RemoveCommas**, **t**, and hence **IsText(t)**, is atomic, 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

Several functions have index parameters that are optional, including **Sum**, **Product**, **Max**, **Min**, **Average**, **Argmax**, **SubIndex**, **ChanceDist**, **CumDist**, and **ProbDist**. For example, with **Sum(x, i)**, you can omit index **i**, and call it as **Sum(x)**. But, if **x** has more than one index, it is hard to predict which index it will sum over. Even if **x** has only one dimension now, you might add other dimensions later, for example for parametric analysis. This ambiguity makes the use of functions with omitted index parameters 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.

Tip

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 atomic) 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 atoms. 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	It assigns to loop variable x successive atoms 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 For x[] := a DO e	It assigns to loop variable x , successive atomic values from array a . It repeats evaluation of expression e for each value. It returns an array of values of e with the same indexes as a .
For x[i] := a DO e	It assigns to loop variable x successive subarrays from array a , each indexed only by i . It repeats evaluation of expression e for each index value of a other than i . As before, the result has the same indexes as a .
For x[i, j ...] := a DO e	It assigns to loop variable x successive subarrays from array a , each indexed only by i, j It repeats evaluation of expression e for each index value of a other than i, j As before, the result has the same indexes as a .

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, an atom 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**.

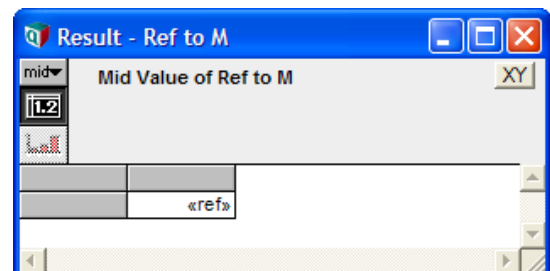
#e is the **dereference operation**. It obtains the value referred to by **e**. If **e** is not a reference, it issues a warning and returns Null.

An example:

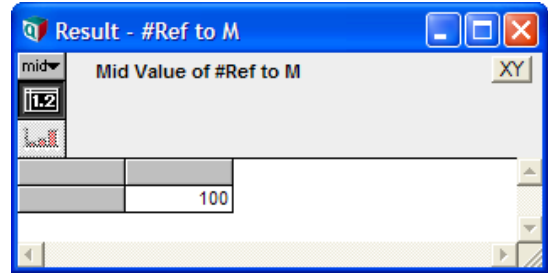
```
Variable M
Definition: 100

Variable Ref_to_M
Definition: \ M
```

The result of **Ref_to_M** looks like this:



You can double-click the cell containing «ref» to view the value referenced, in this case:

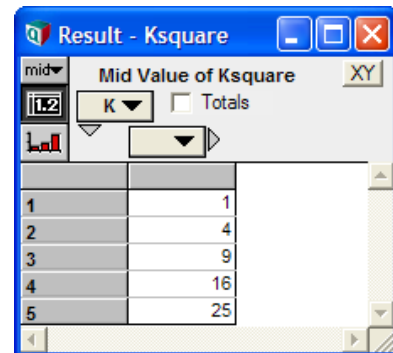


You can also create an array of references. Suppose:

Index K
Definition: 1..5

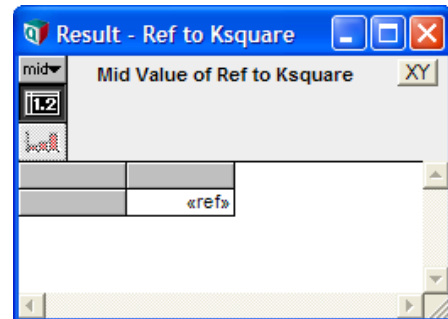
Variable Ksquare
Definition: K^2

Ksquare →

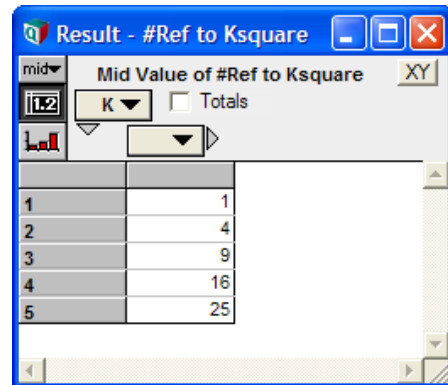


Variable Ref_to_Ksquare
Definition: \ Ksquare

Ref_to_Ksquare →



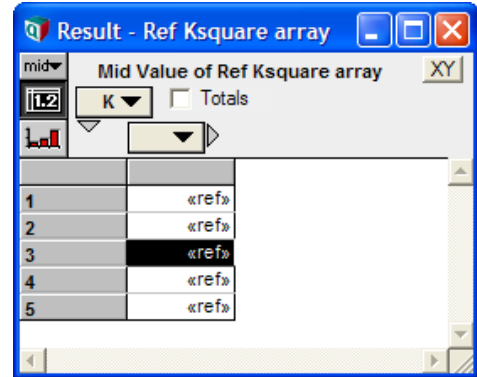
If you click the «ref» cell, it opens:



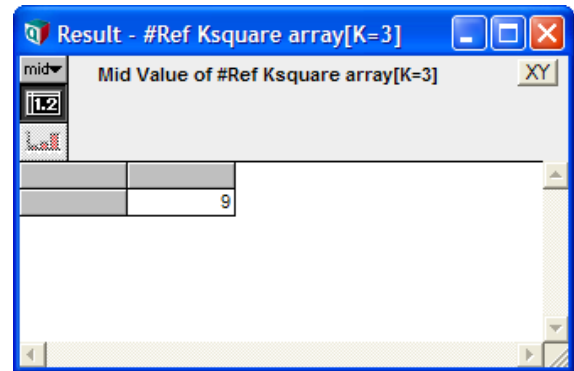
You can also create an array of references from an array, for example:

```
Variable Ref_Ksquare_array
Definition: \ [] Ksquare
Ksquare →
```

The empty square brackets '[]' specify that the values referred to have no indexes, i.e., they are atoms. You can now click any of these cells to see what it refers to.



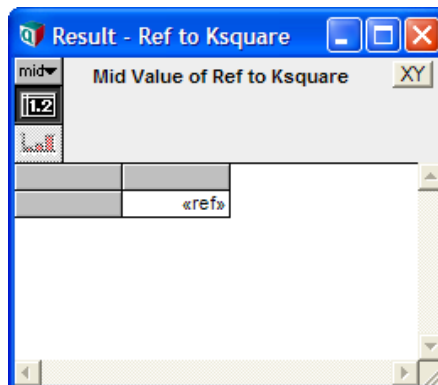
Clicking the third cell, for example, gives:



Managing indexes of referenced subarrays:
 $\ [i, j, \dots] \mathbf{e}$

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 κ was used as an index of the reference array. If instead we write:

```
\ [K] Ksquare →
```



It creates a similar result to $\ \mathbf{Ksquare}$, since κ is the only index of **Ksquare**.

To summarize:

<code>\ e</code>	Creates a reference to the value of expression <i>e</i> , whether it is an atom or an array .
<code>\ [] e</code>	Creates an array indexed by all indexes of <i>e</i> containing references to all atoms from <i>e</i> .
<code>\ [i] e</code>	Creates an array indexed by any indexes of <i>e</i> other than <i>i</i> of references to subarrays of <i>e</i> each indexed by <i>i</i> .
<code>\ [i, j ...] e</code>	Creates an array indexed by any indexes of <i>e</i> other than i, j ... of references to subarrays of <i>e</i> each indexed by <i>i, j ...</i> .

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 an element 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: Atom)
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: Index)

```

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.

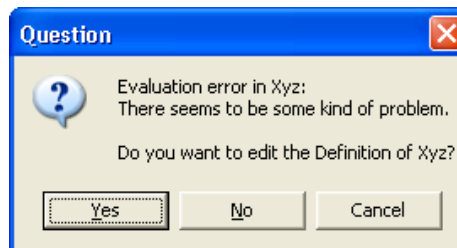
Miscellaneous functions

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 →
```



If you call **Error()** in a check attribute, it will show the error message you supply it for the warning dialog, instead of the default message when the check fails, letting you tailor a message to the application.

Evaluate(*t*)

Evaluates a text value *t* 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, it returns Null; it does 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.

Context of the evaluation

The context in which **b** parses its text parameter **t** is quite different from the definition of the variable that calls **Evaluate**. This creates some subtleties. Consider:

```
Variable A := 99
Variable B := (VAR A := 0; Evaluate('A + 1'))
B → 100
```

Evaluate assumes a global context for evaluating its parameter: Variable **a** in the evaluated text 'A + 1' refers to the global **a**, not the local **a** declared in **b**. More generally:

- **Evaluate(t)** parses the text in **t** at the time it evaluates it, in a context separate from the expression in which the **Evaluate(t)** appears — e.g., the definition of **b** above.
- Thus, text **t** cannot refer to local variables, indexes, or function parameters defined in the context in which **Evaluate(t)** appears.
- Text **t** may itself define and use local variables, but these will not be available outside **t**.
- Automatic dependency maintenance does not work for variables mentioned in evaluated text. For example:

```
B := A+1
C := Evaluate('A+1')
```

When **a** changes, it automatically ensures that **b** is updated when necessary, but does not know that **c** also depends on **a**.

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
```

IgnoreWarnings(expr)

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** in the **Preferences** dialog, because you do want to see warnings that may appear in other parts of the model. For more on that option, see “Preferences dialog” on page 59. For more on warnings, see “Warning” on page 413.

MsgBox(message, buttons, title)

Displays a standard popup model dialog box with the user-supplied **message**, **buttons** (see numerical codes below), and **title** parameters. 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:

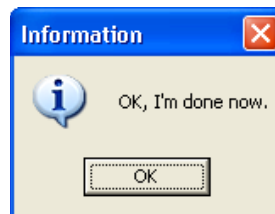
- 16 = Critical (white X on red circle)
- 32 = Question
- 48 = Exclamation
- 64 = Information

MsgBox returns a number depending on which button the user presses:

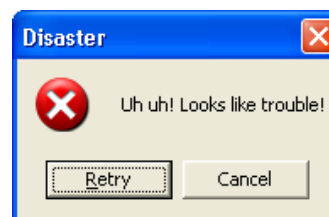
- 1 = OK
- 2 = Cancel (stops any further evaluation)
- 3 = Abort
- 4 = Retry
- 5 = Ignore
- 6 = Yes
- 7 = No

Here are some examples:

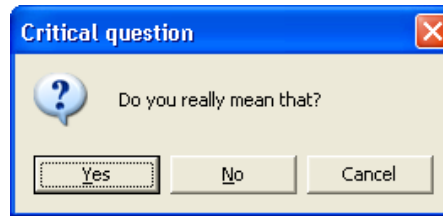
```
Msgbox('OK, I'm done now.',0+64,'Information') →
```



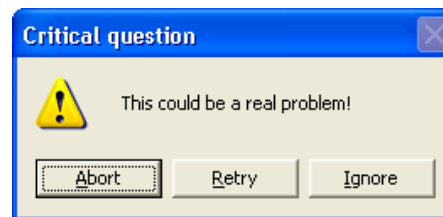
```
Msgbox('Uh uh! Looks like trouble!',5+16, 'Disaster' ) →
```



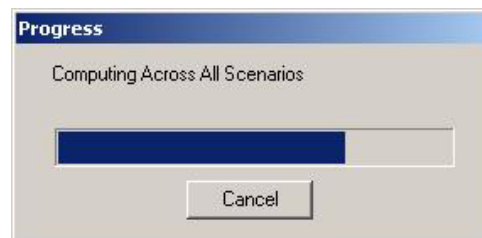
```
Msgbox('Do you really mean that?', 3+32, 'Critical question') →
```



`Msgbox('This could be a real problem!', 2+48, 'Critical question') →`



ShowProgressBar



Declaration `ShowProgressBar(title,text:Text atomic; p:number atomic)`

Description Displays or updates a programmable dialog containing a progress bar. The first time it is called with $p < 1$, the dialog appears. When $0 \leq p < 1$, a **Cancel** button is displayed and the progress meter is updated to the indicated proportion, allowing computation to continue while it is visible. If the user presses **Cancel**, the computation is aborted. When $p = 1$, an **OK** button is shown and the dialog waits until **OK** is pressed to return and then disappears. The dialog is also removed when $p > 1$ or a computation completes.

Example

```
var xOrig := X;
var result :=
  for n[] := @Scenario do (
    ShowProgressBar( "Progress", "Computing Across All Scenarios",
      (n-1)/size(scenario) );
    WhatIf( Y, X, xOrig[@Scenario=n] )
  );
ShowProgressBar( "Progress", "Done", 1 );
result
```

Today()

Returns the current date as the number of days that have elapsed since Jan 1, 1904.

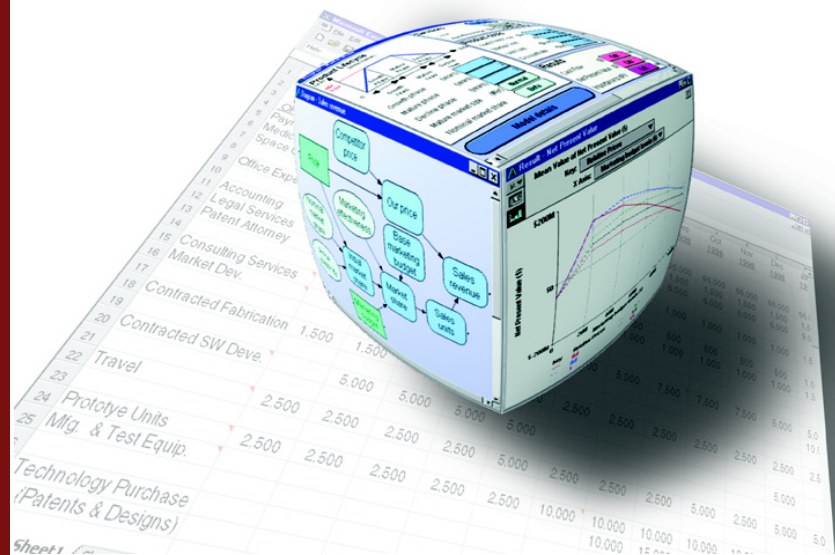
Tip When you evaluate this function, your result probably will be cached. The cached result will become out-of-date if the date changes during the Analytica session.

Chapter 22

Analytica Enterprise

Analytica Enterprise extends the Professional edition with these features:

- **Database access:** Functions to read and write data from and to ODBC databases and external files.
- **Creating buttons:** Objects that users click to run scripts that change the model, and assign new values to variables.
- **Huge arrays:** Expand arrays with indexes of over 30,000 elements, limited only by memory.
- **Save models as Browse-only:** Models that let end users of models modify only variables designated as inputs.
- **Hide definitions:** Prevent end users from viewing data or algorithms that are confidential or proprietary.
- **Performance Profiler:** A library to see which variables and functions take the most CPU time or memory.
- **RunConsoleprocess:** A function that calls another Windows application as subprogram from Analytica.



Tip You need Analytica Enterprise or Optimizer to create models using the features described in this chapter. You can use the Analytica Power Player or the Analytica Decision Engine to run models created with Enterprise or Optimizer with these features, but not to change them. You can use *any* edition of Analytica to run a model that uses buttons, or was saved as Browse-only with Hidden definitions.

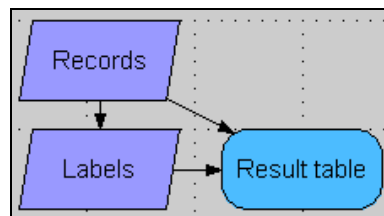
Accessing databases

Analytica Enterprise provides several functions for querying external databases using **ODBC**. ODBC (**O**pen **D**atabase **C**onnectivity) is a widely used standard for connecting to relational databases, on either local or remote computers. It uses queries in Structured Query Language (SQL), pronounced "sequel", to read from and write to databases.

Overview of ODBC SQL is a widely used language to read data from and write data to a relational database. A relational database organizes data in two-dimensional tables, where the **columns** of a table serve as fields or labels, and the **rows** correspond to records, entries, or instances. In Analytica, it is more natural to refer to the columns as **labels** and rows as **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.

The result of an SQL query is a two-dimensional table, called a **result table**. The rows are the records matching the criteria specified by the query. The columns are the requested fields.

Analytica Enterprise provides functions that accept an SQL query, using standard SQL syntax, as a text-valued parameter. These functions return the result of the query as an array with two dimensions: Its 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:



Each of these three nodes 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 they are known in advance; and of course, the **Result_Table** needs to read the table. The Database library provides the functions, **DBQuery**, **DBLabels**, and **DBTable** to define these 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_fields**, as follows:


