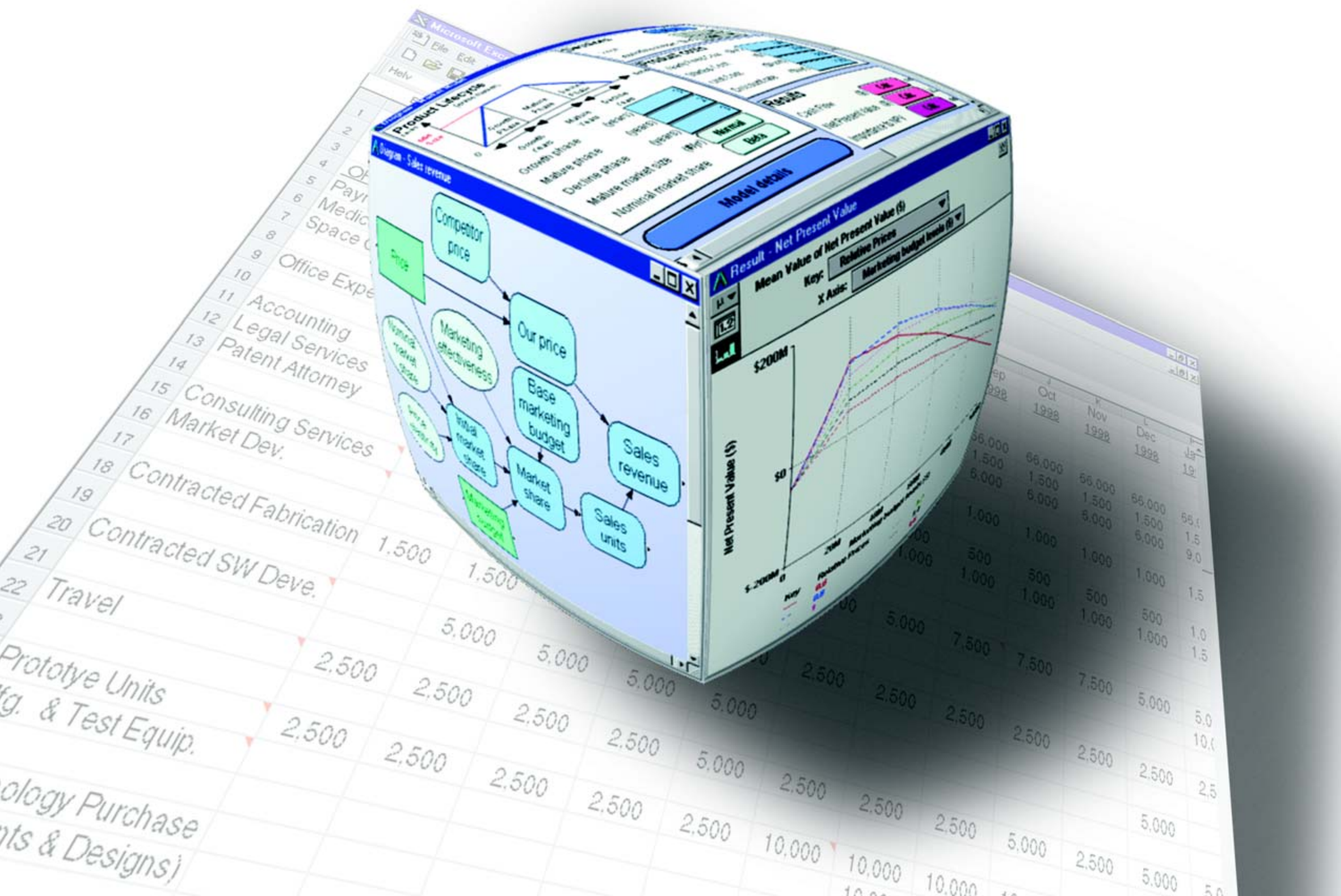
analytica®
Beyond the Spreadsheet

User Guide

Analytica 4.1

May 7, 2008



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

Copyright Notice

Credits

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

Contents

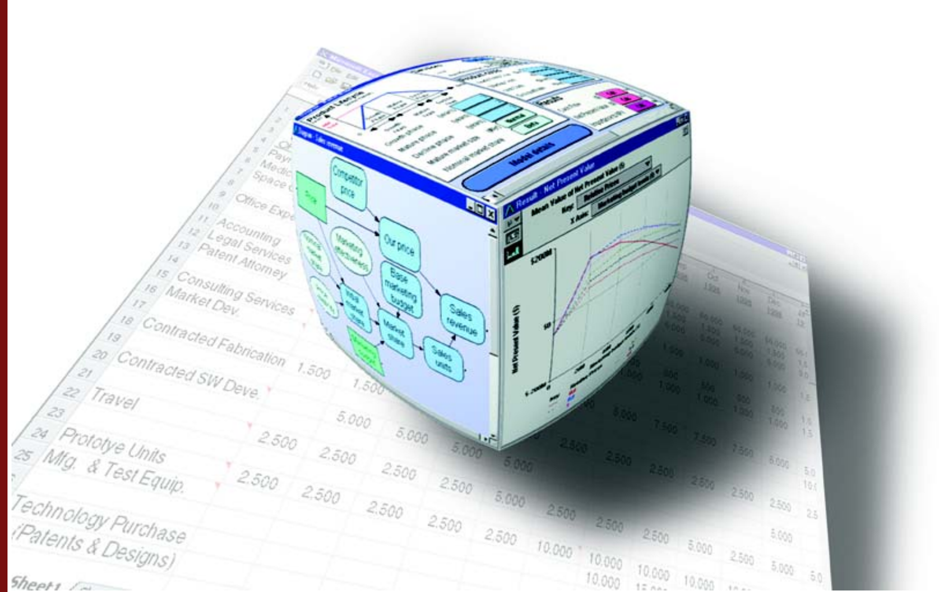
Contents

Introduction

About Analytica

This introduction explains:

- How to use this manual
- How to install Analytica
- Editions of Analytica
- The online help system
- Typographic conventions used in this guide
- How to access Analytica example models
- What's new in Analytica release 4.0
- What's new in Analytica release 4.1



This *User Guide* describes how to use Analytica 4.1. 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 might then want to read a few sections of the *User Guide* listed in the next section to learn more key concepts. You can consult the rest of this guide as a reference when you need more depth.

Tip For the most current information on Analytica, visit Anawiki (the Analytica Wiki online at http://www.lumina.com/wiki). This site includes tips, libraries, and reference materials, along with a search feature.

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* (might 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 might 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. See page 61.

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. See page 69.

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 might 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. See page 143.

Chapter 14: 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. See page 219.

Chapter 21: 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. See page 325.

Hardware and software requirements

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

- Intel 486-66 MHz or better (Pentium 500 MHz+ or AMD Athlon recommended).
- 30 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 4.1 installer from http://www.lumina.com, or inserting the Analytica CD-ROM into your CD or DVD drive, just double-click the installer to start installation. It installs 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 leaves it there, so you can run either version.

The setup program asks you to confirm the directory name in which to install Analytica, by default, C:\Program Files\Analytica 4.1. Most users can accept the default.

License codes

You need a license code to activate the software. Lumina emails 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 might 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

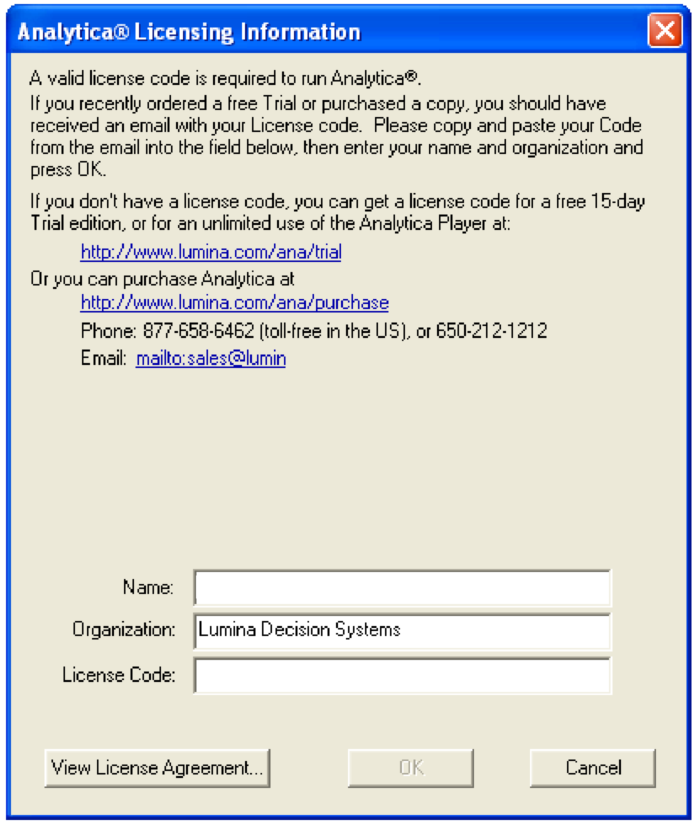
Each license code goes ***stale*** a few days after it is generated. If yours is stale — perhaps, because you didn't install Analytica right away, or, later, if you want to move Analytica onto another computer — fear not! Click the URL on the registration screen, or go to http://www.lumina.com/ana/stale. 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 can still 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 might 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.



Tip 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 patch release When you upgrade a licensed copy with a patch release (e.g., 4.1.0 to 4.1.1), simply run the installer. The fields you entered originally will be filled in, which you can leave unless you wish to make changes. The installer replaces the older release and reuses your existing license code.

To upgrade to a minor or major release You can install Analytica 4.1 and retain an earlier release, such as Analytica 3.1, on your computer. You need a new license code for the new release. The installer gives you the option of cleanly uninstalling the earlier release(s) if any are installed on your computer.

To uninstall Analytica After confirming that Analytica 4.1 is working, you usually uninstall the earlier release. To uninstall the earlier 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 the **Remove** button to start the Wizard.
4. Follow the steps through the uninstall wizard.

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.

Enterprise Offers all the features of Analytica Professional, plus support for 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.

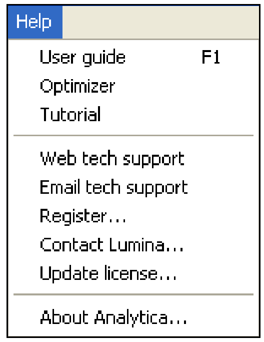
The Analytica Decision Engine (ADE) ADE runs Analytica models on a server computer. It provides an application programming interface (API) to provide access to view, edit, and run models from another application, including a web server. You can create a user interface to models via a web browser, so that many end users can view and run a model via the Internet. You 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.

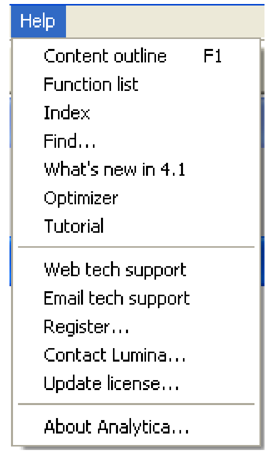
Compare Analytica features by edition

Features	Editions of Analytica					
	Player	**Power Player**	**Trial**	**Professional**	**Enterprise**	**ADE**
Open models, change inputs, and view results	✓	✓	✓	✓	✓	✓
Save model with changed inputs		✓	✓	✓	✓	✓
Create and edit models			✓	✓	✓	✓
No marking of printout		✓		✓	✓	✓
Hierarchical influence diagrams	✓	✓	✓	✓	✓	
Monte Carlo uncertainty analysis	✓	✓	✓	✓	✓	✓
Intelligent Arrays, see page 143	✓	✓	✓	✓	✓	✓
Procedural programming, see page 325			✓	✓	✓	✓
OLE linking with Excel, see page 292	✓	✓	✓	✓	✓	
Outline Window, see page 304	✓	✓	✓	✓	✓	
Create input and output controls and forms, see page 120			✓	✓	✓	
General function libraries: Math, Array, Distributions, Special, Statistical, Text	✓	✓	✓	✓	✓	✓
Advanced function libraries: Advanced math, Financial, and Matrix	✓	✓	✓	✓	✓	✓
Save browse-only models and hide sensitive model details, see page 360					✓	✓
Huge Arrays™ — dimension up to 100 million, see page 362		✓			✓	✓
ODBC database access, see page 358		✓			✓	✓
Time and memory profiling, see page 366		✓			✓	✓
Optimizer available		✓			✓	✓
Application programming interface (see *ADE User Guide*)						✓

Help menu and electronic documentation

Select **Help** from the menu bar to open the **Help** menu.





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 in Adobe Acrobat so you can search for a term.

What's new in 4.1? Opens "What's new in Analytica 4.1?" 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"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 with Lumina contact information: web links, phone numbers, email, and physical mailing address.

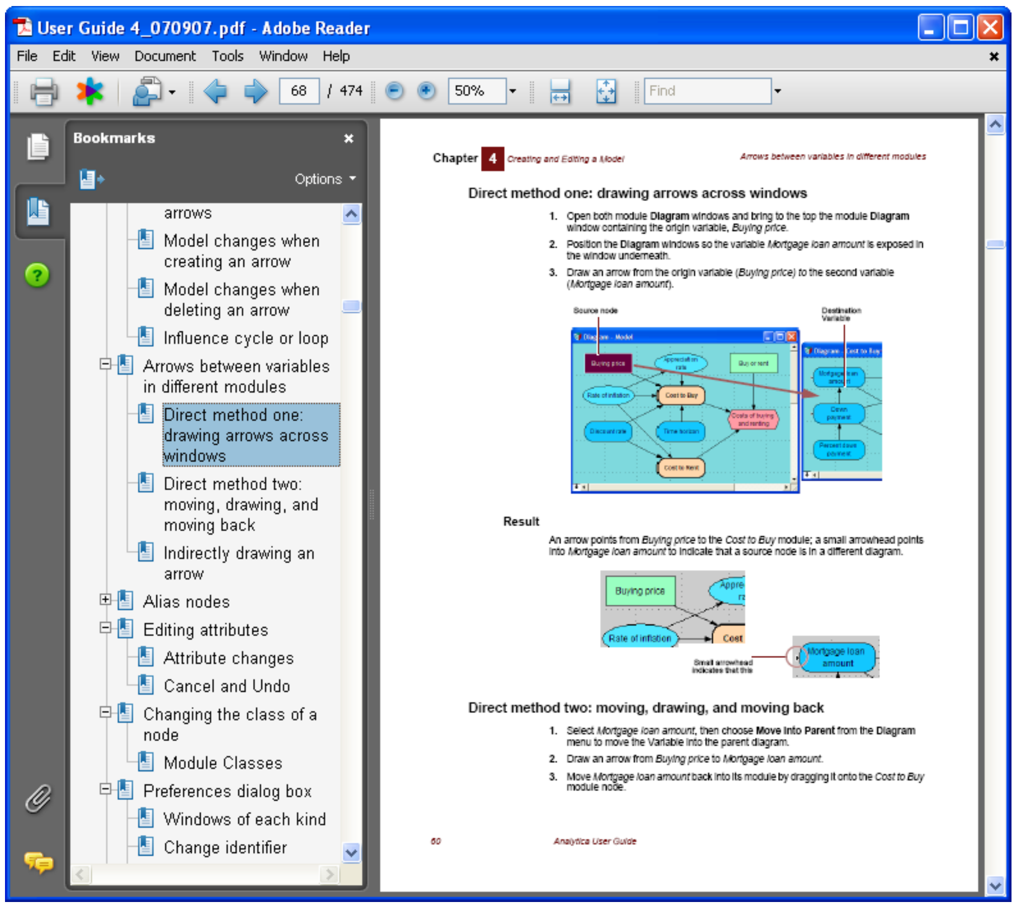
Update license Opens the **Licensing Information** dialog so you can review your or enter a new license code or enter a new code to upgrade your copy of Analytica.

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 collapse the bookmark tree for chapters and sections of the outline.

Function list If you can't remember the name of a function, go to the Function List, after the appendixes. This chapter 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 *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
behavior analysis	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.
Enter, Control+a	A key or key-combination on the keyboard. A letter, such as "a", can 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 **N** 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"page 49). You would enter the expressions, **[1, 2, 3, 4, 5]** and **N^2** into their definitions (see "Creating or editing a definition"page 108). You would *not* enter the assignment "**:=**". The last line says that the expression **Sum(Squares, N)** 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**

  Squares ▶

	1	2	3	4	5
	1	4	9	16	25

- A two-dimensional array

  Index_a ▼, **Index_b** ▶

	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 4.1?

Here are highlights of new and improved features in Analytica 4.1 added since the release 4.0. Additional detail can be found on the Analytica Wiki (at http://www.lumina.com/wiki).

User Guide

A nice introduction to Intelligent Arrays has been added to Chapter 11, "Arrays and Indexes" on page 143. Intelligent Arrays and intelligent array abstraction are seen by many users as the most useful features of Analytica, and we hope this new introduction will help you gain mastery of this powerful functionality.

Installer

Analytica 4.1 has an all-new installer. It provides the option of cleanly uninstalling previous releases while upgrading, provides cleaner upgrades and uninstalls, and runs much faster.

User Interface

Images When you copy/paste a bitmap image into a diagram, the image is automatically compressed into a PNG format to reduce memory and model file size (page 126). The new **Change Picture Format** dialog, accessible from the **Diagram** menu, allows you to convert the internal format to other available formats, including legacy bitmap if it needs to be viewed in Analytica 4.0 or earlier (page 126).

32-bit bitmap images, and images with transparency and alpha blending, are now supported.

Bitmap images draw much more quickly.

Diagrams Diagrams can be exported directly to an image file in PNG, EMF, JPEG, BMP, or TIF formats (page 292).

The **Set Diagram Size** option on the **Diagram** menu has been eliminated. It was never very useful.

Graphs and charts When you click a data point, a balloon shows the full coordinates of that point at full precision.

Autoscaling is improved when the other axis is manually scaled.

Time formats Using a custom format, the time of day (page 209) can now be included as part of a date number format. The fractional part of an integer indicates the time as a fraction of a full 24-hour day.

Make Importance option The **Make Importance** option (page 268) creates two nodes: one to hold the uncertain inputs and one to hold the importance result. Both are now variable nodes, while previously an index node was used for the inputs. The change was made because an index should contain a one-dimensional result, which is not the case for the inputs. Hence, a variable node is more consistent.

Windows Vista A quick animation when windows are opened has been removed. This animation usually lasts for about 1/10 of a second, but in the Vista Aero color scheme was taking over 10 seconds.

File saving File **Open** and **Save** dialogs are now resizable.

If you haven't set the title of your model, it is set automatically to the file name when saved.

Excel Integration

Copy/paste Cell ranges containing multi-line cells (where one cell contains multiple lines of text) can now be copied and pasted in both directions.

OLE linking You can OLE link a named range of cells in an Excel spreadsheet into Analytica (see page 292). This has the advantage of this is that the Analytica model can automatically adapt when the number of rows or columns in the named range changes without having to adjust the link itself. OLE linking from a named range to any other application, even within Microsoft Office itself, doesn't work in general due to a bug in Excel (at least in Excel 2003), but Analytica now successfully works around that bug.

Reading Excel files The Enterprise Edition now includes three new functions that read directly from an Excel workbook file: **OpenExcelFile**, **WorksheetCell**, and **WorksheetRange**. These are still considered experimental in release 4.1, so they don't yet appear on the database **Definition** menu and are not documented in this guide. Consult the Analytica Wiki for documentation on these functions (http://lumina.com/wiki/index.php/Functions_To_Read_Excel_Worksheets).

Expressions

New functions **TextTrim(t)** (page 206) removes leading or trailing spaces.

MakeTime(h,m,s) (page 208) returns the fraction of a day represented by a given time.

ComputedBy(X) (page 329) indicates that the value of a variable is computed as a side-effect during the evaluation of another variable, **X**.

OpenExcelFile, **WorksheetCell**, and **WorksheetRange** read values directly from an Excel workbook file. These require Analytica Enterprise and are documented on the Analytica Wiki.

IsResultComputed(X) (page 349) tests whether the result for **X** is already cached.

BesselJ(x,n), **BesselY(x,n)**, **BesselI(x,n)**, and **BesselK(x,n)** compute Bessel functions. See page 210.

Enhancements to Existing Functions Functions **Sum**, **Min**, **Max**, **Average**, and **Product** now allow multiple indexes to be specified in a single call when reducing across multiple indexes. For example, **Sum(X,I,J,K)**.

Functions **Sum** (page 187), **Min** (page 188), **Max** (page 187), **Argmin** (page 188)**, Argmax** (page 188)**, Average** (page 187), **Product** (page 187), **JoinText** (page 207), **Irr** (page 212), and **Npv** (page 212) now ignore any null-valued cells in the array parameter, performing their respective operation only over the non-null values. **Regression** (page 278) ignores any data points having a null **Y** value. There is also a new parameter data-type qualifier, **OrNull** (page 320).

Functions **Sum**, **Min**, and **Max** now accept an optional **ignoreNonNumbers** parameter (page 186). For example, **Min(X,I,IgnoreNonNumbers:True)** finds the minimum among the numeric values of **X**, ignoring textual or other values.

Functions **Sum**, **Min**, **Max**, **Average**, and **Product** accept an optional **ignoreNaN** parameter (page 186).

Function **Round** (page 137) accepts a second optional parameter, **Round(x,digits)**.

DatePart (page 208) has been extended with several new options for accessing the time fields (hours, minutes, and seconds), and the number of elapsed weeks or week days from either the date origin or the beginning of the year. The time field offsets (hours, minutes, seconds) are accepted by the **DateAdd** function (page 209).

The **Today** function (page 209) can optionally return the current time as well as date, and also optionally return the date/time in coordinated universal time (utc) rather than in the local time zone.

The **Irr** function (page 212) function finds a solution more reliably when a solution exists.

StudentT (page 247) now supports Latin hypercube sampling.

Shuffle (page 251) accepts an optional **Over** parameter (for independent random shuffles).

Concat (page 197) accepts scalars for its first two parameters when its index is specified. Also, the result index, **K**, is now optional.

When **x** is array-valued, the result of **Elasticity(y,x)** (page 271) is now different, being now more consistent with the definition of **Elasticity** being the percent change in **y** when **x** changes by 1%.

DbTable (page 358) now returns **Null**, rather than **NaN**, when a numeric value is missing in the relational table.

When **Random** (page 252) is evaluated in a sample context, the parameters of the distribution are now also evaluated in sample context.

Meta-indexes

Suppose you want a global index to contain a list of identifiers. Analytica's default evaluation rules with evaluate those variables when the value of the index is requested. In some cases, you might really want just a list of handles (i.e., the index is a "meta-index"). The **metaOnly** attribute (page 344) can be set to 1 on a global index object that is defined as a list of identifiers to force this behavior.

What's new in Analytica 4.0?

These are highlights of new and improved features introduced into release 4.0 since Analytica 3.1.

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** (page 89) 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.

Graph style templates (page 96) let you apply and reuse a collection of graph settings, for a consistent style for a model or your entire organization.

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

XY comparison (page 98) now lets you plot one slice against another slice of an array variable over the Comparison index, as well as one variable against another.

Tables

In graphs or tables, you can reorder slicer indexes (any graph indexes not shown on horizontal axis or key) simply by dragging them (page 89).

You can create smarter end-user interfaces by putting **drop-down** menus in cells of an edit table, using **Choice()** (page 176) to let end users select from a list of options. When viewing a table, using Find from the Object menu (*Control+f*) (page 177) lets you search for selected text.

The new **SubTable** (page 202) 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.

Smart **table splicing** (page 173) 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 U.S.-centric. Number formats (page 82) 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** (page 209), **DatePart** (page 208), and **Today** (page 209) add flexibility for computing dates.

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 (page 177) 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 (page 72). 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.

As 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 (page 236) 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**.

New functions **Random()** (page 252) generates single random sample from any distribution. **Shuffle(a, i)** (page 251) randomly shuffles an array. **PDF(X) and CDF(X)** (page 267) 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.

Over parameter You can create an array of independent probability distributions over one or more indexes by adding optional **Over parameter** (page 253) to a univariate probability distribution, e.g., **Normal(0, 1, Over: i, j)**.

Extended functions **Lognormal** (page 243) uses *mean* and *stddev* (standard deviation) as an alternative to *median* and *gsdev* (geometric standard deviation). **Truncate(x, min, max)** (page 251) accepts min and max threshold parameters and preserves sample ordering, and hence rank correlations. **Uniform(min, max, integer)** (page 241) adds the optional parameter **integer** to specify that values be integers in the range. **CumDist(p, r, i, smooth)** (page 249) adds an optional **Smooth** parameter to control interpolation.

Uncertain parameters Many distribution functions are much faster, especially when their parameters are uncertain (hierarchical distributions). **Gamma** (page 218), **Binomial** (page 233), **GammaInv** (page 218) are more accurate for extremely large or small parameter values.

Multivariate Distributions library New distributions include **MultiUniform** (page 256) and **UniformSpherical** (page 256), generalized **Dist_reshape** (page 255), functions for creating time series with serial correlations, and uncertainty about regression coefficients.

Distribution Variations library New distributions include **Smooth_fractile**, **Warp_dist**, **Erlang**, **Pareto**, **Rayleigh**, **Lorenzian**, **NegBinomial**, **InverseGaussian**, and **Wald**.

Running index for statistics By default, the running index defining which dimension statistical functions operate over is **Run** (page 100), the index over random samples. You can 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)** (page 263) 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** (page 257) 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.

Weights for statistics By default, statistics functions use **SampleWeighting** (page 257) when you are using importance sampling. You can also provide an optional parameter **W** to a statistical function to specify a non-default set of weights. For example, **Mean(x, w: x > 0)** (page 263) 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 (page 167), e.g., **X := [A, B]**, Analytica 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 (page 23). 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 (page 331) is an array with the same index(es) as **x**.

You can now subscript an expression (page 174), 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(X, J, 2)**. **PositionInIndex(a, u, i)** gives the position **n** in index **i** for which **a[@i=n] = u** (see "@: Index Position Operator"page 190).

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"page 330).

Argmin and Argmax The new **Argmin(x, i)** (page 188) and existing **ArgMax(x, i)** (page 188) can both now work over multiple indexes, and return the value or position of the indexes containing the minimum or maximum value.

Trig functions We have added the inverse trigonometric functions **ArcCos**, **ArcSin**, and **ArcTan** (page 209), and hyperbolic functions **CosH**, **SinH**, and **TanH** (page 136). They use or return degrees, not radians.

Rank **Rank** (page 192) lets you specify mid, lower, or upper rank in the event of a tie.

RunConsoleProcess **RunConsoleProcess** (page 368) 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.

System functions **GetRegistryValue()** (page 349) returns selected values from the computer registry, such as the default directory for model or data files. **ShowPdfFile()** (page 349) 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) **TypeOf** (page 141) 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 ***handle*** is a pointer to an object, such as a variable or module. The **Inputs**, **Outputs**, or **Contains** attributes create a list of handles to objects (page 344). 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)** (page 344) gives a handle to **X** instead of its value. **HandleFromIdentifier(T)** (page 344), as you might expect, gives a handle if **T** is the text identifier of an object. **IndexesOf(A)** (page 198) returns a list of handles of the indexes of array **A**.

Optional and repeated parameters The qualifier Optional (page 321) 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 "..." (page 322) 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 can be used from the Power Player edition.

Database functions You can now assign result of **DBQuery** (page 358) 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** (page 359) function supports the standard MDX language for querying and writing to multidimensional OLAP hypercube databases, such as offered by Microsoft SQL Server Analysis Services. This greatly expands ways to integrate Analytica with business intelligence and related applications.

MDTable Now **MDTable** (page 195) 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.