```

Index Individuals := DBQuery(Data_source, 'SELECT*FROM Addresses')
Index Address_fields := DBLabels(Individuals)
Variable Address_fields := 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 parameters to the **DBQuery()** function, and are discussed in the following two subsections.

DSN and 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, 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.

Tip 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; PWDBMS=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. For example:

```
'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'
```

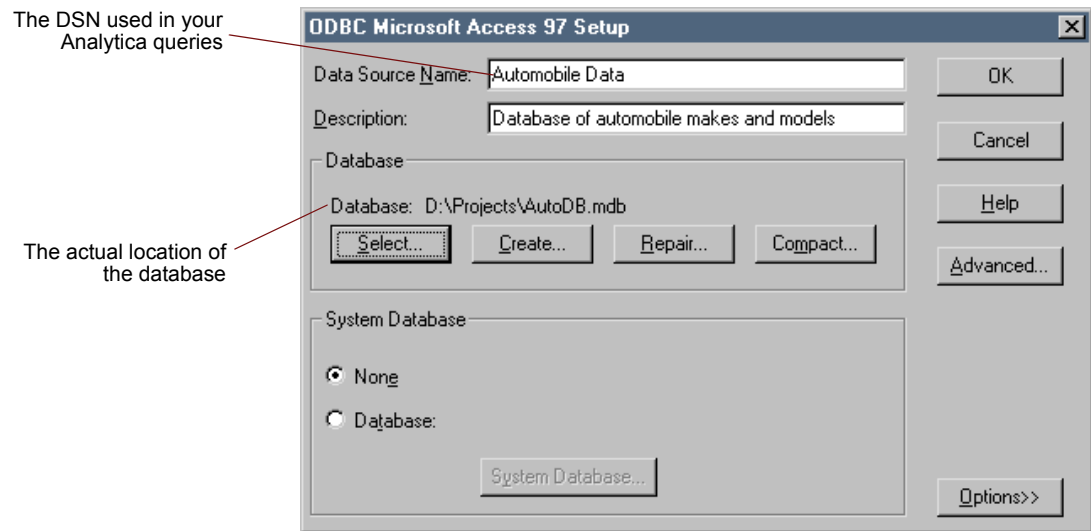
The specific fields used here (UID, PWD, UIDDBMS, PWDBMS, DIRECTORY, etc.) are interpreted by the ODBC driver, and therefore depend on the specific driver used. Any fields interpreted by your driver are allowed.

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 Analytica 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. If you are interested in more demanding applications, please consult one of the many excellent texts on SQL.

The SQL expression to select a complete table in a relational database, where the table is named **VEHICLES**, would be:

```
'SELECT * FROM vehicles'
```

Tip SQL is case insensitive, but Analytica is case sensitive for labels of Column names.

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, for example:

```
'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, for example:

Retrieving an SQL result table

```
'SELECT * FROM myQuery'
```

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'`. Obtain the relational `Result_table` thus:

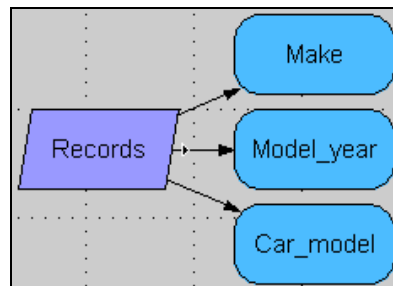
```
Index Records := DBQuery('DSN=Automobile Data',
  'SELECT * FROM vehicles')
Index Labels := DBLabels(Records)
Variable Result_table := DBTable(Records, Labels)
```

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

It is often more convenient for further modeling to create a separate variable for each column of a database table. Each column variable uses the same record index. For example, we might create separate variables for `Make`, `Year`, and `Car model` from the vehicles database table:



In this case, the record index is still 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. For example:

```
Index Records := DBQuery('DSN=Automobile Data',
  'SELECT * FROM vehicles' )
Variable Make := DBTable(Records, 'make')
Variable Model_Year := DBTable(Records, 'year')
Variable Car_Model := DBTable(Records, 'model')
```

Since `Model` is a reserved word in Analytica, we named the variable `Car_Model` instead of just `Model`. But, the second parameter 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.

Alternatively, you can construct a table containing a subset of the columns in a result table. For example, if `vehicles` has a large number of columns, you might create this variable with only the three columns you are interested in:

```
Variable SubCarTable := DBTable(Records, ['make', 'model', 'year'])
```

This table will be indexed by `Records` and by an implicit index (a.k.a. a null 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 an Analytica model. Using the appropriate SQL statements, you can add or delete records from an existing database table. You can also add columns, and create or delete tables, if your data source driver supports these operations.

`DBQuery()` cannot alter the data source, because it processes the SQL statement in read-only mode. Instead, use `DBWrite()`, which is identical to `DBQuery()` except that it processes the SQL statement in read-write mode. `DBWrite()` can make any change to the database that can be expressed as an SQL statement, and is supported by the ODBC driver.

To send data from your model into the database, you must convert that data into a text value — more precisely, into an SQL statement. Analytica offers some tools to help this process. Here, we will illustrate a common case — writing a multi-dimensional array to a table in a database. We use the `ODBC_Library.ana` library distributed with Analytica.

Suppose you want to write the value of variable `A`, which is a three dimensional array indexed by `I`, `J`, and `K`, into a relational table named `TableA`, so that other applications can use the data. 2-D table

First, we need to convert the 3D array into the correct relational table form. Then we convert the table into the SQL text to write to the database.

Our approach is to first convert the three-dimensional array `A` into a two dimensional table, which we store into `TableA`. `TableA` needs two indexes: `ARowIndex` and `ALabelIndex`. These three variables are defined as follows:

```
Index ALabelIndex := Concat(IndexNames(A), ['A'])
Index ARowIndex := sequence(1, Size(A))
Variable TableA := MDArrayToTable(A, ARowIndex, ALabelIndex)
```

`MDArrayToTable(a, i, l)` is described on page 197. `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.

Tip If you want to use column labels in the database that are different from the Analytica index names: Define `ALabelIndex` to be a 1-D array, self indexed. Set the Domain of `ALabelIndex` to be the database labels, and the values of the array to the index names. (The last value is arbitrary).

Our data is now in the form of a 2-D table as needed for a database table. Next we construct the SQL text to write the table to the database. You must choose whether you want to append rows to the existing database table, or replace the table entirely. Or you can replace only selected entries. Your choice affects how you construct the SQL statement. Here, we totally replace any existing data with the new data: After the operation, the database table will be exactly the same as `TableA` in the Analytica model. The SQL statements for performing the write is:

```
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 it. We follow this by 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 will add quotes around them automatically.

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,

```
'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).

Finally, define:

```
Variable Write_A_to_DB := DBWrite(DB, WriteTableSql(A, RowIndex,
LabelIndex, 'TableA'))
```

Creating an output node to write to a database

`write_A_to_DB` writes array `A` to the database whenever it is evaluated. But, this happens when the model user causes `write_A_to_DB` to be evaluated, not necessarily whenever `A` changes. To make it easy for the end user to perform the write, we suggest you make an output node for `writeAtoDB`:

1. Select node `write_A_to_DB` in its diagram.
2. Select the **Make Output Node** command on the **Edit** menu.
3. Move the new output node to a convenient place in the user interface of the model.

Initially, the output node will show the "Calc" button. When you click it, it writes `A` to the database. It also displays the result of evaluating `DBWrite()`, usually an empty window, not very interesting to the user. To avoid this, append `"; 'Done' "` to its definition:

```
Write_A_to_DB := DBWrite(DB, WriteTableSql(A, RowIndex,
                                LabelIndex, 'TableA'); 'Done'
```

Now, when you or an end user of the model, clicks `write_A_to_DB`, after writing `A` to the database, it shows 'Done' in the output node. It reverts to the "Calc" button, whenever `A` changes.

Database functions

The Database library on the **Definition** menu contains five functions for working with ODBC databases:

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'`) and `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 val-

ues. This statement would not normally be used in a model, but may be helpful in understanding the SQL drivers that are available.

Reading and writing text files

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 : Atom Text)
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 line feed, Chr(10), character. It then defines a local index **lines**, to be used as the index of the array of lines that it returns.

If you set optional parameter **showDialog** to true (1), it will always prompt for the file, even if it finds one by that name. Default is not.

WriteTextFile (filename, text:Text; append, warn:Boolean optional; sep:Text optional)

Writes **text** to the file **filename**. The **filename** is relative to the current data directory. 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 modifying an existing 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.

CurrentDataDirectory()

Returns the file path of the **data directory** — the directory used by **ReadTextFile()** and **WriteTextFile()**, if their filename parameter contains no other path. When starting a model, it is the directory containing the model. Any call to **ReadTextFile()** or **WriteText-**

File() that includes a path in filename parameter, will change the current data directory to the directory specified.

CurrentModelDirectory()

Returns the file path of the **model directory** — the directory into which the model is saved, by default. On starting a model, this is the directory containing the model. You can change it by selecting a different directory using the directory browser from **Save as**.

Making a browse-only model and hiding definitions

When you are ready to let others use the models you have created, you may want to save it as browse-only, so that end users can only change the variables you have designated as inputs (by making input nodes for them). You may also want to hide definitions of variables or functions to protect confidential or proprietary data or algorithms. With Analytica Enterprise, you can save models that are locked as browse-only and with hidden definitions, using these steps:

1. Hide selected definitions in your model, for entire model, modules, or by variable.
2. Save your master model file (and any linked submodules) so that you can still view and modify it yourself.
3. Select **Save a copy** from the **File** menu, and check **Lock and obfuscate** and optionally **Save as a browse-only model copy** to save an **obfuscated** copy — that is a file 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).

Tip 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 and unhiding definitions

To hide the definition of a single variable or function, select its node and select **Hide Definition(s)** from the **Object** menu, so it becomes checked. You cannot hide multiple nodes, except by hiding all nodes in a parent module. To hide the definitions of all objects in a module:

1. Select the node of the module in its parent diagram, or open the module and select no nodes inside it.
2. Select **Hide Definition(s)** from the **Object** menu, so it becomes checked.

If a variable, function, or module is hidden, when you try to view its definition, it displays:

[Definition is Hidden]

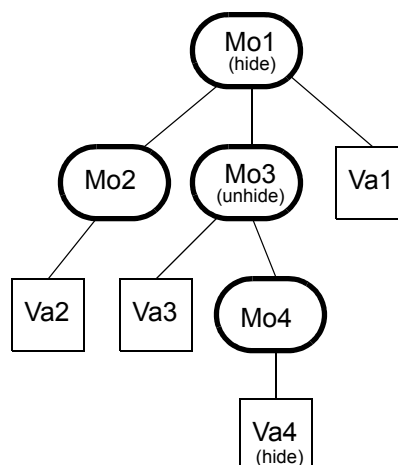
Tip The definition of a variable with an input node is always visible regardless of whether it or its parent module is marked as Hidden.

Unhiding and inheritance of hiding

Definition hiding is inherited down the module hierarchy: If you hide a module, you hide the definitions of all the objects that it contains, including its submodules and all the objects that they contain — unless you explicitly unhide an object or submodule, in which it or the objects it contains will not be hidden. To unhide a variable, function, or module:

1. Select its node in its parent diagram.
2. Select **Unhide Definition(s)** from the **Object** menu, so it becomes checked.

In the module hierarchy shown below, module **Mo1** is hidden, and therefore so are the objects it contains, module **Mo2**, **Va1**, and **Va2**. But module **Mo3** is unhidden, and therefore so are the objects it contains, **Va3** and **Mo4**. However, object **Va4** is itself explicitly hidden:

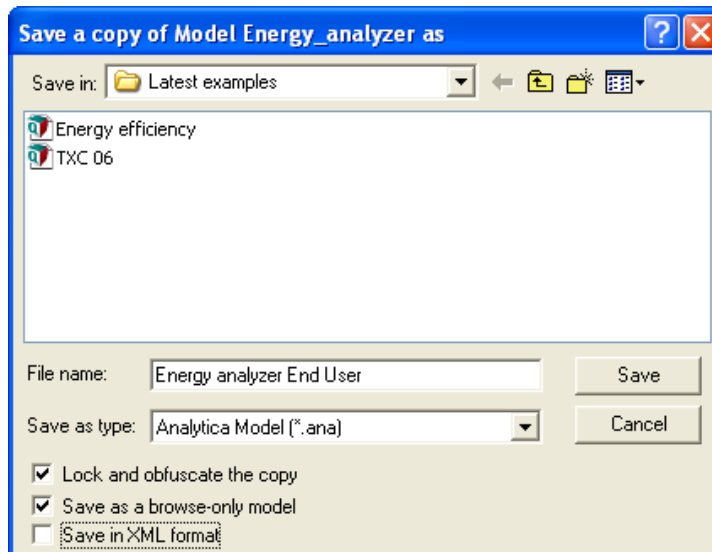


Tip The **Hide Definition(s)** and **Unhide Definition(s)** menu options are disabled if the current model, or any of its linked submodules, has been obfuscated. In this case, obfuscation has locked hiding in place.

After hiding the definitions you want, you can view your model to check everything is as you want. You can still Unhide items if you want to view or edit them. But, after saving the model in obfuscated form, no one, even you, can view hidden definitions or edit any variables that are not inputs, even if they open the model file in a text editor. That's why it's important that you save a master copy for your own use.

Saving an obfuscated copy of your model

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, to make sure that you retain an editable version for your self.

Click the **Lock and obfuscate the copy** checkbox at the bottom of the dialog to save the model in an encrypted form that will make any Hidden definitions unviewable, even if you try to edit the file.

Click **Save as a browse-only model** checkbox if you also want to prevent users from changing any variables not designated as inputs. In that case, the model will be locked in browse-only mode, as if it is being run with Analytica Player or Power Player, even if the user runs the model with an Analytica edition that normally allows editing.

A browse-only model is always obfuscated to prevent anyone from editing the source Analytica file. Thus, it automatically checks **Lock and obfuscate the copy** and the **Save in XML format** option is not available.

If you want end users to be able to use other Enterprise features, such as Database access, File reading and writing, Huge Arrays, or Performance Profiling, they will need the Power Player — or their own Enterprise edition.

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: Do not obfuscate libraries or linked submodules!

If you want to create an obfuscated version of your model, embed any libraries or sub-modules into it, rather than linking them, to avoid accidentally obfuscating them.

Tip if you read an obfuscated library or other module into your model, it will result in obfuscating the parent model, as well as any other separately filed submodules or libraries it may contain. So, you could accidentally end up obfuscating your entire model and rendering it uneditable by anyone, including you! Therefore, we strongly recommend that you do not obfuscate any library or module intended to be used by another model; and that you do not try to read an obfuscated library or module into any model.

Huge Arrays

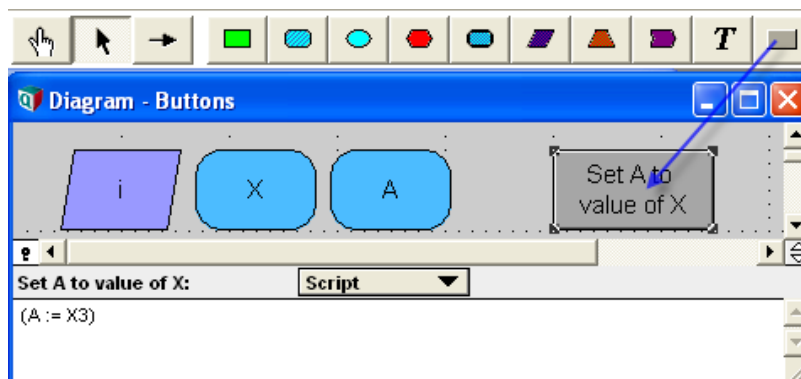
Analytica Enterprise, Optimizer, Power Player, and ADE can manage Indexes and Arrays of up to 100 Million elements in any dimension. The only practical limit on model sizes is the amount of memory. Huge Arrays means they can also handle Sample Size for probabilistic simulation up to this size. (You can set this in the **Uncertainty Setup** dialog from the **Result** menu.) This also let you read in large datasets from databases, using the ODBC functions.

Tip Editions of Analytica other than Enterprise, Optimizer, Power Player, and ADE are limited to index and sample sizes of 32,000 elements.

Creating buttons and scripts

A button is a special kind of object you can add to a diagram. It contains a script that is executed when you press the button (in browse mode). You need Analytica Enterprise (or Optimizer) to create new buttons. You can use buttons with any edition of Analytica.

To make a button To create a new button, enter edit mode, and drag from the button icon at the right end of the new object toolbar onto the diagram (or press *Control+0*):



Button script The button script is in its script attribute. You can view and edit the script in the **Attribute** panel as above, or its **Object** window, like any user-editable attribute. Any change to an identifier used in a button script automatically updates the script, just as it does in a definition of a variable or function.

Script language The script language is similar to the Analytica language used in definitions. Some key differences are:

- A script consists of one or more statements, each on a separate line, with no ";" or other separator between them.
- A statement can be an assignment to change the definition of a global variable — something not allowed in a variable definition.
- A statement in a script can be any expression valid in the Analytica modeling language, including a call to a built-in or user-defined function, as long as it fits on one line.

- A statement or expression in a script must be all on one line. A new line implies a new statement. A script does not accept BEGIN END or parentheses around a sequence of statements.
- A script can call a function that assigns to a global variable. Such a function may be called directly from a script, or indirectly from another function called from a script, and so on recursively. Such a function may *not* be from in an Analytica variable.
- Script statements can use a wide range of script commands, not available in the normal modeling language. Among other things, these can open or close windows. See <http://lumina.com/wiki/index.php/Commands>.

Consult the *Scripting Guide* on Anawiki for details of syntax of scripts.

Tip If you want a button to perform a complex series of steps, it is usually easiest to define those steps in a function, and call the function from the script, rather than write the steps directly into the script. Function definitions offer several advantages over scripts, including the ability to add inputs by drawing arrows to its node and a more flexible (and familiar) syntax.

Assigning to global variables

Assigning a definition in a script

A statement in a button script may assign to a nonlocal (global) variable, for example:

```
A := 100
```

This is *not* permitted in the definition of a variable, which may only assign to *local* variables declared within the definition of the variable, to prevent side effects — where evaluating one variable changes the value of another. See “Assigning to a local variable: $v := e$ ” on page 351.

An assignment statement in a script assigns the definition of the variable to the *expression* assigned, *not to the value* of the expression. Consider these three statements in a button script, assuming **A** and **B** are global (i.e., non-local) variables:

```
A := 1
B := A+1
A := 100
```

The second assignment changes the *definition* of **B** to the expression **A+1**, not the value of the expression, which would be 2. After these three statements, the value of **B** is 101, because the third line sets **A** to 100, which propagates to the definition of **B** is **A+1**.

Assigning a value in a script

In the context of an *expression* rather than a *script statement*, the assignment

```
B := A+1
```

sets variable **B** to the *value* of **A+1**, not the expression **A+1**. An expression is anything in the definition of a variable or function. You may also include an expression within a *script statement* simply by enclosing it in parentheses:

```
A := 1
(B := A+1)
A := 100
```

In this case, after executing this script, the definition of **B** is 2 — the value of expression **A+1** in the second line. Since the definition of **B** is now 2, not **A+1**, the third line, assigning 100 to **A** has no effect on **B**.

**Assigning a value
in a function**

There is an important exception to the rule that you may not assign to globals in a definition: You may assign to a global variable in a function that is called from a button script. It may be called directly or indirectly — that is, called from a function called from a script, and so on recursively:

```
Variable A := 100
Variable B := 2

Function IncrementA
Parameters: (x)
Definition: A := A + x

Button Add_B_to_A
Script: IncrementA(B)
```

When you press button `Add_B_to_A`, it calls function `IncrementA`, which sets the definition of `A` to the current value of `A+B`, i.e., `102`. Like any assignment in a function, it assigns the *value* not the *expression* `A+B`.

This kind of global assignment gives you the ability to create buttons and functions to make changes to a model, including such things as modifying existing model values and dependencies.

**Save a
computed value**

One useful application of assigning to a global variable is to save the results of a long computation. Normally, the cached result of a computation is stored until you change any ancestor feeding into the computation, or until you **Quit** the session. By assigning the result to a global variable, you can save it so that it remains the same when you change an input, or even when you quit and later restart the model.

A common case where this is helpful is a model containing two parts: (1) A time-consuming statistical estimation, neural network, or optimization that learns a parameter set, and (2) a model that applies the learned parameters to classify new instances. After computing the parameters, you can save them into a set of global variables, and then save and close the model. When you restart the model, you can apply the learned parameters to many instances without having to waste time recomputing them.

Consider this example:

```
Variable Saved_A := 0

Function Save_value(x)
Description: Sets Saved_A to be the value of x.
Definition: Saved_A := x

Button Save_A
Script: Save_value(A)
```

When you click button `Save_A`, it calls function `Save_value(A)`, which saves the value of `A` into global `Saved_A`. `Saved_A` will retain this value if you change `A` or any of its predecessors, or even if you quit the session, saving the model file, and later restart the model. Thus, you won't have to wait to recompute `Saved_A`. Of course, the value of `Saved_A` will not update automatically if you change any of its predecessors, the way `A` does. You need to click button `Save_A` again to save a new value of `A`.

If the value of `A` is an array with nonlocal indexes, the definition of `Saved_A` will be an **edit table**, using those indexes. Any subsequent change to those indexes will affect,

Assign to an attribute

and possibly invalidate the table. If you want to make sure this doesn't happen, you may want to save copies of the indexes, and transform the table to use the saved indexes.

You may assign to any user-editable attribute of a (nonlocal) variable or other object, subject to the same restrictions as assigning a value — i.e., you may do it only in a function called from a script, directly or indirectly. You may *not* assign to an attribute in the definition of a variable. The syntax is:

```
<attrib> OF <object> := <text>
```

where **<attrib>** is the name of an editable attribute, including *Title*, *Units*, *Description*, *Definition*, *Check*, *Domain*, and *Author*; **<object>** is the identifier of a user-defined, nonlocal object, variable, function, module, etc.; and **<text>** is a text value. For example:

```
Function Retitle(o, t)
Description: Sets the title of object o to text t.
Parameters: (o: Object; t: Atom Text)
Definition: Title OF o := t

Variable Gray := 0
Title: Gray

Button Change_title
Script: Retitle(Gray, 'Earl '&(Title of Gray))
```

When you click button `Change_title`, it calls function `Retitle` applying it to variable `Gray`, prefixing the old title of `Gray` with `'Earl '` to become `'Earl Gray'`. It will do it again each time you press the button. Notice that the object whose attribute you are resetting may be passed to the function, provided the parameter is qualified as an `Object` in the parameters declaration.

If the text is an array, it flattens the array into a single text value before the assignment — probably not what you want. So, it is best only to assign atomic text values.

If you want to assign a new definition as text (rather than assigning the value of an expression), you may assign to the definition thus:

```
Definition OF X := Y^2
```

You may use this method to assign new values to various internal attributes, such as **NodeLocation**, **NodeColor**, **NodeSize**, and **NodeFont**, letting you change the way nodes appear on a diagram. Consult the *Scripting Guide* on Anawiki for details of syntax.

EvaluateScript(t)

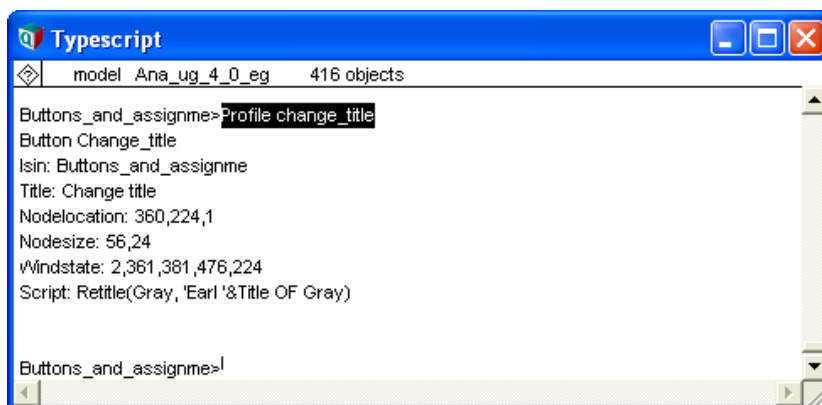
This function evaluates a text value **t** as if it was a script. This means **t** can contain script commands, assignments to globals, and other statements permitted in scripts.

Avoid using **EvaluateScript(t)** except in script functions — that is, functions called from button scripts. This will minimize the danger of undermining the no-side-effects rule.

Typescript Window

The Typescript window offers an old-fashioned command-line user interface, like the Windows CMD program or a Unix shell: It shows a prompt — the title of the model or module — at the start of each line. You can type in a script command. It will print any results as text, and show another prompt. The typescript is occasionally useful for advanced users who wish to inspect internal details of a model. You may also use it to test out commands that you want to use in a button script.

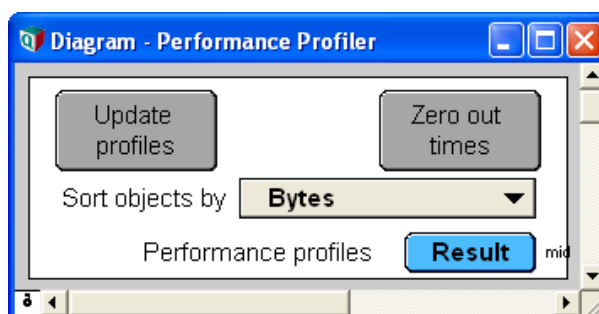
To open the Typescript window, press *Control+' (single apostrophe)*:



Performance Profiler

The Performance Profiler library shows you the computation time and memory space used by each variable and function. If you have a large model that takes a long time to run or uses a lot of memory, you might want to find out which variables or functions are using the Lion's share of the time or memory. As experienced programmers know, the results are often a surprise. With the results from the Performance Profiler, you know where to focus your efforts to make the model faster or use less RAM.

First add **Performance Profiler.ANA** from the **Libraries** folder into your model:



Now display the results (table or graph) for the variables whose performance you want to profile. Open the library, and click **Performance profiles**:

The screenshot shows a window titled "Result - Performance profiles" with a table of performance data. The table has columns for Class, Module, Bytes, CPU msecs, and msec w ancestors. The data is as follows:

	Class	Module	Bytes	CPU msecs	msec w ancestors
Revenue	Variable	Project revenue analysis	2,356,040	301	401
HPV revenue	Objective	Project revenue analysis	168,316	80	511
Triangular2(n)	Function	Index library	24	70	70
Prob_from_inputs(n)	Function	Index library	24	20	20
Product released	Decision	Project revenue analysis	168,316	10	30
R&D cost	Chance	Project revenue analysis	168,426	10	30
Production cost	Chance	Project revenue analysis	168,316	10	20
Merit	Variable	Portfolio optimization	168,316	10	521
R&D budget	Decision	Model details	110	1	1
Project inputs	Variable	Model details	5,228	1	1

This table lists the variables and functions by row, with the **class** of the object, parent **module**, **Bytes** of RAM (random access memory), and **CPU msecs** (milliseconds of time used by the central processing unit). The last column, **msecs w ancestors**, shows the CPU milliseconds to compute each variable or function including all its ancestors — i.e., the variables and functions it uses. The Profiler shows all variables and functions that use more than 24 bytes of RAM (the minimum) or take more than 1 millisecond to compute. Use **Sort objects by** to sort the table by any column.

If you want to inspect a variable or function to see why it's taking so much time or memory, just click its title in the **.objects** index column to open its **Object** window.

Update profiles After computing more results, click this button to update the performance profile to reflect the additional time and memory used.

Zero out times If you want to look at the incremental time used by additional results, or another computation, first click this button to zero out the times already computed.

Understanding memory usage For complex definitions, it may use much more RAM while it is computing than it needs to cache the final result — the Profiler reports only the latter. The Bytes show the RAM used to store the value of each variable, Mid, Probabilistic, or both, depending on which it has computed. Typically, an array takes about 12 bytes per number to store. For example, an uncertain dynamic array of numbers, with an index i of 20 elements, **Time** has 30 elements, and **Sample size** = 1000, would use about $20 \times 30 \times 1000 \times 12 = 7,200,000$ bytes or 7.2 Megabytes. Analytica uses an efficient representation for arrays with many zeroes (sparse arrays) or many repeated values. An array that is an exact or partial copy of another array may share slices. In such cases, it may actually use less memory than it reports.

Understanding computation time The CPU time listed is the time it took to evaluate the mid and/or prob value of each variable or function, depending on which type of evaluation it did. It is zero if the results computed did not cause evaluation of the variable or function. A variable is usually only computed at most once each for its mid and prob value. Rare exceptions include when the variable is referenced directly or indirectly in a parameter to **Whatif** or **Whatifall** (page 286), which might cause multiple evaluations. A function may be called many times. The CPU time reported is the sum over all these evaluations.

Time and virtual memory Like most 32-bit applications on Windows, Analytica can use up to 3 Gigabytes of memory. If your computer doesn't have that much RAM installed, and it needs more than is available, it may use virtual memory — that is, it saves data onto the hard disk. Since

reading and writing a hard disk is usually much slower than RAM, using virtual memory often causes the application to slow down substantially. In this case, finding a way to reduce memory usage below the amount of physical RAM available may speed up the application considerably. Another approach is to install more RAM, up to 4 Gigabytes.

Performance profiling attributes and function

The Performance Profiler library uses a function, two attributes, and a command, which are also available for you to write your own functions using memory or time. For an example of how to use them, you can open up the library.

MemoryInUseBy(v) This function returns the number of bytes in use by the cached result(s) for variable *v* — with the same disclaimer that shared memory may be counted more than once. It includes memory used by mid and prob values if those results have been computed and cached, but it doesn't force them to be computed if they haven't been.

These two special read-only attributes:

EvaluationTime This attribute returns the time in seconds to evaluate its variable or function, not including the time to evaluate any of its inputs.

EvaluationTimeAll This attribute returns the time in seconds to evaluate its variable or function, including the time to compute any of its inputs that needed to be evaluated (and their inputs, and so on.).

ResetElapsedTimings This command sets these attributes back to zero. Like any command, you may use it in a button script, the Typescript, but not in a regular definition.

Tip These features, including the Performance Profiler are only available for Analytica Enterprise, Power Player, and ADE editions.

RunConsoleProcess(program, cmdline, stdin, block)

This function lets an Analytica model run a *console process*, that is, start another Windows application. The application or program may be a simple one with no graphical user interface, or it can interact directly with the user. **RunConsoleProcess()** can provide data as input to the program and return results generated by the application. The **program** parameter contains text to specify the directory path and name of the program. It can feed input to the program via command line parameters in **cmdLine**, via the **stdin** parameter, piped to the *StdIn* input channel of the program, or via a data file created with **WriteTextFile()**. Normally, when the program completes, **RunConsoleProcess** returns a result (as text) any information the program writes to **stdOut**. Analytica can also use **ReadTextFile()** to read any results the program has saved as a data file.

Required parameter

program Text to specify the directory path and name of the Windows application (program) to run. A relative path is interpreted relative to Analytica's **CurrentDataDirectory**. If it cannot find or launch the application, it gives an error message.

Optional parameters:

cmdline	Text given input to the program as command line parameters. (It is separated from the program parameter to protect against a common type of virus attack.)
stdin	Text to be piped to the StdIn input channel of the program.
block	<p>If you omit block or set it to True (1), RunConsoleProcess() <i>blocks</i> — that is, after calling the process, Analytica stops and waits until the console process terminates and returns a result before it resumes execution. While blocked, Analytica still notices Windows events: If you press <i>Control+Break</i> (or <i>Control+.</i>) before the process terminates, it kills the process, and ends further computation by Analytica, just as when Analytica is computing without another process.</p> <p>If you set block to False (0), RunConsoleProcess() spawns an independent process that runs concurrently with Analytica. Within Analytica, it returns empty text. Analytica and the spawned process each continues running independently until it terminates. If you press <i>Control+Break</i> (or <i>Control+.</i>), it interrupts and stops further computations by Analytica, but has no effect on the spawned process. An unblocking process may continue running even after you exit Analytica. Unblocking processes are useful when you want to send data to another application for display, such as a special graphing package or GIS, or for saving selected results. It is difficult for Analytica to get any results or status back from an unblocking process. If you need results back it is usually best to use a blocking process.</p>
curDir	The directory the process should use as its default directory to read and write files. If omitted, it uses the application's own directory as the default.
priority	Sets the priority that Windows should give the spawned process relative to the Analytica process. The default (0) is the same priority as the Analytica process. Setting it to +1 or +2 raises its priority, taking more of the CPU for the process. -1 or -2 lowers the priority, letting other processes (including Analytica) use more of the CPU.
showErr	Controls the display of error messages from a blocking process. By default, if the process writes anything to stderr , Analytica displays it as an <i>error</i> message when the process terminates. If showErr=2 it shows any text in stderr as a <i>warning</i> message. If showErr=0 , it ignores anything in stderr . Analytica always ignores any error in an unblocking process, which is assumed to control the display of its own errors.

RunConsoleProcess() fully supports Intelligent Arrays. If any parameter is passed an array, it runs a separate process for each element of the array. It runs multiple blocking processes sequentially. It runs multiple non-blocking processes concurrently.

Examples

Run a VB Script Suppose the Visual Basic program file **HelloWorld.vbs** is in your model directory and contains:

```
WScript.Echo "Hello World"
```

Your call to **RunConsoleProcess** might look like:

```
RunConsoleProcess("C:\Windows\System32\CScript.exe",
  "CScript /Nologo HelloWorld.vbs")
```

The first parameter identifies the program to be launched. You don't need to worry about quoting any spaces in the path name. The second parameter is the command line as it might appear on a command prompt. This expression returns the text value "Hello World".

To send data to the **StdIn** of the process, include the optional parameter **StdIn**:

```
RunConsoleProcess("C:\Windows\System32\CScript.exe",
  "CScript /Nologo HelloWorld.vbs", StdIn: MyDataToSend )
```

where **MyDataToSend** is an Analytica variable that gives a text value.

To run a batch file Suppose the directory **c:\Try** contains a data file named **data.log** and a batch file named **DoIt.bat** containing:

```
# DoIt.bat - dump the log
Type data.log
```

This batch file assumes it is run from the directory **C:\Try** so does not mention the directory of **data.log**. From Analytica, you call:

```
RunConsoleProcess("C:\Windows\System32\Cmd.exe", "Cmd /C DoIt.bat",
  CurDir: "C:\Try")
```

or you can run it directly:

```
RunConsoleProcess("DoIt.bat", "DoIt.bat", CurDir: "C:\Try")
```

To read data from a URL If you have the program **ReadURL.exe** (which you can download from the Anawiki), you can use it to read the contents of a web page into Analytica:

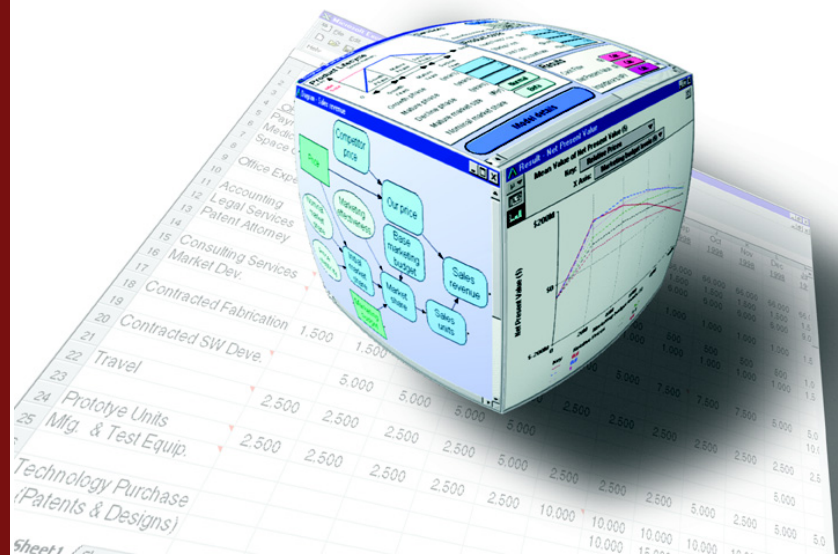
```
RunConsoleProcess("ReadURL.exe", "ReadURL " & url)
```

where **url** is a text string as would appear in the address bar of your browser. You can download the **ReadURL.exe** program by clicking the link and saving. If you save **ReadURL.exe** into a directory other than **CurrentDataDirectory**, you will also need to specify its directory path in the program parameter above.

Appendices

The following appendices shows you:

- How to select an appropriate sample size
- The complete set of Analytica Menus
- The specifications for Analytica
- The list of reserved words and error message types
- Forward and backward compatibility information
- A bibliography
- A list of all the Analytica functions



Appendix 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 232). 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.

Uncertainty about the mean

First, suppose you are primarily interested in the precision of the mean of your output variable \mathbf{y} . Assume you have a random sample of m output values generated by Monte Carlo simulation:

$$(y_1, y_2, y_3, \dots, y_m) \quad (1)$$

You can estimate the mean and standard deviation of \mathbf{y} using the following equations:

$$\bar{y} = \sum_{i=1}^m \frac{y_i}{m} \quad (2)$$

$$s^2 = \sum_{i=1}^m \frac{(y_i - \bar{y})^2}{(m-1)} \quad (3)$$

This leads to the following confidence interval with confidence α , where c is the deviation for the unit normally enclosing probability α :

$$\left(\bar{y} - c \frac{s}{\sqrt{m}}, \bar{y} + c \frac{s}{\sqrt{m}} \right) \quad (4)$$

Suppose you wish to obtain an estimate of the mean of \mathbf{y} with an α confidence interval smaller than w units wide. What sample size do you need? You need to make sure that:

$$w > 2c \frac{s}{\sqrt{m}} \quad (5)$$

or, rearranging the inequality,

$$m > \left(\frac{2cs}{w} \right)^2 \quad (6)$$

To use this, first make a small Monte Carlo run with, say, 10 values to get an initial estimate of the variance of \mathbf{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:

$$m > \left(\frac{2 \times 2 \times 40}{20} \right)^2 = 8^2 = 64 \tag{7}$$

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 \mathbf{y} are relabeled so that they are in increasing order,

$$y_1 \leq y_2 \leq \dots y_m$$

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

$$i = \lfloor mp - c\sqrt{mp(1-p)} \rfloor \tag{8}$$

$$k = \lceil mp + c\sqrt{mp(1-p)} \rceil \tag{9}$$

Suppose you want to achieve sufficient precision such that the α confidence interval for the p th 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 $(p - \Delta p)$ th and $(p + \Delta p)$ th fractiles. What sample size do you need? Ignoring the rounding, you have approximately

$$i = m(p - \Delta p), \quad k = m(p + \Delta p) \tag{10}$$

Thus,

$$k - i = 2m\Delta p \tag{11}$$

From Equations (8) and (9) above, you have

$$k - i = 2c\sqrt{mp(1-p)} \tag{12}$$

Equating the two expressions for $k - i$, you obtain

$$2m\Delta p = 2c\sqrt{mp(1-p)} \tag{13}$$

$$m = p(1-p)\left(\frac{c}{\Delta p}\right)^2 \tag{14}$$

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:

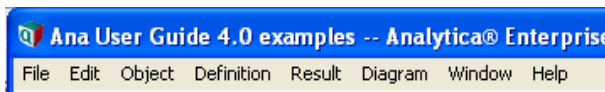
$$m = 0.90 \times (1 - 0.90) \times (2/0.05)^2 = 144 \tag{15}$$

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,

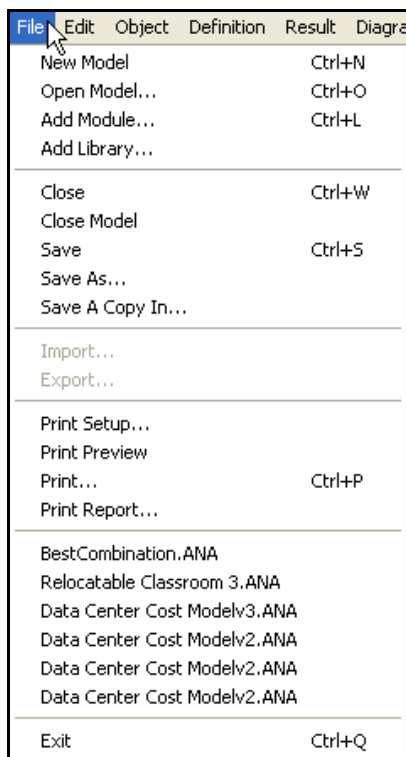
$$m = 0.5 \times (1 - 0.5) \times (2/0.01)^2 = 10,000 \tag{16}$$

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. Menus



File menu



- 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 Opens file finder at Analytica Libraries folder to add a library module.
- Close Closes the active window.
- Close Model Closes the model after prompting you to save the file if it has changed.
- Save Saves the model in its file. If the model has never been saved before, prompts for a file name and folder. If it has linked modules that have changed, it also saves them.
- Save As Saves the active model, filed module, or filed library as a new file, after asking for new file name and folder.
- Save A Copy In Saves a copy of the active model (or filed module) into a new file, after prompting for a file name, leaving the original file name for future saves.
- Import Imports the contents of a text or data file into the selected variable definition. See "Importing and exporting" on page 318.
- Export Exports the contents of the selected field or cells into a file. See "Importing and exporting" on page 318.
- Print Setup Opens a dialog for selecting paper size, orientation, and scaling options for printing.
- Print Preview Opens a view showing where page breaks will occur before the current window is printed.
- Print Opens a dialog for selecting the printer, number of copies you want to print, and other printing options.
- Print Report Opens a dialog for printing multiple diagrams, **Object** windows, and result windows at the same time. See "Printing" on page 25.
- Recent files Lists the six most recently opened Analytica model files. Select one to open that model.
- Exit Quits the Analytica application, after prompting to save any model changes to file.

Edit menu

Edit	Object	Definition	Resu
Can't Undo		Ctrl+Z	
Cut		Ctrl+X	
Copy		Ctrl+C	
Paste		Ctrl+V	
Paste Special...			
Clear			
Select All		Ctrl+A	
Duplicate Nodes		Ctrl+D	
Copy Diagram			
Insert Rows		Ctrl+I	
Delete Rows		Ctrl+K	
Preferences...			
OLE Links...			

- Undo Undoes your last action.
- Cut Cuts the selected text, node(s), graph, or table cells into the clipboard temporarily for pasting.
- Copy Copies the selected text, node(s), graph, or table cells into the clipboard temporarily for pasting. See "Copying and pasting" on page 310.
- Paste Pastes the contents of the clipboard at the insertion point in a text, diagram, or table, or replaces the current selection. See "Copying and pasting" on page 310.
- Paste Special Brings up a dialog to select the format of data to OLE link into an edit table.
- Clear Deletes the selected text or node(s).
- Select All Selects all the text in an attribute field, nodes in a diagram, or cells in a table.
- Duplicate Nodes Duplicates the selected nodes onto the same diagram. See "Duplicate nodes" on page 51.
- Copy Table or Copy Diagram When a table window is active, **Copy Table** copies the entire multidimensional object as a tab-delimited list of tables. When a **Diagram** window is active, **Copy Diagram** copies a picture of the diagram for pasting into a graphics application. See "Copying and pasting" on page 310.
- Insert Rows or Insert Columns Inserts an item into a list, or a row in a table, by copying the current item, or row. If a column in a table is selected, **Insert Columns** inserts an item or column. See "Editing a table" on page 171.
- Delete Rows or Delete Columns Deletes the selected item or items in a list, or rows or columns in a table. See "Editing a table" on page 171.
- Preferences Opens the **Preferences** dialog to examine or change various options. See "Preferences dialog" on page 59.
- OLE Links Opens a dialog to let you change properties for OLE links from external applications into your model. (See "Importing, Exporting, and OLE Linking Data" on page 309.)

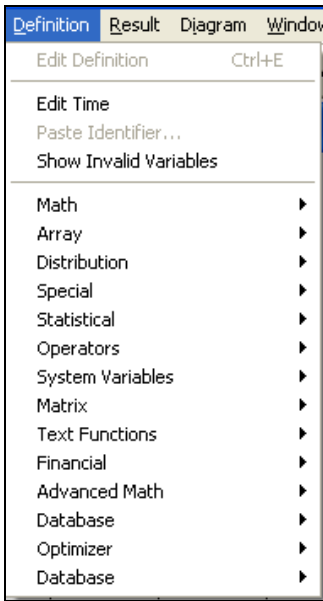
Object menu

Object	Definition	Result	Diagram
Find...		Ctrl+F	
Find Next		Ctrl+G	
Find Selection		Ctrl+H	
Make Alias		Ctrl+M	
Make Importance			
Make Input Node			
Make Output Node			
Show By Identifier		Ctrl+Y	
Show With Values			
Attributes...			
Hide Definition(s)			
Unhide Definitions(s)			

Find	Opens a Find dialog box to search for an object by its identifier or title. See “Finding variables” on page 326.
Find Next	Finds the next object that partially matches the previously defined text value. See “Finding variables” on page 326.
Find Selection	Finds an object by its identifier that matches the currently selected text. See “Finding variables” on page 326.
Make Alias	Creates an alias for the selected object(s). See “Alias nodes” on page 55.
Make Importance	Creates an importance variable (and index) to compute the importance (rank correlation) of all uncertain inputs for the selected variable. See “Importance analysis” on page 282.
Make Input Node	Creates an input node for the selected node(s). See “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.” on page 128.
Make Output Node	Creates an output node for the selected node(s). See “Using output nodes” on page 131.
Show By Identifier	Show the identifier instead of title of each object in the current diagram, edit table, Result window, or Outline view. Toggle to show title again.
Show With Values	Shows the mid values of the variable and all its inputs in each Object window. See “Showing values in the Object window” on page 24.
Attributes	Opens the Attribute dialog box to set the visibility of attributes and define new attributes. See “Managing attributes” on page 327.
Hide Definition(s)	Marks the currently selected node or module as hidden, so that their definitions are invisible. (Analytica Enterprise only)
Unhide Definition(s)	Unhides the currently selected node or module. This overrides any settings in parent modules to hide definitions. (Analytica Enterprise only)

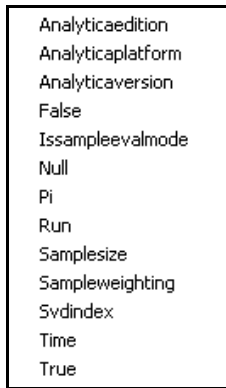
Definition menu

This menu is hierarchical. Each library lists the functions or other constructs it contains. The middle partition lists built-in libraries. At the bottom, are any libraries you have created or added. If you view and select a subitem when editing a definition, it will paste it into the definition.



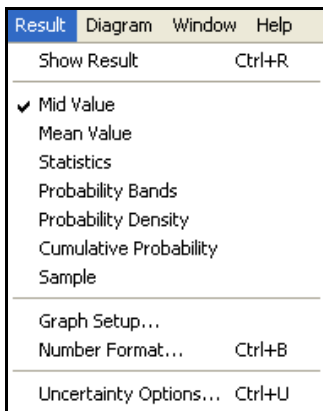
- Edit Definition** Opens the appropriate view for editing the definition of the selected variable. If the variable is defined as a distribution or sequence, the **Object Finder** opens. If it is defined as a table or probability table, its edit table window opens. Otherwise, an **Object** window or **Attribute** panel opens, depending on the Edit attributes setting in the **Preferences** dialog box. See “Preferences dialog” on page 59.
- Edit Time** Opens the **Object** window for the **time** system variable. See “The Time index” on page 298.
- 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” on page 121.
- Show Invalid Variables** Displays a window listing all variables with invalid or missing definitions. See “Invalid variables” on page 329.
- Math** See “Math functions” on page 146.
- Array** See Chapter 11, “Arrays and Indexes,” and Chapter 12, “More Array Functions.”
- Distribution** See Chapter 15, “Probability Distributions.”
- Special** Displays a list of unusual or less commonly used functions in the Special library.
- Statistical** See “Statistical functions” on page 274.
- Operators** Arithmetic, comparison, logical, and conditional operators. See “Operators” on page 142.
- System Variables** System Variables submenu (see below).
- Matrix** See “Matrix functions” on page 203.
- Text Functions** See “Converting a number to text” on page 147.
- Financial** See “Financial functions” on page 214.
- Advanced Math** See “Converting a number to text” on page 147.
- Optimizer** Appears only if you have the Optimizer activated. See *Optimizer Guide* for more.
- Database** Appears only in Analytica Enterprise. See “Database functions” on page 381.
- your libraries** It lists the names of any libraries that you have defined or added to the model, each with a submenu that lists the functions contained in the library. See Chapter 20, “Building Functions and Libraries.”

System Variables submenu



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 232.
Time	The index variable identifying the dimension for dynamic simulation (the Dynamic() function). See "The Time index" on page 298.
True	The logical (Boolean) constant that evaluates numerically to nonzero.

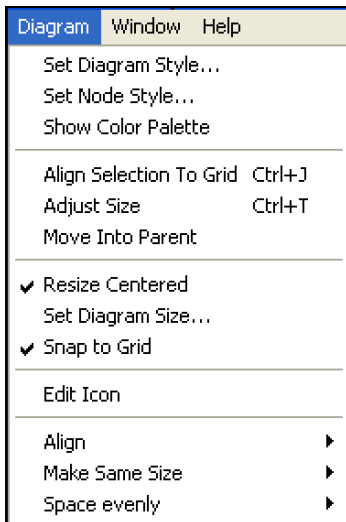
Result menu



Show Result	Opens a Result window for the selected object. See "The result window" on page 28.
Mid Value	Displays the Mid or deterministic value. See "Uncertainty views" on page 32.
Mean Value	Displays the mean of the uncertain value. See "Uncertainty views" on page 32.