Performance Profiler library The Performance Profiler library (page 366) 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

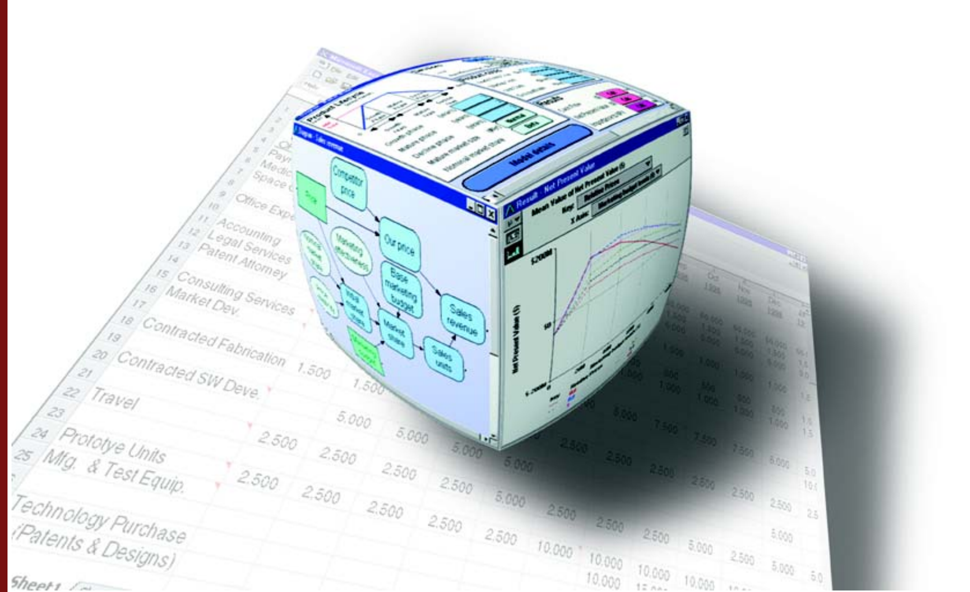
The Analytica Optimizer uses the new 7.0 release of the Premium Solver from Frontline Systems. New features include the **Grouped Integer** variable type, where a solution must assign a different integer from **1** to **n** to each variable in the group. The quadratic programming solver, **QpDefine**, now supports quadratic constraints in additional 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**, **Probit Regression,** and **Poisson Regression** using the Optimizer. These are described in the *Analytica Optimizer* manual.

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:

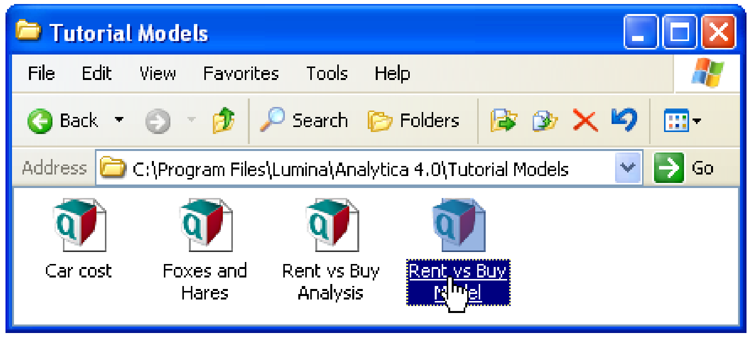
- Start a model
- Explore the **Diagram** window
- Classes of objects
- Explore the **Object** window
- Explore the Attribute panel
- Print the contents of windows



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**".

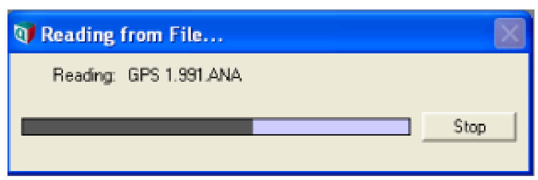
To open a model The simplest way to open an existing model is just to double-click the icon for the model file in the Windows directory.



Another way to open a model is to:

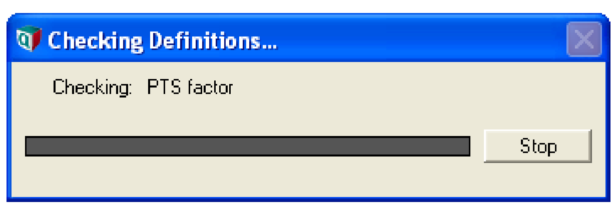
1. Start up Analytica by double-clicking the icon of the Analytica application, or selecting **Analytica** from the Windows **Start** menu. Analytica opens a new, untitled model.
2. In the top-left of the Analytica application window, press on the **File** pull-down menu, and select **Open Model**. A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

However you start a model, Analytica shows 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 stops 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 stops checking. Diagrams might have missing arrows and cross-hatched nodes indicating unchecked definitions. If you later ask to show the result of a variable, it checks any variables needed. Thus, clicking **Stop** simply defers some checking, and causes no problems with the model.

If the model contains any variables whose definitions are missing or invalid, they are listed in the **Invalid Variables window** (page 309). You can still compute results for variables with valid definitions, as long as they don't depend on variables whose definition is invalid.

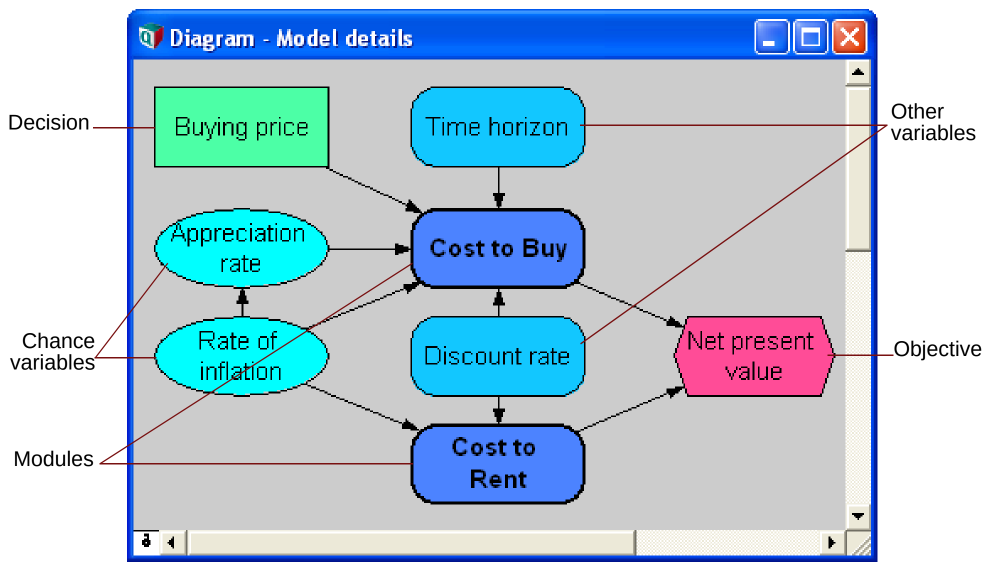
To close a model To close a model, select **Close Model** from the **File** menu. If you have made any changes to the model, a dialog asks you whether you want to save the changes before closing — except if you are using the Player Edition, which doesn't let you save a changed model.

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 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 prompts 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 can 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 can 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 the **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** (page 23).
- 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

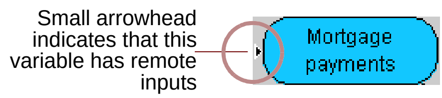
To see the diagram that contains the active module or variable, click the **Parent Diagram** button  in the navigation toolbar. The module or variable is highlighted in the parent diagram.

Tip

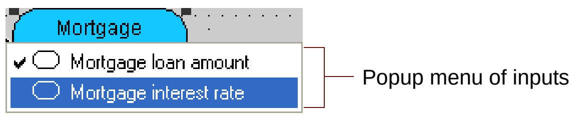
If the active diagram is of the top model, it has no parent diagram, and the **Parent Diagram** button is grayed out.

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).



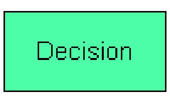
To see a list of the inputs (or outputs), remote and local, press the arrowhead on the left (or right) of the node.



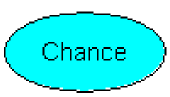
To jump to a remote input or output, select it from the list and stop pressing. It opens the **Diagram** window containing the remote variable, and highlights its node.

Classes of variables and other objects

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

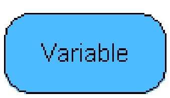


A rectangle depicts a ***decision variable*** — a quantity that the decision maker can control directly. For example, whether or not you take an umbrella to work is your decision. If you are bidding on a contract, it is your decision how much to bid.

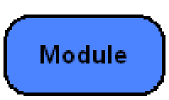


An oval depicts a ***chance variable*** — that is an uncertain quantity whose definition contains a probability distribution. For example, whether or not it will rain tomorrow is a chance variable (unless you are a rain god). And whether or not your bid is the winning bid is a chance variable in your model, although it is a decision variable for the person or organization requesting the bid.

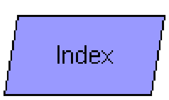
A hexagon depicts an ***objective variable*** — a quantity that evaluates the relative value, desirability, or utility of possible outcomes. In a decision model, you are trying to find the decision(s) that maximize (or minimize) the value of this node. Usually, a model contains only one objective.



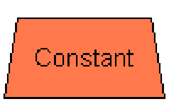
A rounded shape (with thin outline) depicts a ***general variable*** — a quantity that is not one of the above classes. It can be uncertain because it depends on one or more chance variables. Use this class initially if you're not sure what kind of variable you want. You can change the class later when it becomes clearer.



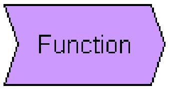
A rounded node (with thick outline) depicts a ***module*** — that is, a collection of nodes organized as a diagram. Modules can themselves contain modules, creating a nested hierarchy.



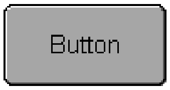
A parallelogram depicts an ***index variable***. An index is used to define a dimension of an array. For example, *Year* is an index for an array containing the U.S. GNP for the past 20 years. Or *Nation name* is an index for an array of GNPs for a collection of nations. Indexes identify the row and column headers of a table, and the axes and key of a graph (see "Introducing indexes and arrays"page 144).



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”page 363).

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.

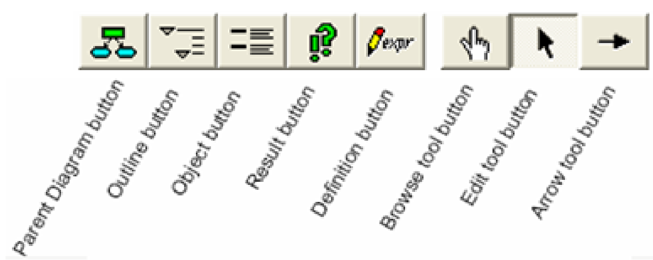
To select multiple nodes Click a node while pressing the *Shift* key to add it to the set of selected nodes. You can remove a node from the selection by clicking it again while pressing *Shift*.

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 shows the selection rectangle and selects all nodes within the rectangle.

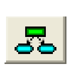
To deselect all nodes Just click the background of the diagram outside any node.

The toolbar

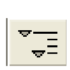
The toolbar appears across the top of the Analytica application window. It contains buttons to open various views of the model, and to change between browse and edit modes.



Navigation toolbar The first five buttons on the toolbar 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** (page 19) 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*.



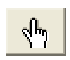
Outline button: Click to open the **Outline window** (page 304). The outline highlights the object you were previously looking at. The keyboard shortcut is *F3*.

Object button: Click to open the **Object window** (page 23) for the selected node in a diagram or the active module. The keyboard shortcut is *F4.*

Result button: Click to open a **Result window** (page 30) (table or graph) for the selected variable. This button is grayed out if no variable is selected. If you have selected more than one variable, it offers to create a compare variable that shows a result combining the values of all the variables. The keyboard shortcut is *Control+r* or *F5.*

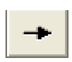
Definition button: Click to view the definition of the selected variable. If the variable is defined as a probability distribution or sequence, it opens the function in the **Object Finder** (page 112); if the variable is an editable table (edit table, subtable, or probability table), it opens the **Edit Table** (page 171) window. Otherwise, an **Attribute panel** (page 24) or an **Object window** (page 23) opens, depending on the *Edit Attributes* setting in the **Preferences dialog** (page 58). 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 does not let you change other variables. See “Browse mode”page 23.

Edit tool: Lets you create new objects, and move and edit existing objects. See “Creating and editing nodes”page 49.

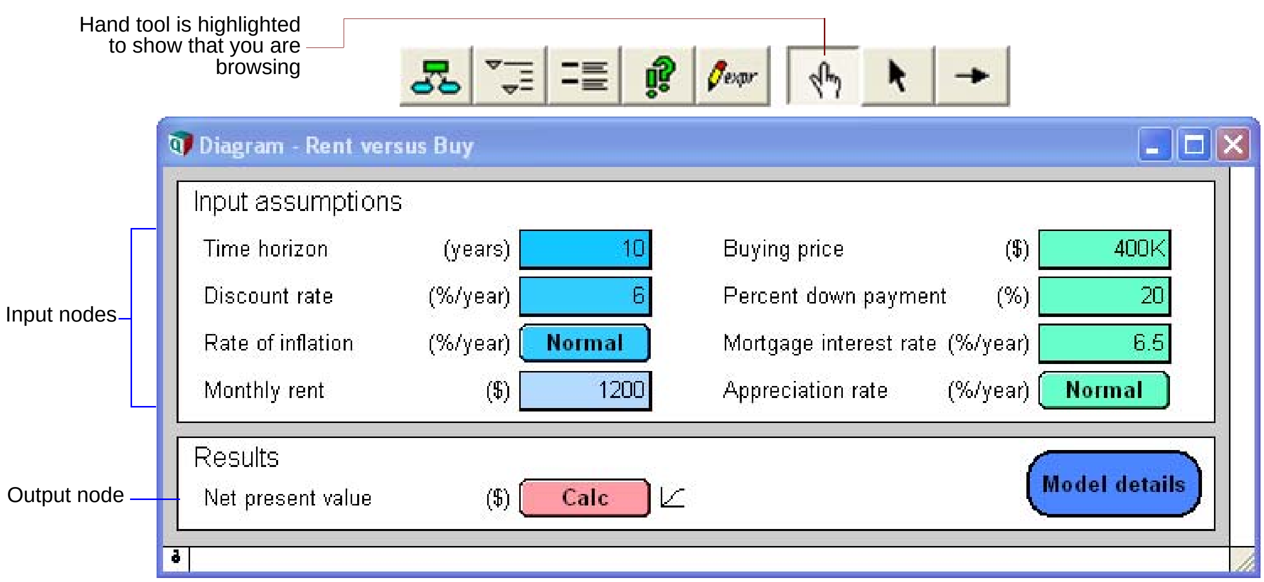


Arrow tool: Lets you draw arrows (influences) between nodes on a diagram. See “Drawing arrows”page 51.

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 might look like this (instead of an influence diagram).



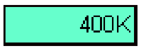
You can change the values in the ***input nodes*** directly. The ***output node***, Net present value, shows a **Calc** button. Click it to compute and see its value. Double-click the **Model details** node to open a diagram showing details of the model (the influence diagram shown above).

Browse mode

An existing model opens in browse mode. In this 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 pull-down menu lets you choose from a list of options. Press the menu to see the list.

Click the **List** 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"page 165.

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"page 171.

Click to view and edit a probability distribution in the **Function Finder**. For more, see "Probabilistic calculation"page 224.

Viewing output node values

Click the **Calc** button to compute and display the value of this output variable. When computing is complete, it shows 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.

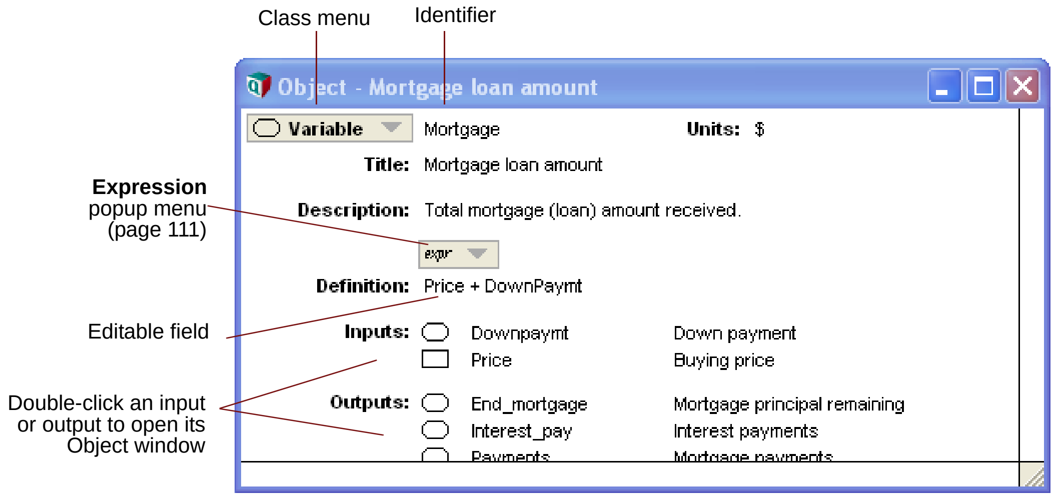
The **Result** button shows that an array has been calculated. Click it to open a **Result** window showing a table or graph. See Chapter 2, "Result Tables and Graphs" for more.

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"page 23).

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**:

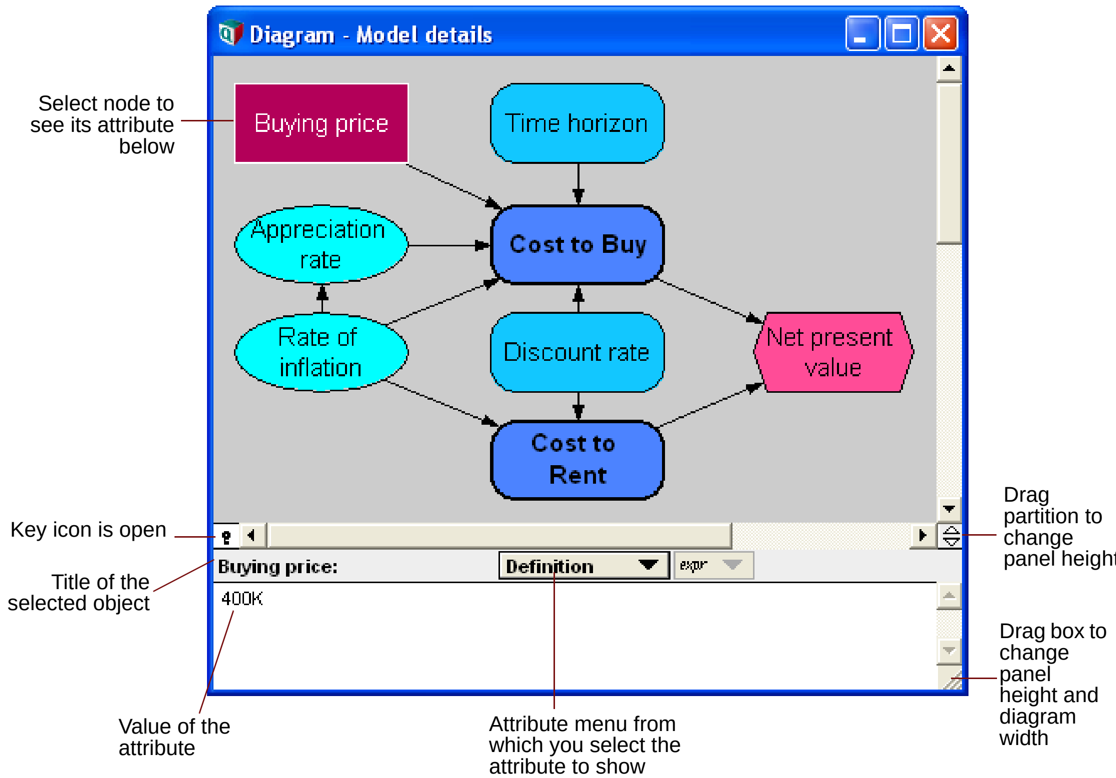
- Double-click **X** in a **Diagram** window.
- Select **X** in its **Diagram** window and click the **Object** button in the navigation toolbar.
- Double-click the entry for **X** in the **Outline window** (page 304).
- If a **Result** window for **X** is displayed, click the **Object** button in the 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

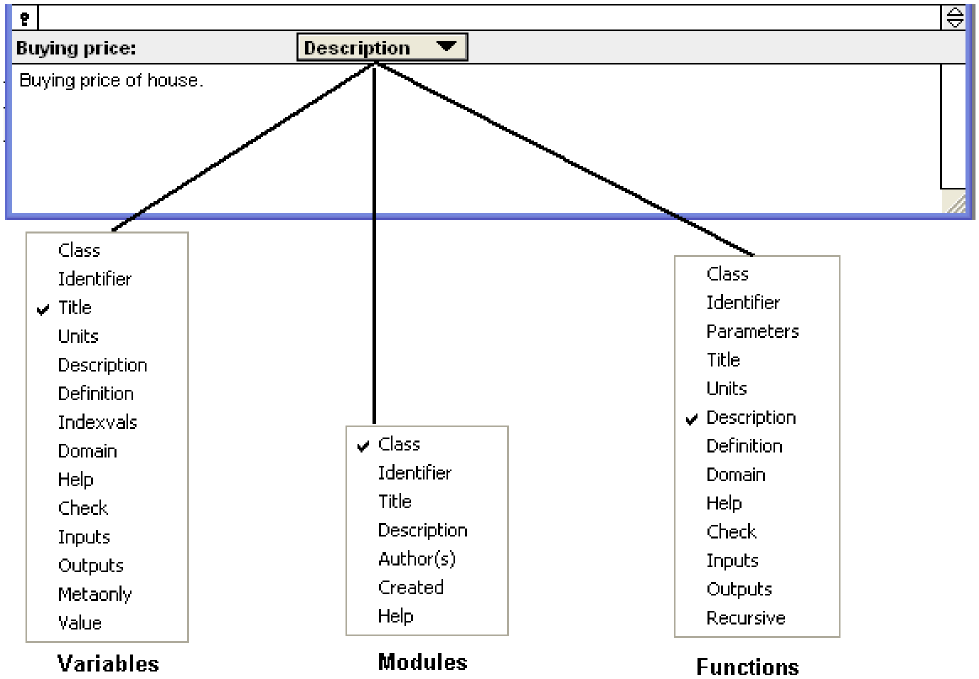
The ***Attribute panel*** offers a handy way to rapidly explore the definitions, descriptions, or other attributes of the variables and other nodes in a **Diagram** window. You can open the panel below the diagram, and use it to view or edit any attribute of the node you select. It shows the same attributes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



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

- 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.



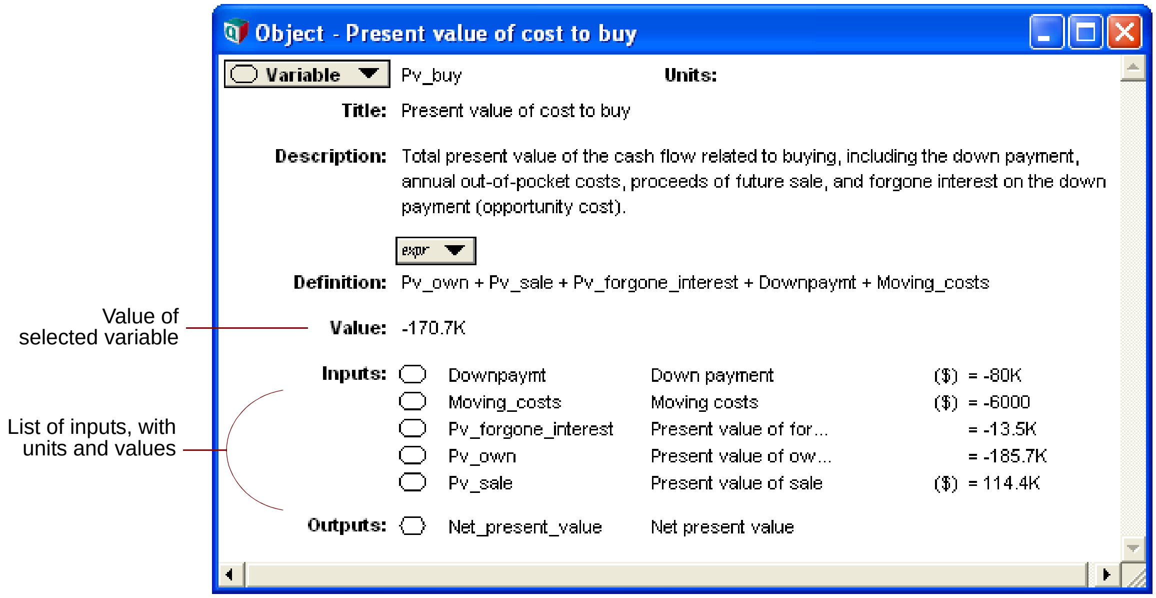
If you try to see an attribute not defined for an object, it shows its description.

See the “Glossary” for descriptions of these attributes. To display other attributes or to add new attributes, see “Managing attributes” on page 306.

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.



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

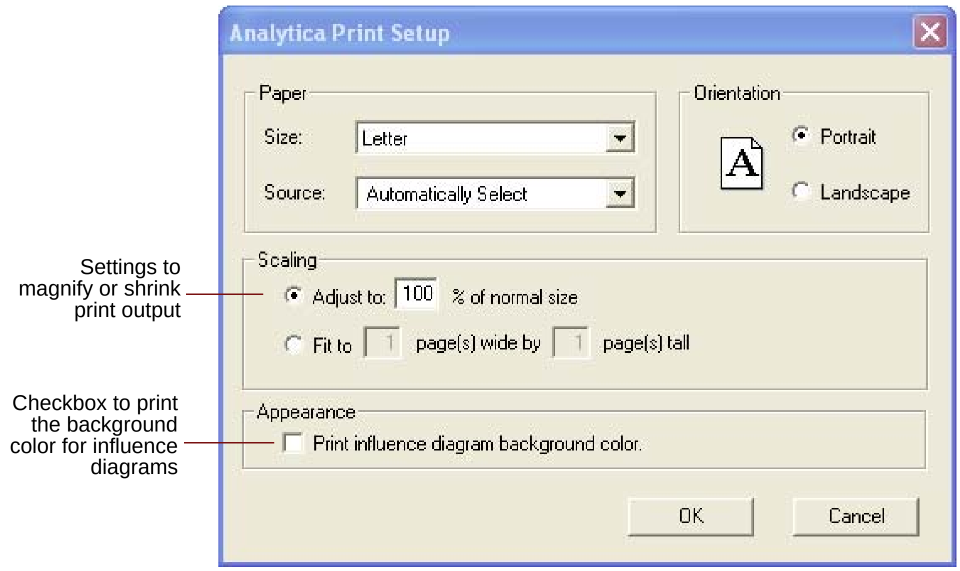
When you select the **Print preview** command on the **File** menu, it displays a **Preview** window to show what will be printed and where page breaks will occur. You can adjust print settings such as scaling until you get the desired page breaks. When previewing a result table or graph, you can toggle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

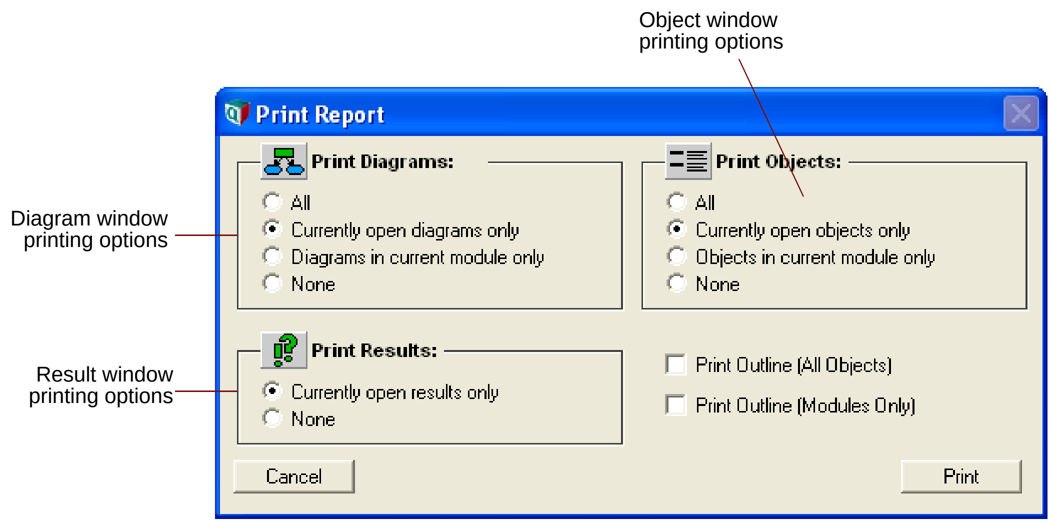
You can adjust the magnification of your printouts using the **Print Setup** command on the **File** menu, or by using the **Setup** button on the **Print Preview** window, in two ways:

- **Adjust to** *p* **% of normal size:** Use p<100% to shrink output, or 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 might be less than *n* x *m*.



Printing the background There is a checkbox in 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* checkbox. 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.