Statistics	Displays statistics of the uncertain value in a table as set in the Uncertainty Setup dialog box. See "Uncertainty views" on page 32.
Probability Bands	Shows probability bands (percentiles) as set in the Uncertainty Setup dialog box. See "Uncertainty views" on page 32.
Probability Density	Displays a probability density graph for an uncertain value. For a discrete probability distribution, Probability Mass replaces this command. See "Uncertainty views" on page 32.
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 views" on page 32.
Sample	Displays a table of the values determined for each uncertainty sample iteration. See "Uncertainty views" on page 32.
Graph Setup	Displays a dialog box to specify the graphing tool, graph frame, and graph style. See "Graphing roles" on page 91.
Number Format	Displays a dialog box to set the number format for displays of results. See "Number formats" on page 86.

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 232.

Diagram menu



- Set Diagram Style Displays a **Diagram style** dialog to set default arrow displays, node size, and font for this diagram. See “Diagram Style dialog” on page 81.
- Set Node Style Displays **Node style** dialog to set arrow display and font for the selected node(s). See “Node Style dialog” on page 82.
- Show Color Palette Displays the color palette to set the color of the diagram background or of selected nodes. See “Recoloring nodes or background” on page 80.
- Align Selection To Grid Aligns selected node(s) to the diagram grid. See “Align to the grid”.
- 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 82.
- Move Into Parent Moves the selected object from the current diagram to its parent diagram. See “The Object window” on page 22.
- 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 selected nodes” on page 76.
- Set Diagram Size Opens a dialog to let you set the area covered by the diagram. The diagram size increases automatically to show any nodes outside the original area.
- Snap to Grid Turns alignment to the diagram grid on or off in edit mode. See “Align to the grid”.
- Edit Icon Opens a window to draw or edit an icon for the selected node. See “Adding icons to nodes” on page 133.
- Align See description of submenu below.
- Make Same Size See description of submenu below.
- Space evenly See description of submenu below.

Align submenu

Left Edges	(CTRL+Left)
Centers Left and Right	(CTRL+F9)
Right Edges	(CTRL+Right)
Left and Right Edges	(CTRL+=)
Top Edges	(CTRL+Up)
Centers Up and Down	(SHIFT+F9)
Bottom Edges	(CTRL+Down)

Left Edges	Aligns left edges.
Centers Left and Right	Aligns centers along the same horizontal line.
Right Edges	Aligns right edges.
Left and Right Edges	Moves and changes width so left and right edges are aligned vertically.
Top Edges	Aligns top edges.
Centers Up and Down	Aligns centers along the same vertical line.
Bottom Edges	Aligns bottom edges.

Make Same Size submenu

Width	(=,Right)
Height	(=,Down)
Both	(=,=)

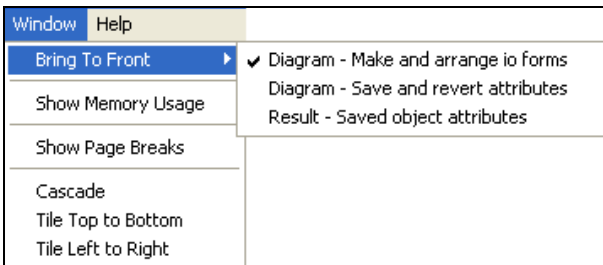
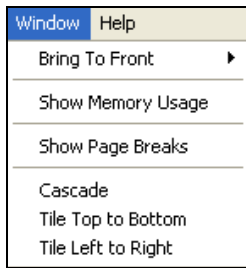
Width	Makes all nodes the same width.
Height	Makes all nodes the same height.
Both	Makes all nodes the same width and height.

Space evenly submenu

Across
Down

Across	Spaces nodes evenly horizontally between leftmost and rightmost node.
Down	Spaces nodes evenly vertically between top and bottom node.

Window menu



Bring To Front	Displays a list of the current windows; select one to display on top.
Show Memory Usage	Opens a window showing memory usage. See “Numbers and arrays” on page 410.
Show Page Breaks	Shows page breaks for the active diagram.
Cascade	Rearranges all open windows using a standard size, organized so that you can see the title bar of each one.
Tile Top to Bottom	Rearranges all open windows so that they fill the application window horizontally.
Tile Left to Right	Rearranges all open windows so that they fill the application window vertically.

Help menu



User guide	Opens the <i>User Guide</i> .
Optimizer	Opens the <i>Optimizer Manual</i> (only appears in Optimizer-enabled version of Analytica).
Tutorial	Opens the <i>Analytica Tutorial</i> .
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 software registration 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.

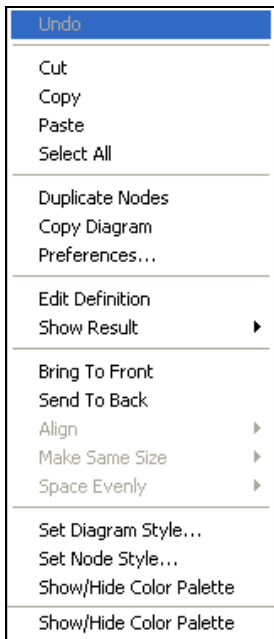
Tip The options that appear on the help menu will vary depending on your computer setup and the version of Analytica you have. If you do not have Adobe Acrobat installed on your computer, the items that appear above the line will change to only:

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

Right mouse button menus

Click the right mouse button on one or more nodes, a diagram background, or in other windows to get a menu of useful commands. The list of commands depends on the context. This menu is what you get when you right-click a node.

These two menu options appear *only* when you right-click one or more nodes. This is the only way to move some nodes in front of others:



- 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. Analytica Specifications

Hardware and software

CPU's supported	Pentium or higher and equivalent AMD processors recommended
System Software	Windows XP, Vista, and Server
Memory requirements	128 MB (512 MB+ recommended)
Application size	Approximately 6 MB

Objects

Number of system objects	738
Maximum user-defined objects	31,900
Maximum number of local-variables	No fixed limit

Uncertainty

Probability methods	Random Latin HyperCube Median Latin HyperCube Monte Carlo
Maximum sample size	99,999,999 for Analytica Enterprise, Optimizer, Power Player and ADE 30,000 (other Editions) <i>limited by available memory</i>
Random sampling methods	Minimal Standard L'Ecuyer Knuth

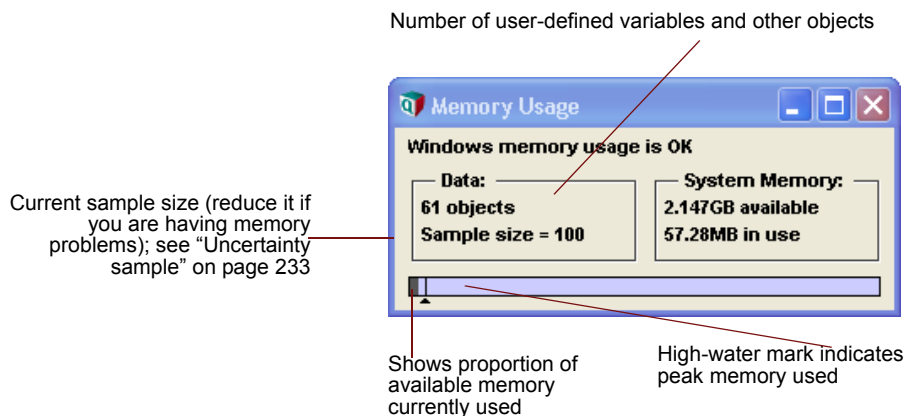
Numbers and arrays

Number precision	15 significant digits for floating-point numbers 9 digits for integers
Maximum elements in a dimension	99,999,999 (Analytica Enterprise, Optimizer, Power Player, and ADE) 30,000 (other Editions)
Maximum dimensions in an array	15

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.



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 232) or changing the application memory size (see next section).

Appendix D. Identifiers Already Used


Each object, whether built-in or created by you, must have a unique identifier. This identifier must start with a letter, and can be up to 20 characters including letters, digits, and underscores. If you try to create an identifier already in use, it will warn you and append a digit to make it unique.

To see all identifiers currently in use:

1. Press *Control+r*, to open the **Typescript** Window
2. Type `List`, followed by *Enter*.

Appendix E. 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 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” on page 59). 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 369 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 parameter 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 tries to perform a division by zero, it displays a warning with an option to continue calculating. Three possible error codes may be returned as a result of an invalid calculation:

Code	Meaning
INF	Infinity, such as $1/0$
NAN	Not A Number: Results from invalid functions such as Sqrt (-1), or $0/0$.
NULL (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.

You can test for these results in an expression using "**X=INF**", **Isnan(X)**, or **X=NULL**.

System error If you see this message type, please contact Lumina Decision System's technical support department to report the error. (See inside the front cover for contact information, or go to www.lumina.com.)

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 "Numbers and arrays" on page 410).

Appendix F. Forward and Backward Compatibility

Backward compatibility

Models created in earlier releases of Analytica can be loaded, viewed, evaluated, and modified with Analytica 4.0. There is no fundamental difference in file format, so no file conversion must take place. There are, however, some changes that could affect your results when migrating a model from a previous release to 4.0.

When you are trying a model for the first time in 4.0, the first thing you should do is ensure that *Show Result Warnings* is checked in the **Preferences** dialog. While evaluating your model, avoid selecting *Ignore Warnings* if warnings do appear. If any expression in your model produces a warning that you can live with, surround the expression with **IgnoreWarnings(...)** to suppress the warning, so that you don't feel compelled to select the **Ignore Warnings** button. When you leave warnings on while your model evaluates, many potential backward compatibility issues, if present, will be reported to you.

The most commonly encountered difference is the multiplication of **NaN** or **INF** by zero. In earlier releases of Analytica, multiplying **INF** or **NaN** by zero results in 0, while now it results in **NaN** (with a warning, if "Check result warnings" is on). The new 4.0 treatment here is in accordance with the IEEE 754 binary floating point arithmetic standard. It was not uncommon by Analytica 3.1 modelers to zero-out **NaNs** and **INFs** with a multiplication by zero. Now you may need to use IF-THEN-ELSE instead. If you find certain results have suddenly changed to NaN in 4.0, this is the likely reason.

There have been many bug fixes in Analytica 4.0, so if for some reason your model utilized an undocumented feature that was really a bug, a change in model behavior could result. There are also numerous uncommon situations where there are syntactic and evaluation differences between the releases. In a correctly functioning model from a previous release, you are unlikely to encounter these, but they are documented in detail on the Analytica Wiki at http://lumina.com/wiki/index.php/Changes_in_4.0_that_could_impact_3.x_models.

Generally when you load a model into Analytica and evaluate uncertain variables in an identical sequence, the identical random samples are returned. (Also, when you reset the random seed, you can reproduce the same sample.) In most, but not all, cases, Analytica 4.0 will return the same sample returned by Analytica 3.1; however, this is not guaranteed, and there are several cases where the sample is different. Although the samples in 4.0 and 3.1 come from the same distribution, the precise points in the random samples may be different, causing changes in your results. Uncertain results inherently have a certain "sampling error" arising from the fact that a finite sample size is used, and these differences, when they occur, are reflecting this sampling error. Two uses of distribution functions that are likely to result in a sampling difference are certain hierarchical uses of distributions, in which the parameters to distribution functions are themselves uncertain, and use of the **Truncate** function (which now preserves rank order). In the hierarchical cases, several distribution functions are more efficient now, requiring fewer random numbers to be generated when producing the entire sample. In either case, once a different number of pseudo-random numbers are utilized, you will see all samples from that point on changed.

Forward compatibility

It is also possible to run models created or edited in Release 4.0 in earlier releases of Analytica, such as Analytica 3.1, provided you don't rely on functions, features, or functionalities new to Analytica 4.0. The models will load into Analytica 3.1, although you may encounter problems during parsing or evaluation in the places where 4.0 features are used. A few 4.0 features may be stripped out of the model if it is re-saved from 3.1,

including, for example, graphing settings for graphs viewed while the model was loaded in 3.1.

There are two issues related to edit tables that could potentially create a problem when loading a model edited with 4.0 into an earlier release of Analytica. If a computed index has changed in the model since the downstream edit tables have been accessed, some edit tables may not yet be fully spliced. When loaded into Analytica 3.1, unspliced edit tables will not successfully parse. To avoid this, prior to saving the model from Analytica 4.0, access the typescript windows by pressing the *F12* key and type:

```
SpliceTable all
```

A second issue arises if any of your edit tables have blank (empty) cells. Edit tables with blank cells will not parse in earlier releases, so you must ensure that all cells in your edit tables have value, even if just 0 or null.

In general, because there are so many new features in 4.0, it is likely that you may have to test and debug your model in 3.1 to eliminate the use of new features or functions, if its use in 3.1 is required. Please see “What’s new in Analytica 4.0?” for information on the many things that are new to 4.0.

Appendix G. 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

When viewing this list online, click 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, Argmin, Argmax, Average, Max, Min, Product, Subindex, Sum, CondMin, CondMax, PositionInIndex

Transforming Arrays

Cumproduct, Cumulate, Integrate, Normalize, Rank, Sortindex, Uncumulate

Selecting from Arrays

v[I=v], x[time=n], Slice, Subscript

Interpolating

Cubicinterp, Linearinterp, Stepinterp

Other Array Functions

Concat, ConcatRows, IndexNames, Size, Sortindex, Subindex, Subset, Unique, Rank, IndexesOf, IndexValue

Tables and Arrays

MDArrayToTable, MDTable

Matrix Functions

Decompose, Determinant, Determtable, DotProduct, Invert, MatrixMultiply, Transpose, EigenDecomp, 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, Prohtable, Uniform

Statistical Functions

Frequency, Getfract, Kurtosis, Mean, Mid, Probability, Probbands, Rankcorrel, Regression, Sample, Sdeviation, Skewness, Statistics, Variance, CDF, PDF, Correlation, Covariance

Text Functions

&, Asc, Chr, FindinText, JoinText, SelectText, SplitText, TextUpperCase, TextLength, TextLowerCase, TextSentenceCase, TextReplace

Sensitivity Analysis

Correlation, Dydx, Elasticity, Rankcorrel, Regression, Whatif, WhatIfAll

Special Functions

Dynamic, Error, Evaluate, IgnoreWarnings, Iterate, Subindex, Time, Today, MsgBox, ShowProgressBar, 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, ReadTextFile, WriteTextFile

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, Form, Index, Library, Model, Module, Objective, Variable

Parameter Qualifiers

All, Atom, Array, Ascending, Coerce, Context, ContextSample, Descending, Mid, Index, IsNotSpecified, Nonnegative, Number, Optional, Positive, Prob, Reference, Sample, Text, Unevaluated, Variable

Optimizer Functions

Refer to the *Optimizer Guide* for information on these functions.

LpDefine, LpFindIIS, LpObjSA, LpOpt, LpRead, LpSolution, LpStatusNum, LpStatusText, LpWrite, LpWriteIIS, NlpDefine, QpDefine

This glossary includes a compilation of terms specific to Analytica as well as statistical terms used in this manual.

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. See “Alias nodes”.

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. See “Editions of Analytica”.

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. See “Editions of Analytica”.

Analytica Enterprise

An 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. See “Editions of Analytica”.

Analytica Professional edition

The standard, fully functional edition of Analytica. Analytica Pro provides all the features and functionality required to create, edit, and evaluate models. See “Editions of Analytica”.

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. See “Editions of Analytica”.

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. See “Introduction to arrays”.

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. See “Drawing arrows”.

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. See “Drawing arrows”.

Attribute

A property or descriptor of an object, such as its title, description, definition, value, or inputs. See “Managing attributes”.

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. See “The Attribute panel”.

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. See “Analyzing Model Behavior”.

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. See “Making a browse-only model and hiding definitions”.

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. See “Browse mode”.

Chance variable

An uncertain variable that cannot be directly controlled by the decision maker. It is usually defined by a probability distribution. A chance variable is depicted as an oval node. See “Classes of variables and other objects”.

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. See “Automatic checking for valid values”.

Class

The type of Analytica object: decision, chance, objective, or index variable; function; module; library; form; model. See “Classes of variables and other objects”.

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. See “Add a conditioning variable”.

Constant

A variable whose value is not probabilistic, and does not depend on other variables, such as the number of minutes in an hour. See “Classes of variables and other objects”.

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.” See “Parametric continuous distributions” and subsequent sections.

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 graphical 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. See “Cumulative probability”.

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, passwords, etc. See “DSN and data source”.

Decision variable

A variable that the decision maker can control directly. Decision variables are represented by rectangular nodes. See “Classes of variables and other objects”.

Definition

A formula for computing a variable’s value. The definition 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. See “Creating and Editing Definitions”.

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. See “Making a browse-only model and hiding definitions”.

Description

Text explaining what the node represents in the real system being modeled. It is limited in length to 32,000 characters. See “Attributes of a function”.

Deterministic table

A deterministic function that gives the value of a variable \mathbf{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 \mathbf{x} that corresponds to each combination of values of its inputs. See “Creating a determtable”.

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. See “Classes of variables and other objects”.

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. See “Introduction to arrays”.

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.” See “Parametric discrete distributions”.

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()**, **prohtable()**, and **determtable()** functions.

DSN

The Domain Service Name (DSN) provides connectivity to a database through an ODBC driver. The DSN contains the database name, directory, database driver, user ID, password, and other information. See “DSN and data source” on page 375.

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. See “Dynamic Simulation”.

Edit table

A definition that is a table is also called an edit table because it can be edited. See “Viewing an array as an edit table” and “Editing a table”.

Edit tool

A 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. The edit tool is in the shape of the normal mouse pointer cursor. See “Creating and editing nodes”.

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. See “Using Expressions”.

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. See “The Expression popup menu”.

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. See “Using filed modules and libraries”.

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. See “Using filed modules and libraries”.

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. See “Classes of variables and other objects”.

Graph

Format for displaying a multidimensional result. To view a result as a graph, click the **Graph** button. See also “Table.”

Graphing role

n aspect of a graph or chart used to display a dimension (or Index) of an array value. They include the horizontal axis, vertical axis, and key. See “Graphing roles”.

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.” See “Identifiers and titles”.

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. See “Importance analysis”.

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. See “What is an 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. See “Index selection”.

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. See “Classes of variables and other objects”.

Influence arrow

See “Arrow.”

Influence cycle

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. See “Influence cycle or loop”.

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. See “The Object window”.

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 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. See “Kurtosis(x)”.

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. See “Libraries”.

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. See “Create a list”.

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. See “Creating an index”.

Matrix

A two-dimensional array of numbers with indexes of equal length. See “Matrix functions”.

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. See “Probability density”.

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. See “Models”.

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. See “Classes of variables and other objects”.

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. See “Organizing a module hierarchy” and “Show module hierarchy preference”.

Multimodal distribution

A probability distribution that has more than one mode. See “How many modes does it have?”.

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. See “Classes of variables and other objects”.

Normal distribution

The bell-shaped curve, or Gaussian distribution.

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. See “Making a browse-only model and hiding definitions”.

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. See “Object Finder dialog”.

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. See "The Object window".

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. See "Classes of variables and other objects".

ODBC

Open Database Connectivity (ODBC) is a widely used standard for connecting to relational databases, on either local or remote computers, and issuing queries in Standard Query Language (SQL). See "Overview of ODBC" on page 374.

OLE Linking

A standard in the Windows operating system for sharing data between applications. See "Using OLE to link results to other applications".

Operator

A symbol, such as a plus sign (+), that represents a computational process or action such as addition or comparison. See "Operators".

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. See "The Outline window".

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 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. See "Analyzing Model Behavior".

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 p th 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

The bands that 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. See “Probability Bands option”.

Probability density function (PDF)

A graphical 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. See “Probability density”.

Probability distribution

A probability distribution describes the relative likelihood of a variable having different possible values. See “Probability distributions” and “Probabilistic calculation”.

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 (PDF),” which is used with continuous variables. See “Probability density”.

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. See “Protable(): Probability Tables”.

Protable

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.”

Quantile

See “Percentile.”

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. See “Array-reducing functions”.

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. See “Seeing remote inputs and outputs”.

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. See “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). See “Sampling method”.

Scalar

A value that is a single number.

Scatter plot

A graph that plots the samples of two probabilistic variables against each other. See “Scatter plots”.

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. See “Sensitivity analysis functions”.

Side effects

Changes to a global variable during evaluation other than those expected in its definition. See “Assigning to a local variable: $v := e$ ” on page 351.

Skewed distribution

A distribution that is asymmetric about its mean. A positively skewed distribution has a thicker upper tail than lower tail; and vice versa, for a negatively skewed distribution. See “Is the quantity symmetric or skewed?”.

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. See “Skewness(x)”.

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. See “Selecting, slicing, and subscripting arrays”.

Slicer

See “Slicers” on page 93.

SQL

Standard Query Language or SQL is a standard interactive and programming language for getting information from and updating a database. See “Accessing databases” on page 374.

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. See “Suffix characters”.

Symmetrical distribution

A distribution, such as a normal distribution, that is symmetrical about its mean. See “Is the quantity symmetric or skewed?”.

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. A definition that is a table is also called an **edit table**. In the **Result** window, click 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. See “Is the quantity symmetric or skewed?”.

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. The title can contain any characters, including spaces and punctuation. Compare to “Identifier.” See “Edit a node title”.

Uncertain value

See “Probvalue.”

Uniform distribution

A distribution representing an equal chance of occurrence for any value between the lower and upper values.

Units

The increments 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. Classes of variable include decision, chance, and objective. See “The Object window”.

Variance

A measure of the uncertainty or dispersion of a distribution. The wider the distribution, the greater its variance.

Alphabetical Index

Symbols

- (subtraction) operator 142
- # (dereference) operator 208, 364
- & (concatenation) operator 210
- * (multiplication) operator 143
- + (addition) operator 142
- .. (sequence) operator 166
- / (division) operator 143
- :: (scoping) operator 144
- := (assignment) operator 351
- < (less than) operator 143
- <= (less than or equal to) operator 143
- <> (not equal) operator 143
- = (equal) operator 143
- > (greater than) operator 143
- >= (greater than or equal to) operator 143
- \ (reference) operator 343
- ^ (exponentiation) operator 143

A

- About Analytica command 408
- Abs** () function 146
- abstraction, automatic 155
- Accept button 118
- Add Library command 332, 401
- Add Module command 332, 401
- Add Module dialog box 332
- Adjust Size command 75, 406
- Advanced Math command 404
- aliases, creating 55
- Align Selection to Grid command 76, 406
- All qualifier 343
- AnalyticaPlatform system variable 405
- AnalyticaVersion system variable 405
- Arccos** () function 213
- Arcsin** () function 213
- Arctan** () function 147
- Arctan2** () function 213
- Area** () function 186
- Argmax** () function 190
- arithmetic operators, meanings 142
- Array command 404
- Array library 404
- array qualifiers 342
- Array** () function 181, 182, 184
- arrays
 - abstraction 155
 - changing index of 183
 - defining 182
 - defining variables as 168

Index

- functions that create 165
 - huge 387
 - introduction 153
 - modeling 151–165
 - one-dimensional 320
 - operations on 155–159
 - three-dimensional 321
 - two-dimensional 320
 - value sources 155
 - values 25
- arrows
- arranging 74
 - arrow tool 21
 - automatically drawn 52
 - between modules 53
 - bold 332
 - creating 51
 - drawing 51–55
 - drawing between modules 56
 - dynamic 306
 - hiding 77
 - small arrow head 53
 - to and from indexes 154
- Asc 210
- Assignment Operator 351
- Attrib of Ident** function 329
- Attribute panel
- closing 24
 - using 23
- attributes
- creating new 328
 - displaying 328
 - displaying Check 124
 - in a definition 329
 - managing 327–329
 - of functions 327, 340
 - of modules 327
 - of variables 327
 - renaming 328
 - user-created 327
- Attributes command 403
- Attributes dialog box 124, 328
- Author attribute 327
- Average()** function 186
- B**
- background printing 26
- behavior analysis 40
- Bernoulli()** function 241
- Beta()** distribution function 255
- BetaFn()** function 222
- BetaI()** function 222
- Boolean
- number format 87
 - operators 144
 - values 142
 - variables 227
- Bring to Front command 77, 408, 409
- browse mode 21
- browse tool button 20
- button objects 19
- C**
- Cancel button 118
- Cascade command 408
- ceil()** function 146
- cells
- adding 172
 - copying and pasting 171
 - deleting 165, 172
 - editing 171
 - inserting 165
 - selecting 171
- Certain()** function 263
- chance variables 18
- Chancedist()** function 251
- Change identifier 59
- Check attribute
- displaying 124
 - features 327
- check value bounds 61
- check variable class 61
- ChiSquared()** distribution function 260
- Choice option 129
- Choice()** function 196
- Chr 210
- Class attribute 327
- class, changing for nodes 58
- Clear command 402
- Close command 401
- Close Model command 16, 17, 401
- Coerce qualifier 343
- colors
- changing 80
 - grouping nodes 80
 - in influence diagrams 80
 - input and output node 132
 - palette 80
- columns, separating 378
- Combinations()** function 223
- comments in definitions 117
- comparison operators 143
- computation time 398
- Concat()** function 201
- concatenation operators 210
- conditional dependencies 248

- conditional deterministic table 248
 - conditional operators 145
 - conditional probability tables 248
 - confidence intervals 398, 399
 - constants 19
 - Contact Lumina command 408
 - context qualifier 341
 - ContextSample qualifier 341
 - continuous distributions 227
 - conventions, typographic 9
 - Copy command 310, 402
 - Copy Diagram command 310, 402
 - Copy Table command 310, 402
 - Correlation()** function 278
 - cos()** function 147
 - cosh()** function 213
 - Created attribute 327
 - cross-hatching 119
 - Cubicinterp()** function 199
 - Cumdist()** distribution function 261
 - Cumipmt()** function 214
 - CumNormal()** function 223
 - CumNormalInv()** function 223
 - Cumprinc()** function 215
 - Cumproduct()** function 190
 - Cumulate()** function 190
 - Cumulative Probability command 35, 405
 - cumulative probability options 236
 - curve fitting, *see* **Regression** function
 - Cut command 402
 - cycle, influence 52, 427
 - cyclic dependencies 52, 427
- D**
- data
 - copying diagrams 310
 - identifying source 375
 - import/export format 319
 - importing and exporting 309–322
 - numerical 322
 - pasting from programs 310
 - pasting from spreadsheets 310
 - Database command 404
 - Database library functions 381
 - databases
 - configuring DSN 376
 - querying 374
 - writing to 379
 - Date number format 87
 - DBLabels()** function 381
 - DBQuery()** function 381
 - DBTable()** function 382
 - DBTableNames()** function 382
 - DBwrite()** function 379, 382
 - decimal number format 140
 - decision variables 18, 65, 74
 - Decompose** function 206
 - default view 30
 - Definition attribute 327
 - Definition button 20
 - Definition menu
 - overview 403
 - pasting from a library 123
 - definitions
 - adding identifiers 118
 - alphabetical list 421
 - changes to influence diagrams 119
 - comments in 117
 - creating 116–119
 - cross-hatching 119
 - description 116
 - editing 116–119, 123
 - exporting 318
 - hiding 385
 - hiding and unhiding 384
 - importing 318
 - incomplete 329
 - inheritance 385
 - invalid or missing 404
 - overview 340
 - special editing key combinations 117
 - syntax check 118
 - updating arrows 119
 - Degrees()** function 147
 - Delete Columns command 172
 - Delete Rows command 165, 172, 402
 - dependencies
 - conditional 248
 - cyclic 52, 427
 - with the **Dynamic()** function 305
 - dereference operator 364
 - Description attribute 327
 - descriptions 340
 - Determinant()** function 205
 - deterministic conditional tables 248
 - Determtable()** function 249, 250
 - determtables 248
 - Diagram menu 406
 - Diagram Style dialog box 81
 - Diagram window
 - description 17
 - maximum number of 335
 - diagrams
 - adding frames 135
 - adding graphics 135
 - adding text 135
 - copying 310

Index

- opening details 18
- organizing 75
- screenshots 83
- dimensions, modeling arrays and tables 152
- discrete probability distributions
 - creating 251
 - vs. continuous 227
- Distribution button 22
- Distribution command 404
- Distribution library 404
- distributions
 - exponential 257
 - logistic 258
 - lognormal 254
 - multivariate 265
 - symmetric vs. skewed 228
- Domain attribute 327
- Domain Service Name 375
- dot product 204
- DRIVER** attribute 375
- DSN
 - configuring 376
 - identifying data source 375
- Duplicate Nodes command 51, 402
- Dydx()** function 285
- dynamic arrows, showing or hiding 306
- dynamic models 74
- dynamic simulation 298–307
- Dynamic()** function 298–307

E

- Edit Definition command 116, 404
- Edit Icon command 133, 406
- Edit menu 402
- Edit Table buttons 22
- Edit Table window
 - copying 310
 - importing to 318
 - opening 171
 - viewing arrays 155
- Edit Time command 298, 404
- Edit tool button 20
- EigenDeComp()** function 206
- Elasticity()** function 286
- Email Tech Support command 408
- Erf()** function 223
- ErfInv()** function 223
- errors
 - evaluation 368, 414
 - factor 254
 - fatal 415
 - lexical 413
 - message types 413–415

- naming 414
- out of memory 415
- syntax 413
- Evaluate** function 368
- evaluation errors 368, 414
- evaluation mode qualifiers 341
- Exit command 17, 401
- Exp()** function 146
- Exponent number format 87, 140
- exponential distribution 257
- Exponential()** distribution function 256
- Export command 318–322, 401
- export format 319
- Expression popup menu 120, 169
- expressions
 - listing 120
 - parenthesis matching 117
 - syntax of 144
 - types 120
 - using 140–150

F

- Factorial()** function 147
- False system variable 142, 405
- fatal errors 415
- File Info attribute 327
- File menu 401
- filed libraries 59, 330
- Filed Library class 59
- Filed Module class 58
- filed modules 330
- files, changing locations 313, 317
- Find command 326, 403
- Find dialog box 326
- Find Next command 327, 403
- Find Selection command 327, 403
- Fixed Point number format 87
- Floor()** function 146
- For...Do** function 353
- Form class 59
- form modules 132
- fractiles 399
- frames, adding to diagrams 135
- Frequency()** function 279
- FunctionOf()** 331
- functions
 - about 19
 - attributes 340
 - built-in 177
 - creating 339
 - pasting 118
- Fv()** function 215

G

Gamma () distribution function 257
GammaFn () function 223
GammaI () function 223
GammaIInv () function 224
 Gaussian probability distributions 254
 generalized linear regression 292
Geometric distribution function 242
Getfract () function 277
 Graph Setup command 94, 405
 Graph setup dialog box 94
 graphics, adding to diagrams 135
 graphing roles, about 91
 graphs

- displaying 30, 31
- exporting 310
- scatter 102, 291
- X-Y 105, 289

 grid, aligning to 76

H

hardware specifications 410
 Help attribute 327
 Help menu 408
 hidden definitions

- creating 385
- inheritance 385
- setting 403
- unhiding 384

 Huge Arrays, overview 387
Hypergeometric distribution function 242
 hyperlinks, model documentation 137

I

icons, adding to nodes 133
Ident (I=U) function 194, 196
Ident (Time-n) function 300
 identifiers

- changing 59
- naming 414
- overview 118, 340

 Identifiers attribute 327
 IgnoreWarnings 369
 Import command 318–322, 401
 import format 319
 Index button 173
 Index qualifier 342
 indexes

- changing on arrays 183
- creating 161, 162, 170
- defining 381
- description 154
- dialog box features 170

- displaying arrows 154
- in diagrams 154
- modeling 151–165
- recognizing nodes 19
- removing from tables 170
- selection area 29
- summing over 157
- using in OLE linking 313

IndexesOf () function 202
IndexNames () function 202
 INF 141
 infinity 141
 influence arrows, *see* arrows
 influence cycle 52, 427
 influence diagrams

- copying 310
- decision variables 65
- definition changes 119
- editing 73–78
- overview 17

 input nodes, using 128–130
 Inputs attribute 327
 inputs, remote 18
 Insert Columns command 402
 Insert Rows command 165, 402
 Integer number format 87, 140
Integrate () function 192
 intellectual property, protecting 384
 interpolation functions 199
Invert () function 205
Ipmt () function 216
Irr () function 216
Isnan () function 148
Isnumber () function 148
Istext () function 148
Isundef () function 148
Iterate function 356
J

Join () function 190

K

key combinations for editing 117
 key icon 23
 Knuth random number generator 235
Kurtosis () function 276

L

L'Ecuyer random number generator 235
 labels

- displaying 164
- listing 120

 Last Saved attribute 327

Index

lexical errors 413

Lgamma() function 214

libraries

adding to a model 331

Array 404

Array Functions 202

creating 345

custom 404

Database 381

Distribution 230, 404

Distribution System 121

Distribution Variations 12, 265

filed 59, 330

Financial 219

Generalized Regression 14

Math 404

matrix 404

Multivariate Distributions 12, 265

Operators 404

optimizer 404

Performance Profiler 14, 391

removing from a model 331

saving 332

Special 404

statistical 404

Text 210

text functions 404

Trash 51

user 403

user-defined functions 345

using 346

Library class 58

linear regression 292

Linearinterp() function 200

List buttons 22, 129

lists

creating 40, 162

displaying 164

editing 165

navigating 165

numbers 164

Sequence option 163

vs. lists of labels 164

Ln() function 146

logical operators 144

logical values 142

logical variables 227

Logistic() distribution function 258

Lognormal() distribution function 254

Logten() function 146

M

m to n sequence 166

magnification, printouts 25

Make Alias command 55, 403

Make Importance command 283, 403

Make Input Node command 129, 403

Make Output Node command 131, 403

Math command 404

Math library 404

Matrix command 404

matrix functions 203–205

Matrix library 404

Matrix multiplication 204

Matrix() function 204

MDArrayToTable() function 197, 379

MDTable() function 198

Mean Value command 33, 405

Mean() function 275

median Latin hypercube sampling method 234

memory

Memory Usage window 410

requirements 410

usage 408

MemoryInUseBy 393

menus

command descriptions 401–409

right mouse button 409

Mid qualifier 341

Mid Value command 32, 405

Mid() function 280

Min() function 186

Minimal Standard random number generator 235

Mod() function 147

mode 228

Model class 58

models

building 64

closing 16

combining 333

creating 48

defined 16

defining 48

documentation 69

dynamic 52, 74

editing 48–57, 327

expansion 69

hyperlinks 137

integrated 333

modular 334

navigating 325

obfuscating 384

opening 16

opening details 22

protecting intellectual property 384

saving 332

separating columns 378

- switching 17
 - testing 67
 - using in XML format 135
 - viewing details 18
 - Module class 58
 - modules
 - about 19
 - filed 58, 330
 - hierarchy 324
 - linking 386
 - organizing hierarchy 79
 - Monte Carlo sampling method 234
 - Move Into Parent command 406
 - MsgBox** function 369
 - multivariate distributions 265
- N**
- naming errors 414
 - NAN 141
 - natural cubic spline 199
 - New Model command 401
 - Node Style dialog box 82
 - nodes
 - adding icons 133
 - aligning 76
 - arranging 74
 - changing class 58
 - changing size 51
 - consistent sizes 74
 - creating 49
 - creating aliases 55
 - customizing 82
 - default size 82
 - duplicating 51
 - editing title 49
 - grouping related 79
 - identifying types 18
 - in fonts 82
 - selecting 19
 - shape descriptions 18
 - text node type 135
 - title characteristics 73
 - undefined 61, 84
 - visual grouping 80
 - Z-order 77
 - Nonnegative qualifier 343
 - Normal()** distribution function 254
 - Normalize()** function 193
 - Nper()** function 216
 - Npv()** function 216
 - Number Format command 405
 - Number qualifier 343
 - numbers
 - combining with text 164
 - formats 86, 140
 - lists of 164
 - numerical data formats 322
- O**
- obfuscated copies 384
 - obfuscated models, linking 386
 - object attributes, reading 329
 - Object button 20
 - Object Finder dialog box 121
 - Object menu 403
 - Object window
 - maximum number of 335
 - opening 22
 - using 22
 - objective variables 19, 64, 74
 - ODBC 374
 - OLE linking
 - activating other applications 317
 - auto recompute links 61, 313
 - automatic vs. manual updates 313, 317
 - changing file locations 313, 317
 - linking data from Analytica 311–314
 - linking data into Analytica 314–317
 - number formatting 313
 - OLE Links command 402
 - Open Source button 317
 - Paste Special dialog 316
 - procedure, from Analytica 311
 - procedure, to Analytica 314
 - refreshing links 314
 - table example 314
 - terminating links 317
 - using indexes 313
 - one-dimensional array format 320
 - Open Database Connectivity 374
 - Open Model command 17, 401
 - Open Source button 317
 - operators
 - arithmetic 142
 - Boolean 144
 - comparison 143
 - conditional 145
 - logical 144
 - scoping 144
 - text concatenation 210
 - Operators command 404
 - Operators library 404
 - Optimizer command 404, 408
 - Optimizer library 404
 - order of precedence 144
 - ordering qualifiers 344

Index

- Outline button 20
- Outline window 325
- output nodes
 - using 131
 - viewing values 22
- Outputs attribute 327
- outputs, remote 18

P

- page breaks 408
- palettes
 - color 80
 - Result 28
- parameter qualifiers 340
- Parameters attribute 327
- parameters, overview 340
- parametric analysis 40
- parent diagram
 - returning to 23
 - viewing 18
- Parent Diagram button 20
- parenthesis matching 117
- Paste command 310, 402
- Paste Identifier command 121, 404
- Paste Special command 402
- Percent number format 87
- percentiles 277
- Permutations()** function 223
- Pi system variable 405
- Pmt()** function 217
- Poisson()** distribution function 242
- popup menus
 - creating 129
 - using 22
- PositionInIndex()** function 188
- Positive qualifier 343
- Ppmt()** function 217
- precedence, order of 144
- precision 410
- Preferences command 59, 336, 402
- Preferences dialog box 126, 324, 336
- Print command 25, 319, 401
- print options
 - magnification 25
 - multiple windows 26
 - page preview 25
 - printing to files 319
 - scaling 25
 - setting 25
- Print Preview command 401
- Print Report command 26, 401
- Print Setup command 25, 401
- printing options

- background 26
- Prob qualifier 341
- Prob Table button 247
- Probability Bands command 34, 405
- probability bands, settings 236
- Probability Density command 34, 405
- probability density options 236
- probability distributions
 - beta 255
 - Chi-squared 260
 - choosing 226–229
 - computing 240
 - continuous 227
 - defining a variable as 229
 - discrete 227, 251
 - functions 239–260
 - Gaussian 254
 - normal 254
 - triangular 253
 - truncating 263
 - uniform 252
- Probability Mass command 34, 405
- Probability Table command 246
- probability tables 246–248
- Probability()** function 277
- Probbands()** function 277
- Probdist()** distribution function 262
- Prohtable()** function 248
- prohtable 246–248
- Probvalue attribute 327
- Product()** function 187, 188
- profiling
 - time 391
- Pv()** function 217

Q

- qualifiers
 - All 343
 - array 342
 - Coerce 343
 - context 341
 - ContextSample 341
 - evaluation mode 341
 - Index 342
 - Mid 341
 - Nonnegative 343
 - Number 343
 - ordering 344
 - parameter 340
 - Positive 343
 - Prob 341
 - Reference 343
 - Sample 341

- Scalar 342
 - Text 343
 - type checking 343
 - Unevaluated 343
 - Variable 342
 - quantiles 277
- R**
- Radians()** function 147
 - random Latin hypercube sampling 234
 - random number methods 235
 - random seed 235
 - Rank()** function 193
 - Rankcorrel()** function 279
 - Rate()** function 217
 - Recent files 401
 - Recomputing results 30
 - reducing functions 185
 - Reference qualifier 343
 - Register command 408
 - Regression()** function 292
 - remote variables 18
 - resampling 307
 - Resize Centered command 51, 406
 - Result button 20
 - result graphs, exporting 310
 - Result menu 405
 - result tables
 - copying 310
 - getting data 382
 - retrieving 378
 - Result tool palette 28
 - Result window
 - default view 30, 60
 - maximum number of 59, 336
 - table view 30
 - working with 28–30
 - results
 - comparing 36
 - recomputing 30
 - viewing 17
 - Round()** function 147
 - Run system variable 170, 231, 280, 405
- S**
- Safe Intermediates 61
 - Sample command 405
 - Sample qualifier 341
 - sample size
 - description 233
 - selecting 398
 - setting 233
 - sample()** function 280
 - Samplesize system variable 233, 280, 405
 - sampling methods
 - choosing 234
 - median Latin hypercube 234
 - Monte Carlo 234
 - random Latin hypercube 234
 - selecting 233
 - Save A Copy In command 332, 401
 - Save As command 332, 401
 - Save command 332, 401
 - Scalar qualifier 342
 - scalar variables 152
 - scale, printouts 25
 - scatter plots 102, 291
 - Scoping operator 144
 - scoping operator 144
 - screenshots, taking 83
 - sdeviation()** function 275
 - Select All command 76, 402
 - Self, probability tables 247
 - Send to Back command 77, 409
 - sensitivity analysis 284–286
 - sequence operator 166
 - Sequence option 163
 - Sequence()** function 166
 - Set Diagram Style command 81, 406
 - Set Node Style command 82, 406
 - shells, stand alone 334
 - Show By Identifier command 118, 403
 - Show Color Palette command 80, 406
 - Show Invalid Variables command 329, 404
 - Show Memory Usage command 408, 411
 - Show module hierarchy 61, 324
 - Show Page Breaks command 25, 408
 - Show Result command 405
 - Show result warnings 61
 - Show undefined 61
 - Show With Values command 326, 403
 - ShowProgressBar** function 371
 - Size()** function 203
 - skewed distributions 228
 - Skewness()** function 276
 - slice()** function 195
 - slicers, about 93
 - Snap to Grid command 76, 406
 - software specifications 410
 - Special command 404
 - Special library 404
 - specifications 410
 - SplitText()** function 211
 - SQL
 - accessing databases 374
 - case sensitivity 377
 - retrieving result tables 378

Index

- specifying queries 377
- SqlDriverInfo()** function 382
- Sqr()** function 146
- Sqrt()** function 147
- Standard Query Language 374
- Statistical command 404
- Statistical library 404
- Statistics command 33, 405
- Statistics()** function 280
- statistics, setting 235
- Stepinterp()** function 200
- string, *see* text values
- StringLength()** function 210
- StudentT()** distribution function 258
- Subindex()** function 187, 188
- Subscript** function 194
- Suffix number format 87, 140
- Sum()** function 157, 185
- symmetrical distributions 228
- syntax
 - checking in definitions 118
 - errors 413
- system constants 141
- System Variables submenu 404

T

- Table()** function 182, 183
- tables
 - adding cells 172
 - copying 310
 - copying and pasting cells 171
 - creating 168–170, 182
 - defining variables as 168
 - deleting cells 172
 - deterministic conditional 248
 - displaying 30
 - editing 171–??
 - editing cells 171
 - import/export data format 319
 - lookup 200
 - modeling 151–165
 - numerical data formats 322
 - removing indexes 170
 - selecting cells 171
- Tan()** function 147
- terminology 421
- text
 - adding to diagrams 135
 - combining with numbers 164
 - joining 210
 - values 142
- text concatenation operators 210
- Text functions command 404

- Text functions library 404
- Text qualifier 343
- three-dimensional array format 321
- Tile Horizontally command 408
- Tile Vertically command 408
- time profiling 391
- Time system variable 298, 405
 - defining 300
 - description 170
 - details 300–303
 - modeling changes 298
 - using in a model 303
- Title attribute 327
- titles
 - attribute description 340
 - characteristics 73
 - editing 49
- today()** function 371
- transformed beta distribution 255
- transforming functions 190–??
- Transpose()** function 207
- Triangular()** distribution function 253
- True system variable 142, 405
- Truncate()** distribution function 263
- Tutorial command 408
- two-dimensional array format 320
- type checking qualifiers 343
- typographic conventions 9

U

- uncertainty factor 254
- Uncertainty Sample option 233
- Uncertainty Setup command 406
- Uncertainty Setup dialog box 232
- Uncumulate()** function 193
- undefined nodes 61
- Undo command 57, 402
- Unevaluated qualifier 343
- Unhide Definition(s) command 403
- Uniform()** distribution function 252
- Unique()** function 168
- Units attribute 327, 340
- Update License command 408
- Use Return to enter data 61
- user libraries 345, 403
- user-created attributes 327
- user-defined functions 337–346

V

- Value attribute 327
- values
 - arrays 25
 - checking bounds 61

- checking validity 124–126
- disabling checking 126
- listing 120
- text 142

Variable qualifier 342

variables

- automatic renaming 59
- chance 18
- class checking 61
- defining as arrays 168
- description 17
- finding 326
- general 19
- invalid 329
- objective 19
- public 333
- remote 18
- scalar 152

Variance() function 275, 277

W

Warning icon 118, 413

warnings, *see* errors

Web Tech Support command 408

weibull() distribution function 259

whatif() function 286

while...do function 355

Window menu 408

windows

- browsing 21
- managing 335
- numbers of 59
- print settings 26
- see also* Diagram window, Object window, Outline window

Windows system software 410

writeTableSql() function 380

X

xirr() function 218

XML format, using models 135

xnpv() function 218

XY button 108, 289

X-Y results 105, 289

Z

Z-order, nodes 77

Index

Analytica windows and dialogs

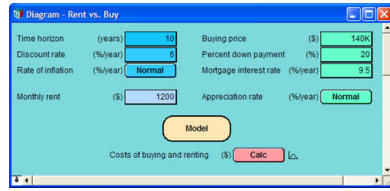


Diagram Window:
Inputs and Outputs

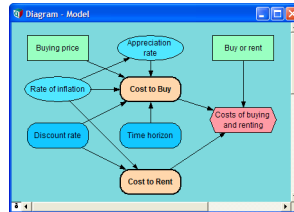
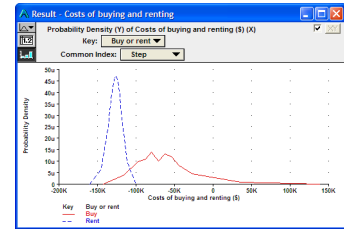
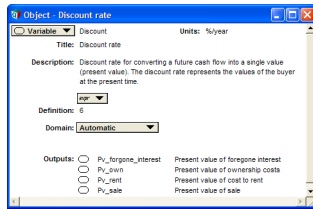


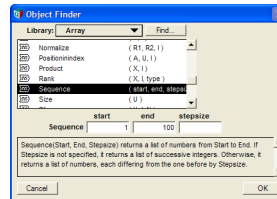
Diagram Window:
Influence Diagram



Result Window — Graph View



Object Window



Object Finder

Step	Totals
Buy	-187.1K
Rent	-159.8K
Totals	122.5K

Result Window — Table View

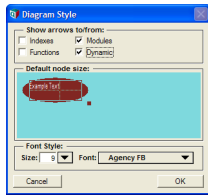
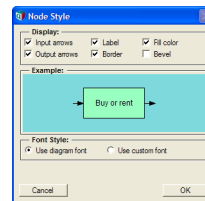
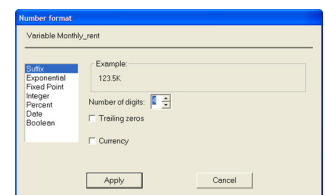


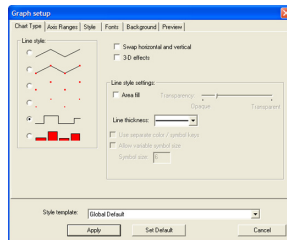
Diagram Style Dialog



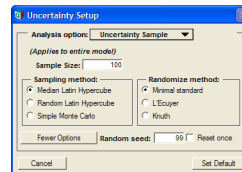
Node Style Dialog



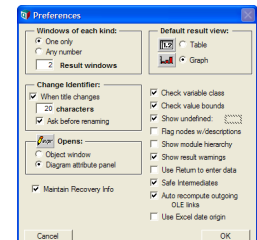
Number Format Dialog



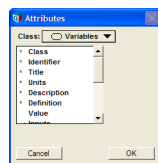
Graph Setup Dialog



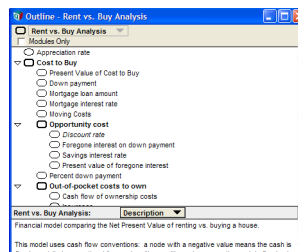
Uncertainty Setup Dialog



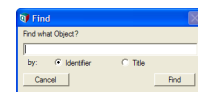
Preferences Dialog



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
			-2	%	percent
3	K	Kilo	-3	m	milli
6	M	Mega or Million	-6	μ	micro (mu)
9	G	Giga	-9	n	nano
12	T	Tera or Trillion	-12	p	pico
15	Q	Quad	-15	f	femto

Tip If integer or fixed point is selected, numbers larger than 10^9 display in exponential format.