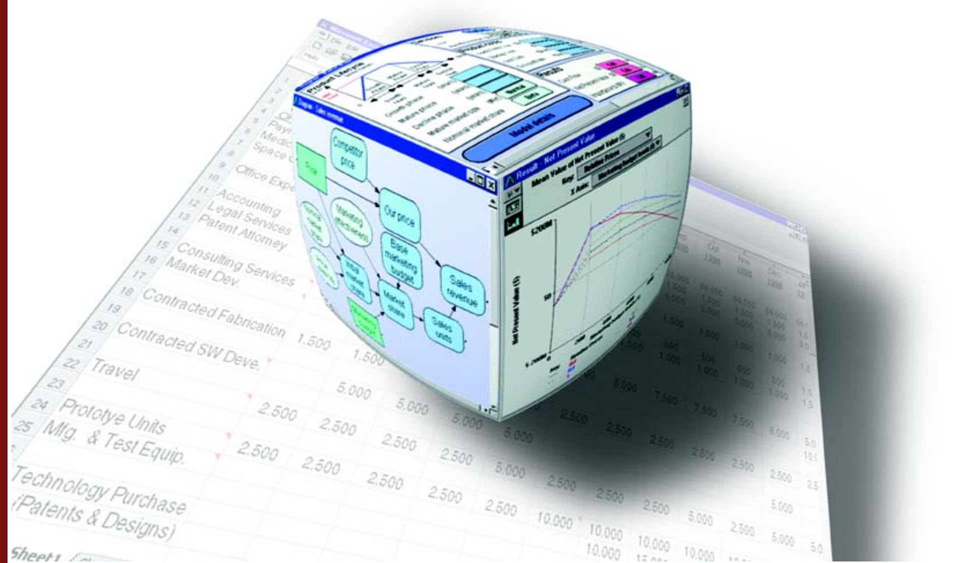
Check *Print Outline (All Objects)* to print a list of all objects in the model, each in its parent module, indented to show the module hierarchy.

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:

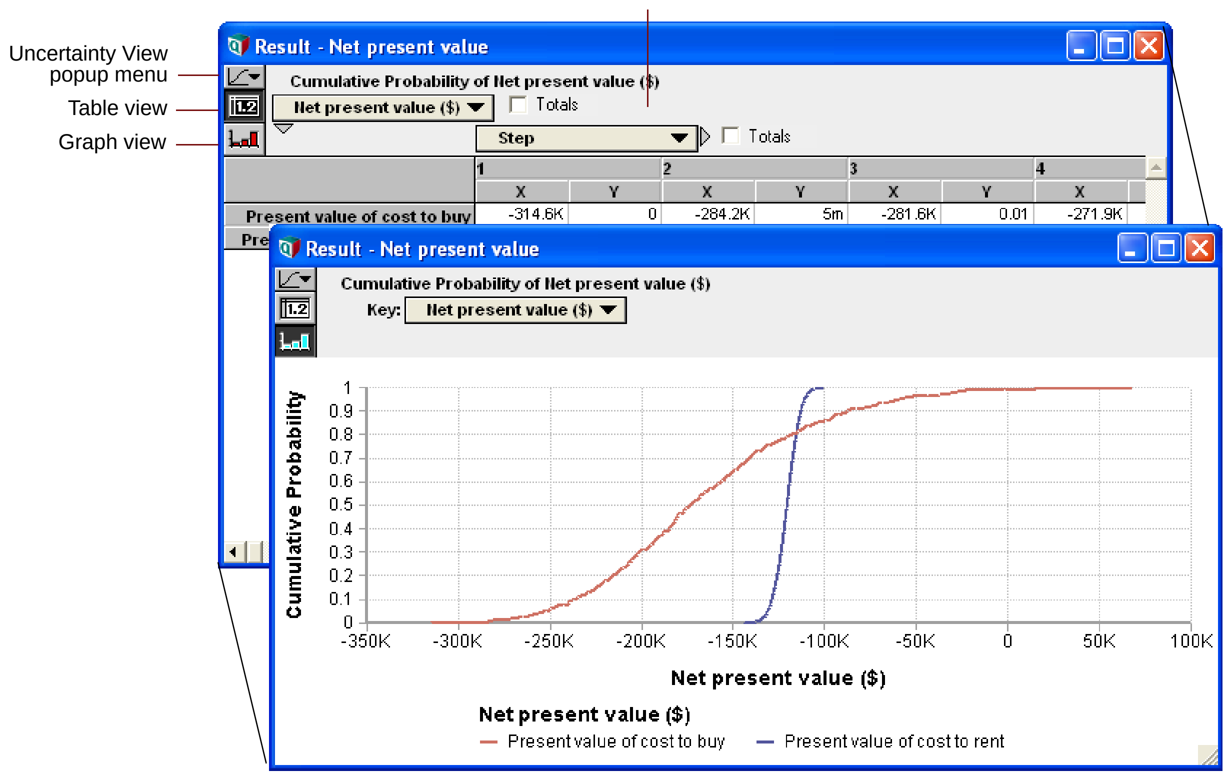
- View **Result** windows as graphs or tables.
- Rearrange or pivot results, exchanging rows and columns, or graph axes and keys, and slicer dimensions.
- Select an uncertainty view to display probabilistic results.
- Compare two or more variables in the same table or graph.



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



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.

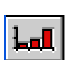
To open a **Result** window for an output node, simply click its **Calc** or **Result** button.

Result controls

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

Press the **Uncertainty View** popup menu (page 33), to select how to display an uncertain quantity.

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

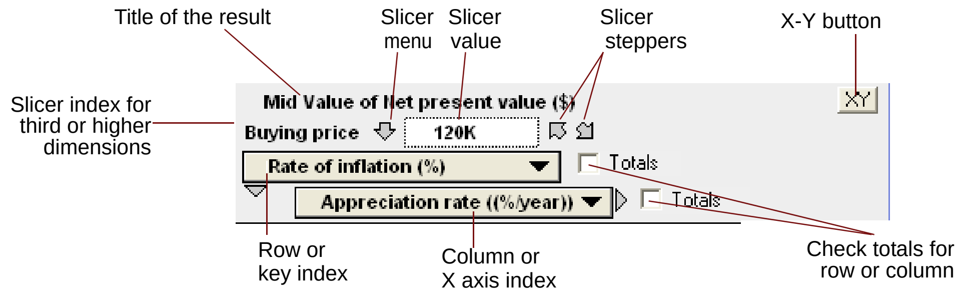


Click this button to display the result as a **graph**.

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 goes 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):

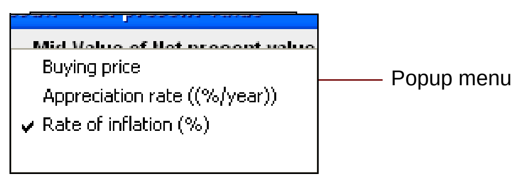
Title Shows the uncertainty view (mid, mean, etc.), the title of the variable, and its units, e.g., **`Mid Value of Costs of buying and renting ($)`**.

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 XY to plot this variable against one or more other variables, or to plot one slice of this variable against another slice. See “XY comparison”page 98.

Totals checkboxes 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” becomes 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** (page 58).

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.

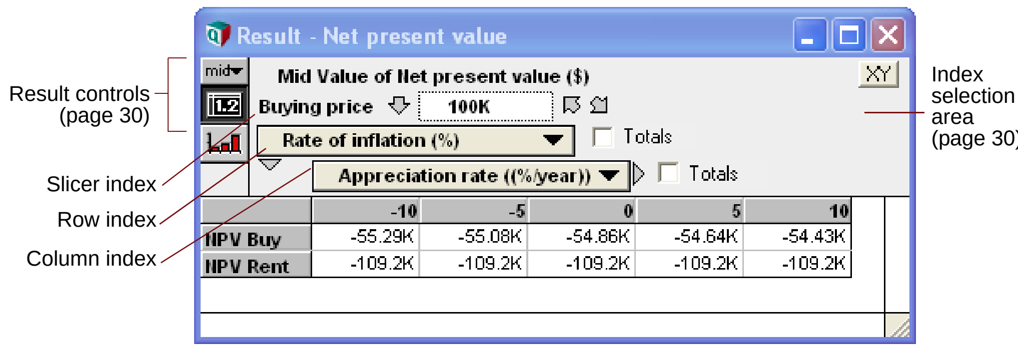
Recomputing results

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 [1.2] on the top-left to switch 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.

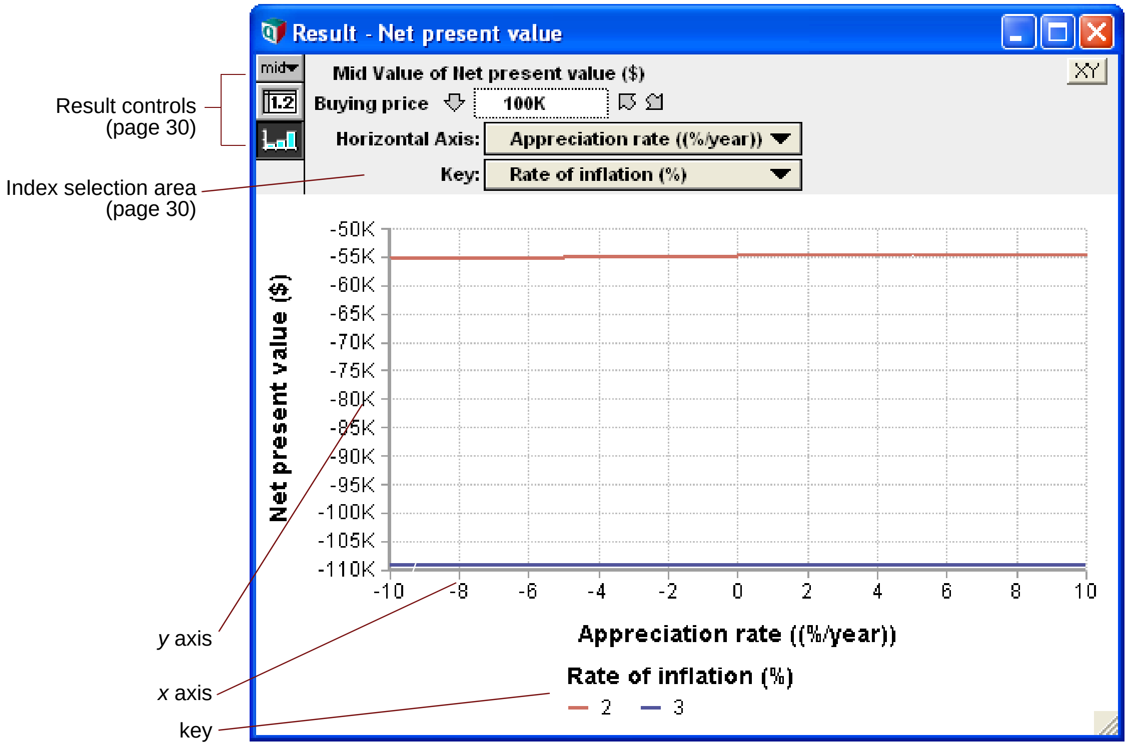
Column index Use this menu to select which index to display across the columns of the table. Select blank to display a single column.

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 and 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** (page 82) offers many options, including currency signs, dates, and Booleans.

Viewing a result as a graph

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

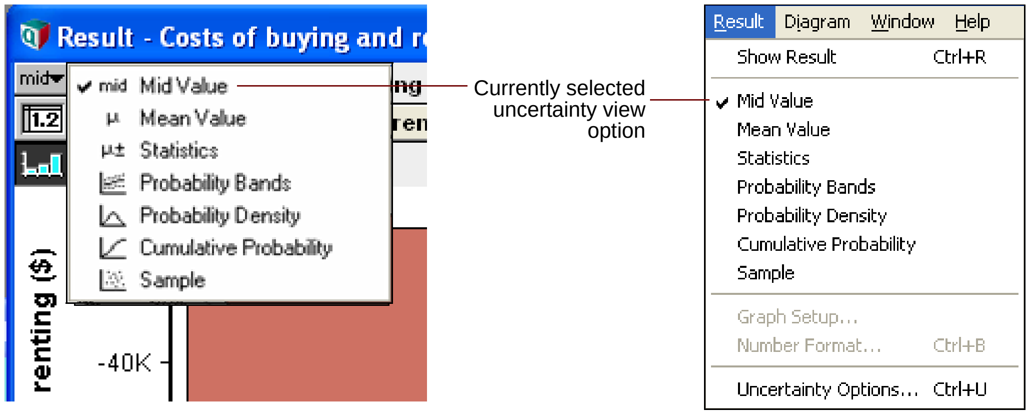
Graph setup options There is a rich variety of ways to customize the graph, including line style (lines, data points, symbols, barcharts, stacked bars, thickness, transparency), axis ranges, log or inverted axes, grid and tickmarks, background colors, and font color and size. To change these settings, open the **Graph Setup dialog** (page 89) and do one of the following:

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

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, can also have an uncertain or probabilistic value, which we term its ***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 seven ***uncertainty views***, including the mid value (which is not uncertain) and six ways to display a prob value. You can select the uncertainty views from a menu in the top-left corner of a **Result** window. Or you can select a variable, and select an uncertainty view option from the **Result** menu.



Uncertainty View popup menu from Result window

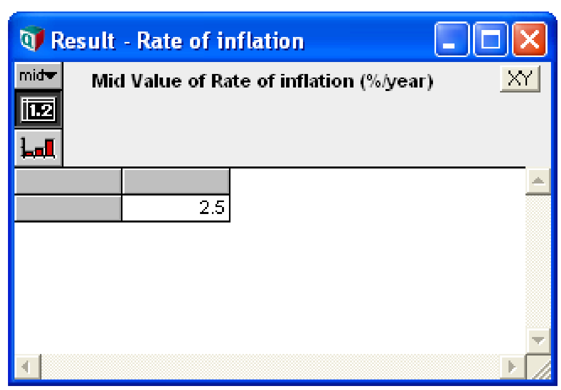
Result menu uncertainty view options

The checkmark 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:

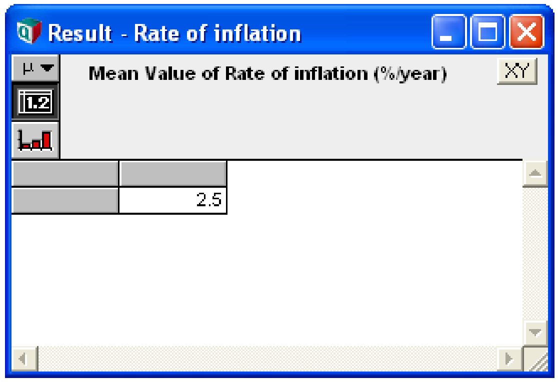
```
Chance Rate_of_inflation := Normal(2.5, 1)
```

Mid value The mid value is 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.



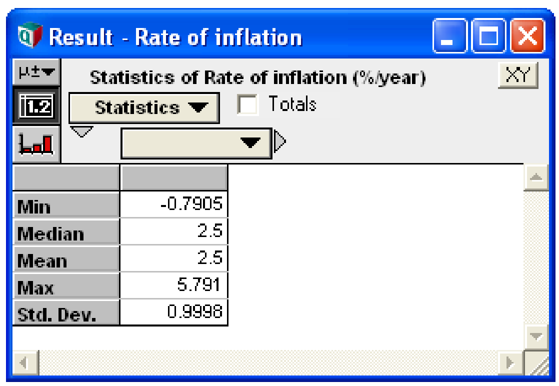
Tip A mid value is much faster to compute than a prob(abilistic) value, since it doesn't use Monte Carlo simulation to compute a probabilistic sample. It is often useful to look first at the mid value of a variable as a quick sanity check. Then you might select an uncertainty view, which causes its prob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

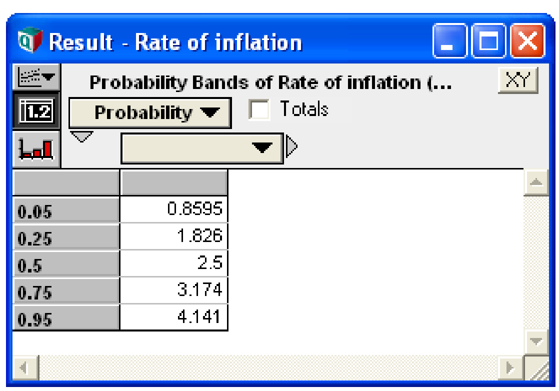


Tip The mean and the other uncertainty views below are estimates based on the Monte Carlo (or Latin hypercube) sample. The precision of these estimates depends on the sample size and the sampling method. A larger sample size gives higher precision and takes more time and memory to compute. You can **modify the sample size** (page 372) and sampling method in the **Uncertainty setup dialog** (page 225) from the **Result** menu.

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** (page 228) of the **Uncertainty Setup** dialog from the **Result** menu.

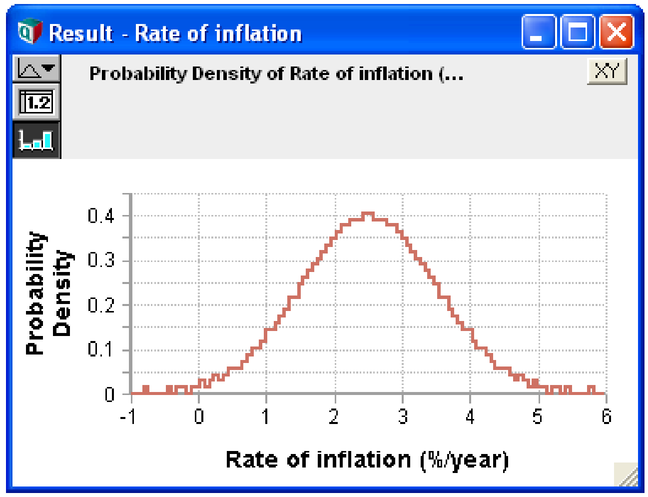


Probability bands An array of percentiles (fractiles) estimated from the random sample, by default the 5%, 25%, 50%, 75%, and 95%iles. You can select which percentiles to show in the **Probability Bands tab** (page 228) of the **Uncertainty Setup** dialog from the **Result** menu.



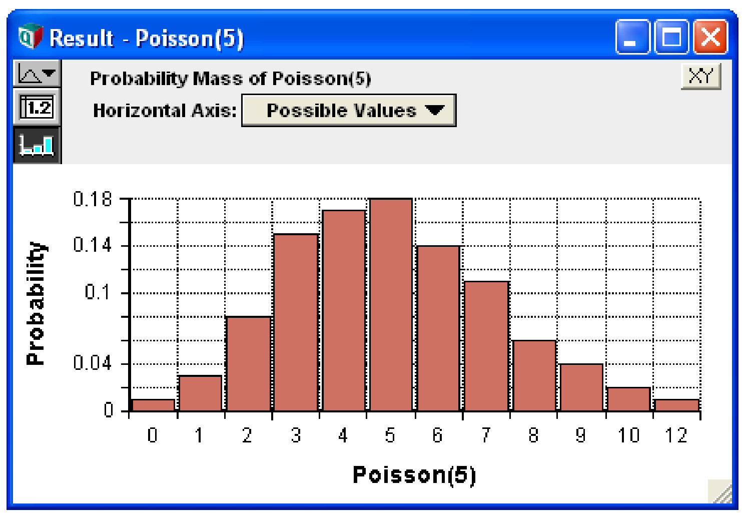
Probability density Select **probability density** to display the uncertain distribution as a probability density function (PDF).

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 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.

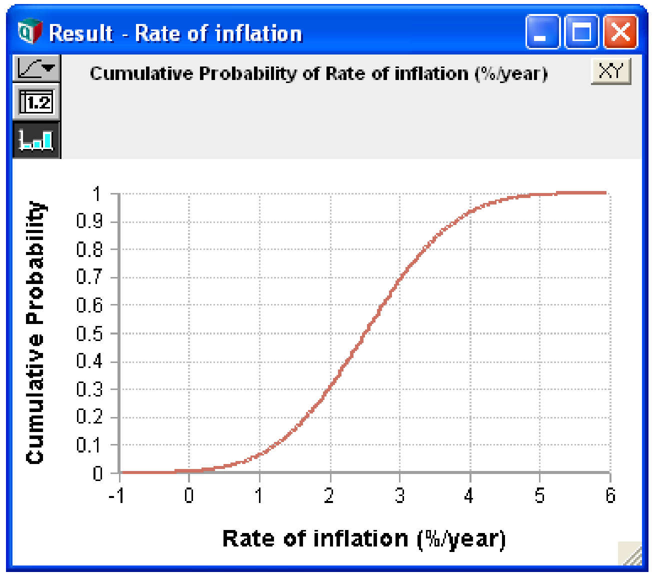


Usually, it figures out whether to use a probability density or mass function. Very rarely, you might need to tell it the domain is discrete. See “The domain attribute and discrete variables”page 236, “Is the quantity discrete or continuous?”page 220, and “Probability density and mass graphs”page 234 for more.

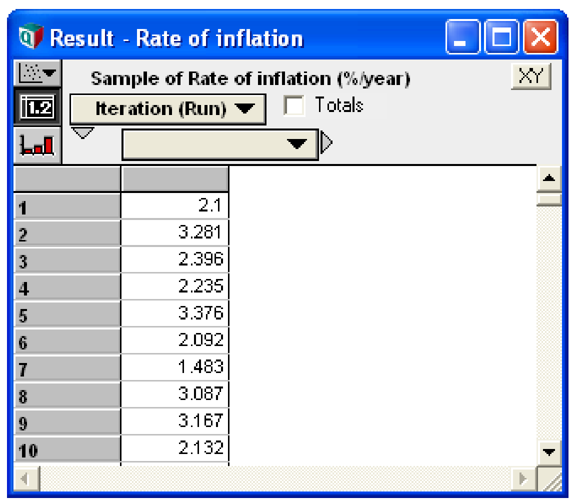
Cumulative probability

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 is 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.

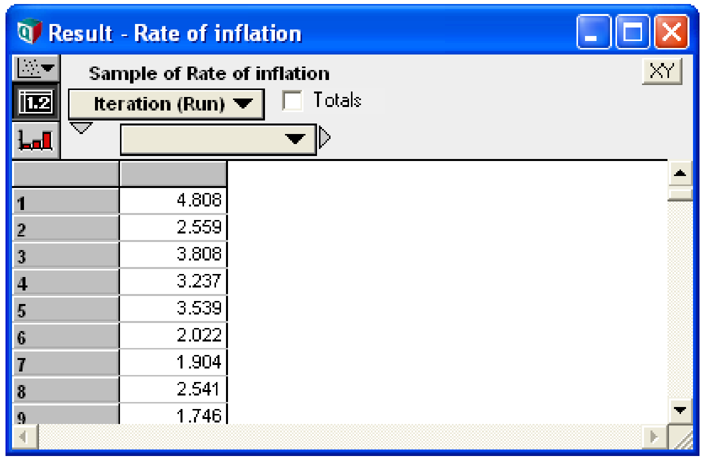
The steeper the curve, the more likely the quantity will have a value in that region. The PDF is the slope (first derivative) of the CDF. Conversely, the CDF is the cumulative integral of the PDF.



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.



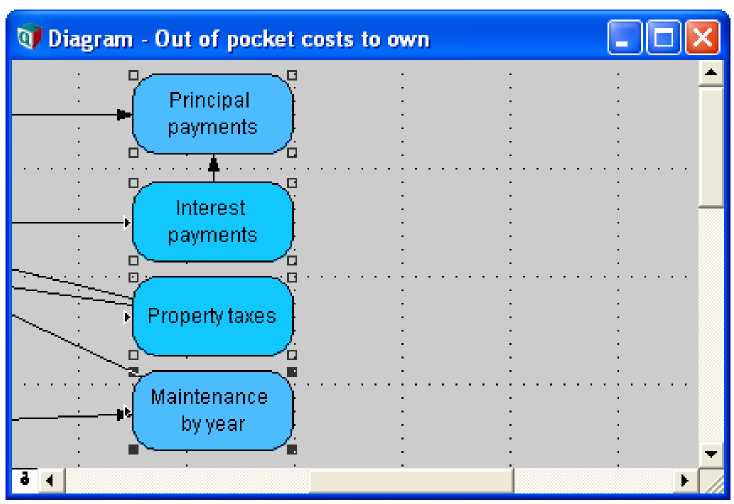
Like any other graph, you can display a sample as a table by clicking [1.2] to see the underlying numerical values.



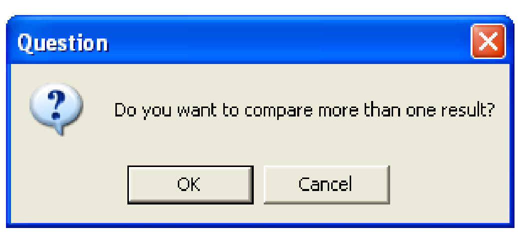
Comparing results

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

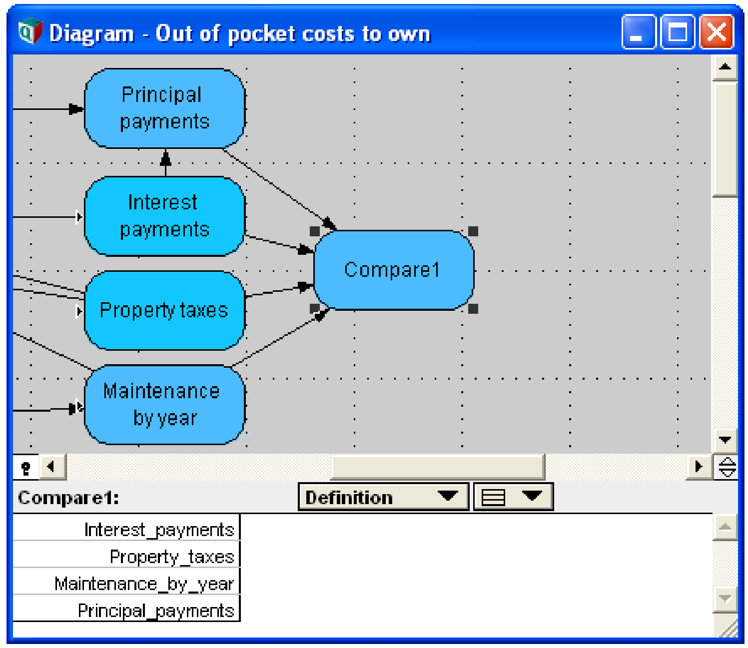
1. Select the variables together in the diagram, using *Shift+click* to add each to the selection, or dragging a selection rectangle around them.



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.



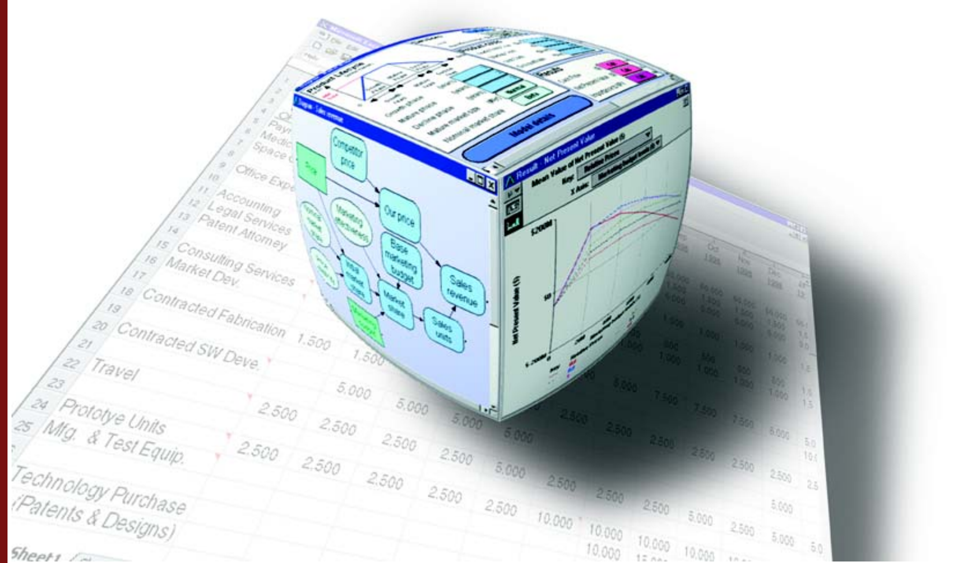
The result of **`Compare1`** is a graph containing an index containing the titles of the variables being compared. This is the **`Self`** index of the **`Compare1`**. It also includes all the indexes of the array variables being compared — in this case, **`Time`** and **`Buying Price`**.

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

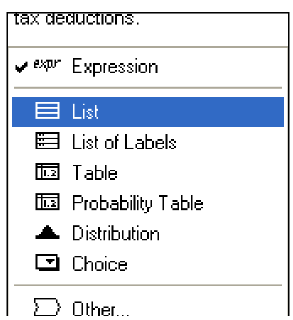
The first step in analyzing model behavior is to select one or more input variables as parameters and to assign each parameter a list of possible values.

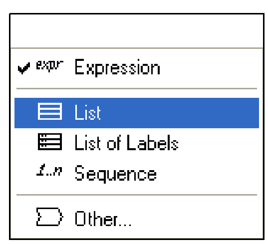
Which inputs to vary? You can vary any numerical input variable of your model, including decision and chance variables. Often you will want to vary each decision variable to see which value gives the best results according to the objectives. You might also want to vary some chance variables to see how 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 might 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 are sufficient. If you have a special interest in a particular parameter (for example, if you suspect it has a strongly nonlinear effect) you can 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.

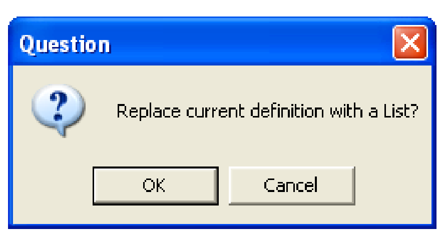
Creating a list Change the definition of each parameter to a list, thus:

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 ***expr*** (Expression) menu above the definition and select the **List** option. (Do *not* select the **List of Labels** option.)





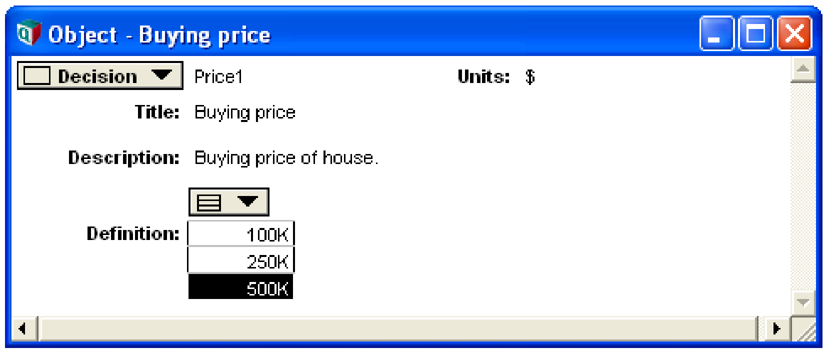
4. A dialog asks for confirmation. Click **OK**.



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

New one-element list

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



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”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 (page 167), for example:

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

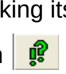
How many inputs to vary

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 can 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 might 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

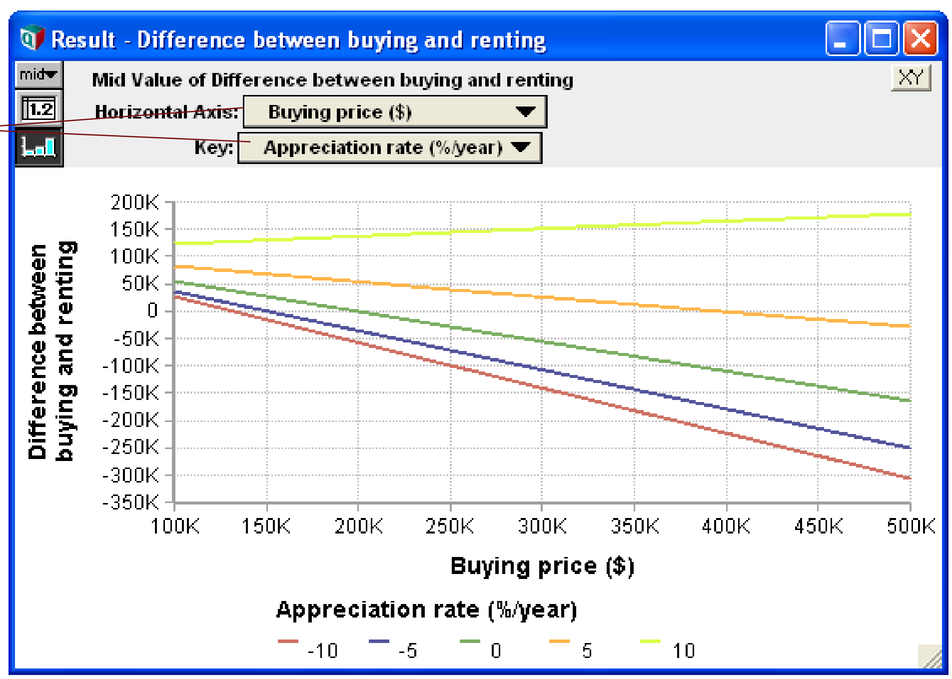
When you have assigned a list to one or more inputs, you can examine their effect by viewing the result on an output variable. If your model has an objective, start by looking at that variable.

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 might 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"page 30) to get additional insights into model behavior.

Result graph with dimensions reversed



Understanding unexpected behavior

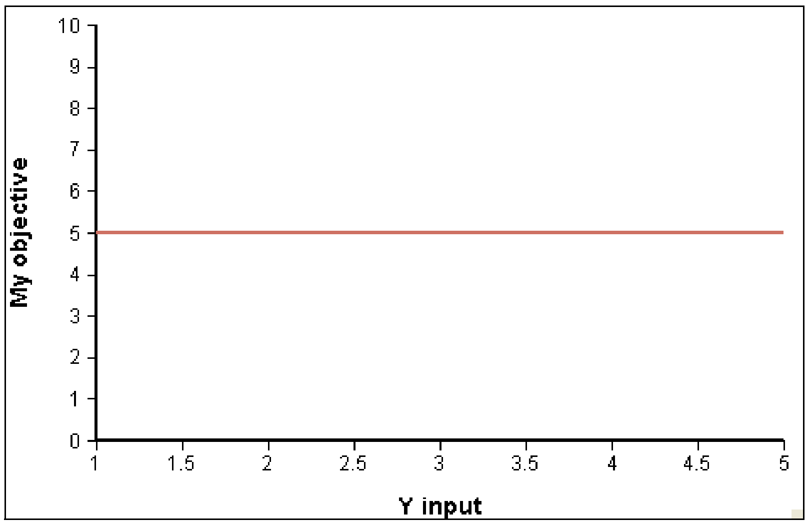
If you find the model's behavior unexpected or inexplicable, you might want to look more deeply into how the behavior arises. An easy way to do this is simply to look at the results for other variables between the input(s) and the output(s) in which you're interested. You can work forward from an input towards the output, or backward 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 might be that some intermediate relationship has an effect different from what you expected. There might be an error in a definition. In either case, this kind of exploration can be very revealing about the model. You might end up improving the model or gaining a deeper understanding of the system it represents.

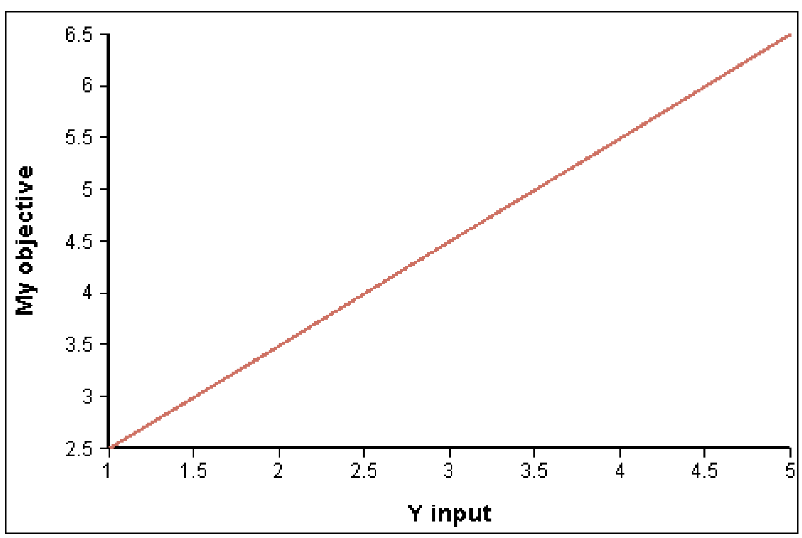
Understanding model behavior

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.

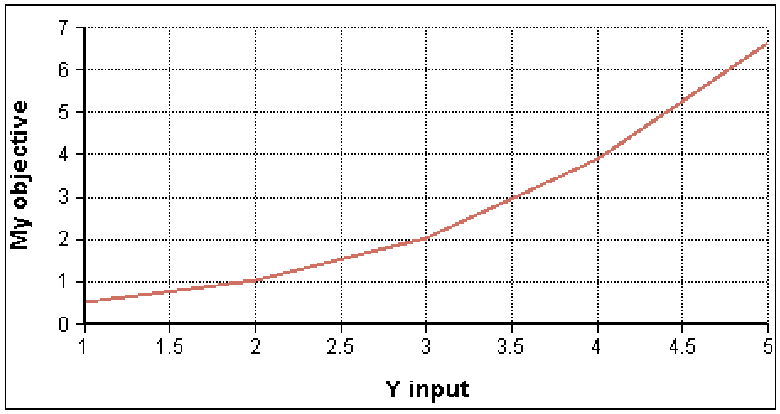
- A horizontal line shows that changes in the input over the specified range have no effect on the output.



- 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.



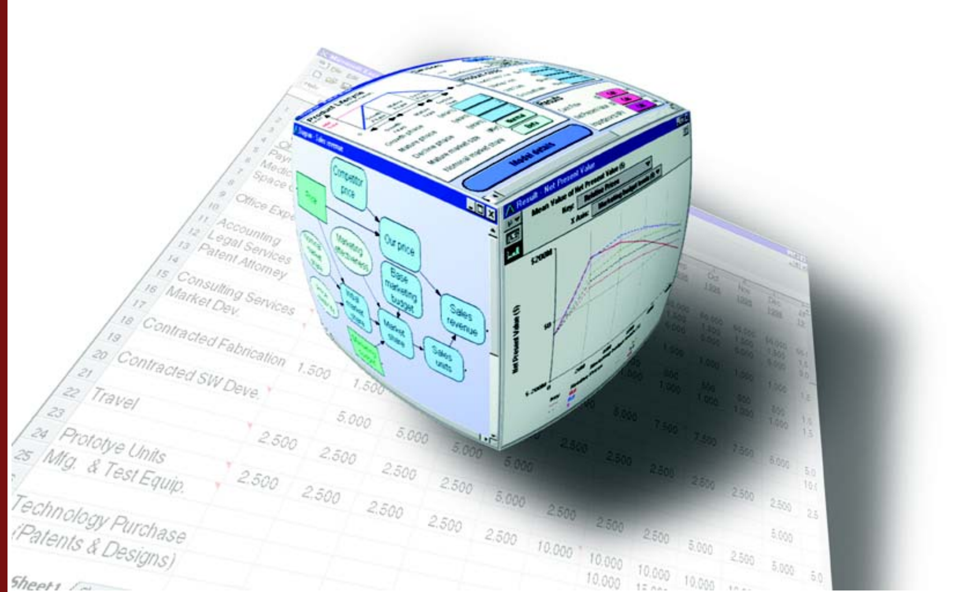
- A bent or curved line shows that there is a nonlinear dependence. (If you have only two values for the input, the graph will be a straight line even if there is a nonlinear dependence.)


My objective
7
6
5
4
3
2
1
0
1
1.5
2
2.5
3
3.5
4
4.5
5
Y input

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
- Edit attributes
- Change the class of an object
- Work with the **Preferences** dialog



Creating and saving a model

To start a new model

Start Analytica like any Windows application by selecting **Analytica** from the Windows **Start** menu or double-clicking the Analytica application file. A new, untitled model opens.

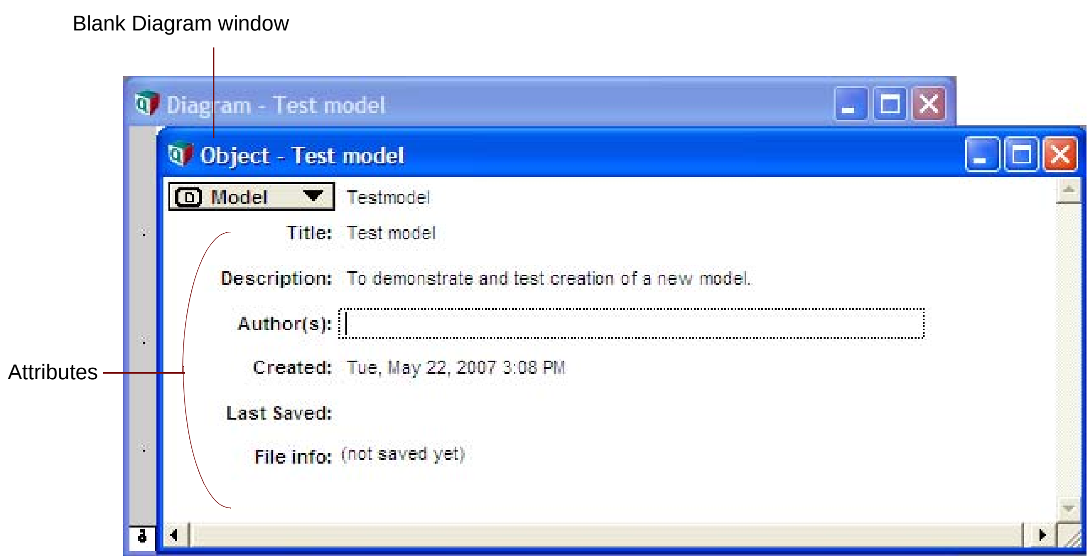
If you are already running an Analytica model, you can also select **New Model** from the **File** menu. Since one instance of Analytica can't run two models at once, it needs to close the existing model. If you have changed it, it first prompts you to save it.

The model's Object window

The model's **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, type these attributes:

- **Title:** A word or phrase to identify the model, typically up to 40 characters. Usually the identifier of the project is set automatically to the first 20 characters of the title, substituting underscores (_) for spaces or other characters that are not letters or numbers.
- **Description:** One or several lines of text describing the purpose of this model, and any other important information about the model or project that all users of the model should know.
- **Author(s):** Windows usually fills in the name of the Windows user as the default. You can edit or add to this if you like.



After adding these attributes into the **Object** window, bring the **Diagram** window to the top using one of these methods:

- Click the **Parent Diagram** button .

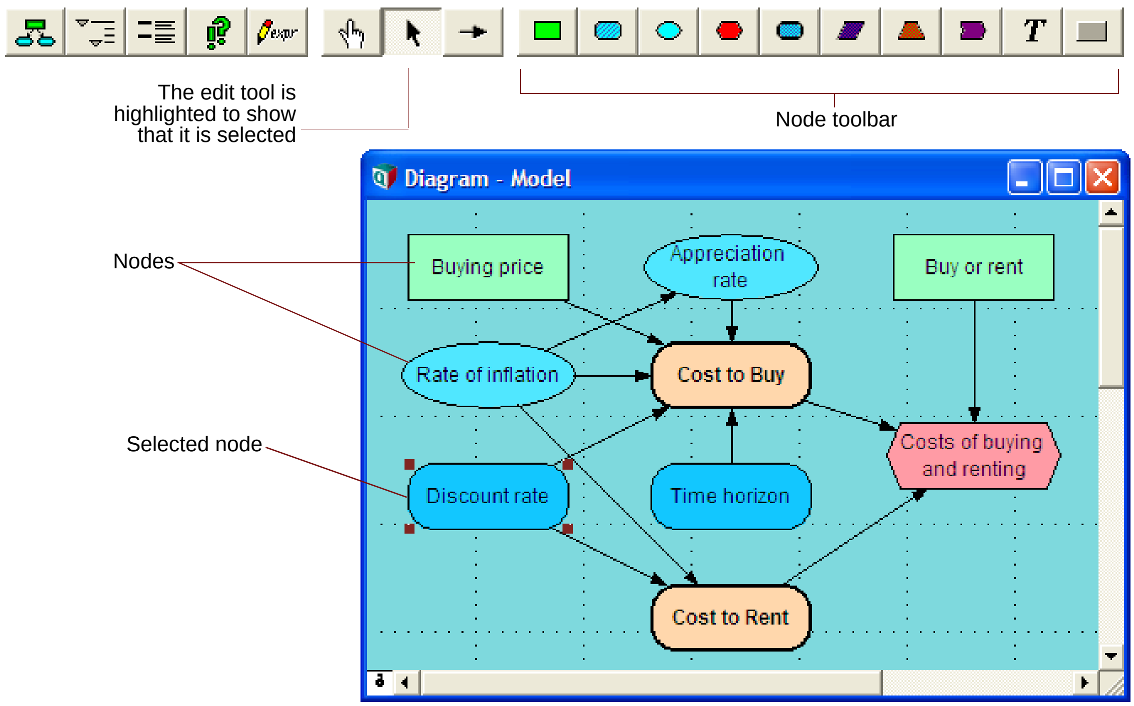
  or

- Click anywhere in the **Diagram** window behind the **Object** window.

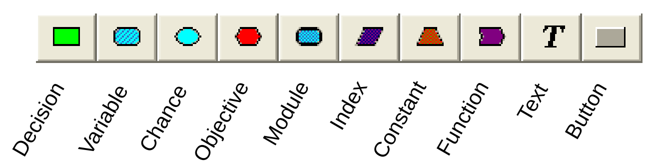
You are now ready to draw an influence diagram for the new model.

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.



For a description of each node shape (or class), see “Classes of variables and other objects”page 20.



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

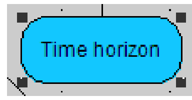
Create a node To create a new node, press the mouse button with the cursor over the node class you want in the node toolbar, and drag the node to the location you want in the diagram. When creating a new node, you can type a title directly into it.

Edit a node title To edit the title of an existing node:

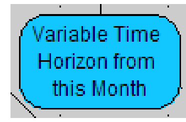
1. Make sure you are in edit mode.
2. Click the node once to select it.
3. Click the node’s title. (Pause momentarily between mouse clicks to prevent them being interpreted as a double-click, which would open the node’s **Object** window.)
4. Type in a new title to replace the old one. Or click a third time to put a cursor into the existing title where you can add text. Or double-click to select a word to replace.
5. After editing the title to your satisfaction, 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.

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

Identifiers and titles

Every object has a unique ***identifier*** of up to 20 characters. An identifier must start with a letter, and contain only letters, digits, or underscores (_). Formulas in the definition of a variable or function refer to other variables or functions by their identifier.

Most objects also have a ***title***, which is usually displayed in its diagram node. A title can contain any number of characters of any type, including spaces. A title should be a meaningful word or phrase. Avoid obscure acronyms. It's usually best to keep a title to under 50 characters.

Making an identifier from a title

By default, when you enter a title, it also generates an identifier for the object 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, unlike title, must be unique. So, if by chance an object exists with the same identifier, it appends a number to the new identifier to keep it unique.

If you edit the title again, it usually asks if you want to change the identifier to match the changed title. Generally, it's best to have them match. But, sometimes you might want to retain the original identifier. You can change this default behavior by unchecking **Change identifier when title changes** in the **Preferences** dialog from the **Edit** menu (page 58).

Automatic update when identifier changes

If an identifier changes, Analytica automatically updates any definitions referring to that identifier it to use the new version, and so keeps the model consistent.

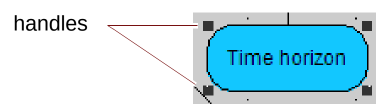
If you want, you can edit an identifier directly in the **Object** window or **Attribute** panel, like any other user-editable attribute.

Show identifiers instead of titles

By default, it shows the title of each node in a diagram or result window. To show the identifiers instead, select **Show by Identifier** from the **Object** menu, or press *Control+y* to toggle this behavior.

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, press the right mouse button on the node (not on a handle) and drag it to where you want it.

You can also adjust the position of one or more selected nodes with the *arrow* keys (*up*, *down*, *left*, *right*). By default, each *arrow* press moves the node(s) by eight pixels. If you uncheck **Snap-to-grid** in the **Diagram** menu, each *arrow* press moves the node(s) by one 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 has the same location in the diagram of the new module that it 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 asks 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 more nodes. 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**, but without writing to the clipboard. Duplicating a node creates a new object identical to the original, but it adds a number to its identifier to make it unique and locates it below and to the right of the original node.

Duplicating a set of nodes retains the same dependencies 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 get two new variables:

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

Note that (a) it appends “1” to the identifiers to make them distinct from their original nodes, and (b) 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”page 107).

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. You can change these settings in the **Diagram Style dialog** (page 78) and **Node Style dialog** (page 79).

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 asks if you really want to remove it.

Tip When you **enter or edit a definition** (page 108), Analytica automatically updates the arrows into the variable to reflect those other variables that it mentions (or does not mention).

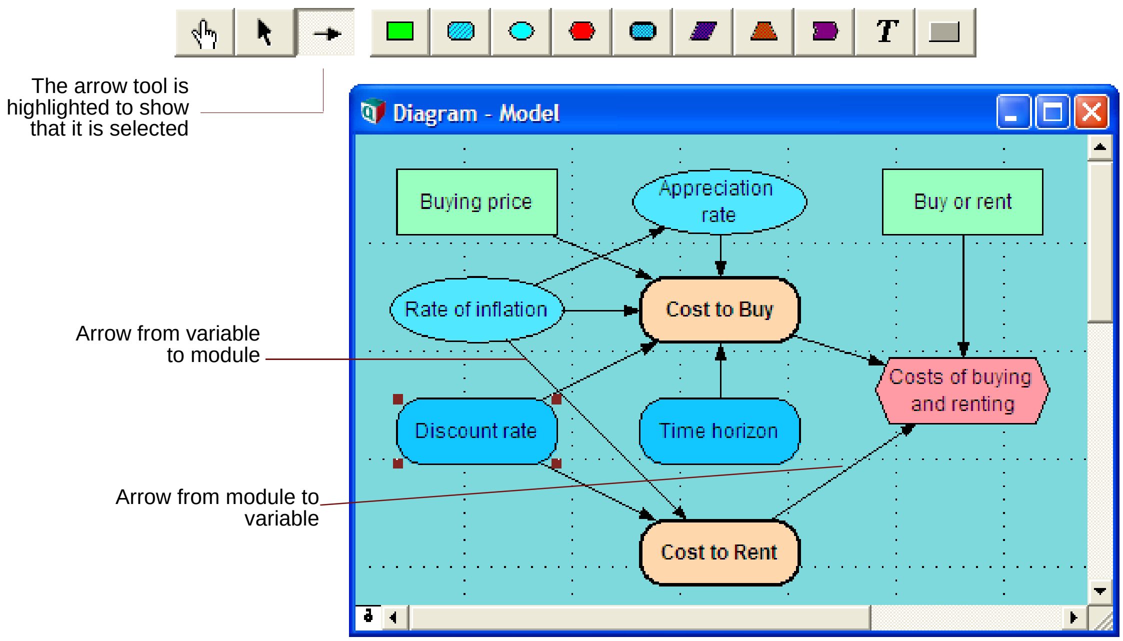
Influence cycle or loop

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

If you try to draw arrows that would make a cycle, it warns and prevents 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

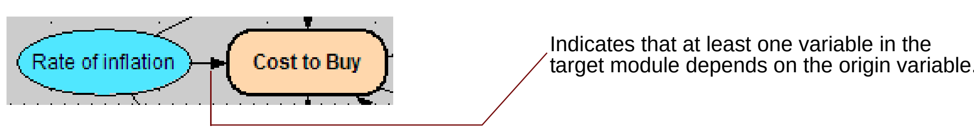
When there are arrow between variables in different modules, they are reflected by arrows to and from the module nodes.



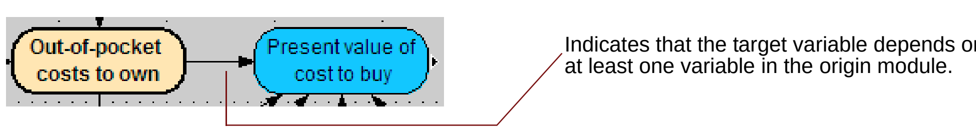
Arrows between variable and module nodes are illustrated below.

Arrow from variable node to variable node

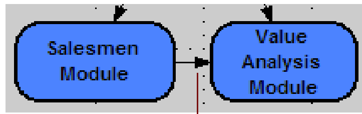
Arrow from variable node to module node



Arrow from module node to variable node

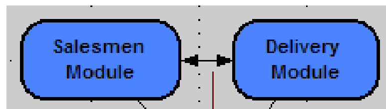


Arrow from module node to module node



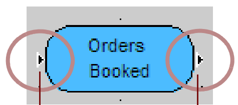
Indicates that the target module contains at least one variable that depends on at least one variable in the origin module.

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" on page 20).

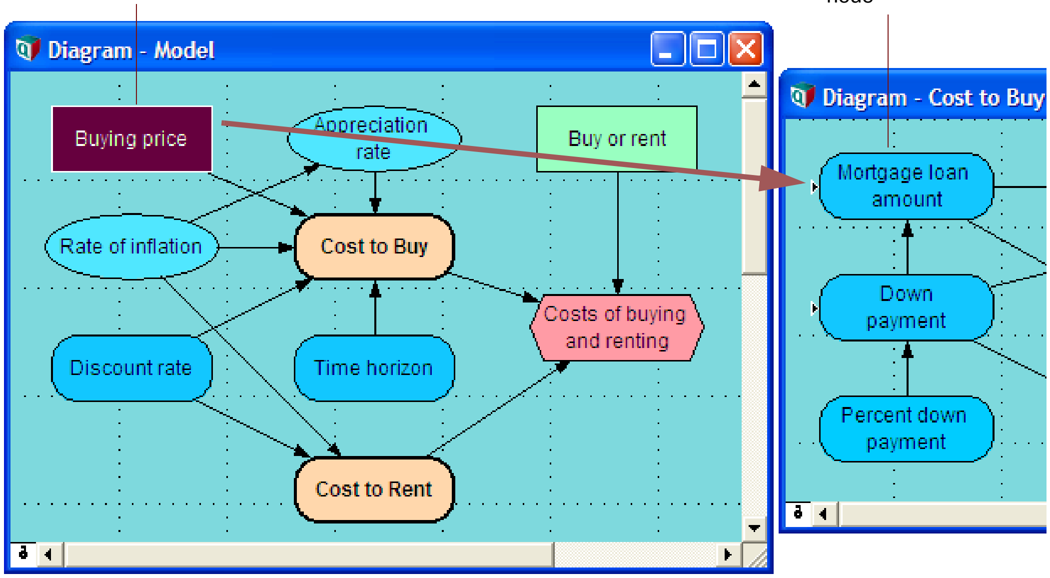
input arrow output arrow

How to draw arrows between different modules

There are four methods to draw arrows between nodes in different modules. Suppose you want to draw an arrow from the variable **`Buying price`** to the variable **`Mortgage loan amount`** in another module.

Source node

Destination node

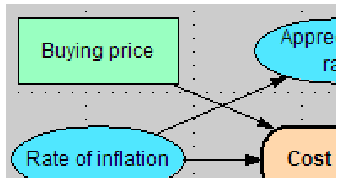


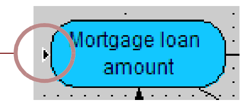
Draw arrow across windows

The most direct method works when you can arrange the diagrams so that both the origin and destination nodes are visible on screen at the same time:

1. In arrow mode , press on the origin node, **`Buying price`**, so that it highlights.
2. Drag an arrow to the destination node, **`Mortgage loan amount`**, which also highlights, and release the button.

If, as in this illustration, 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 arrow-head 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 in the usual way. Finally, you move the node back to the diagram it came from. This is convenient if you have large diagrams and a small screen so that its hard to arrange the two diagrams so that both nodes are visible at the same time.

Copy the identifier of the origin into the definition of the destination Copy the identifier of the origin variable, open the definition of the destination variable, and paste it in (see “Creating or editing a definition”page 108). When the definition is complete and accepted, it automatically draws 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 makes 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 similar method when the origin module and destination node are in the same diagram. Drawing an arrow between them creates an alias of the destination in the origin module. See the next section for more about aliases.

Alias nodes

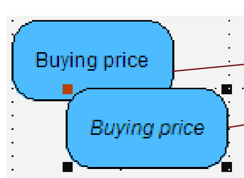
An ***alias*** is a copy of a node, referring to the same variable, module, or other object as the original node. It’s often useful to display an alias node in a different module than its original node. For example, if module **`M1`** contains variable **`X`**, and **`X`** has outputs in another module **`M2`**, it’s 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 can 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*.

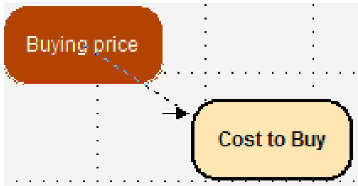
You can create an alias directly with the **Make alias** command, or indirectly by drawing an arrow to or from a module node. These methods are described below.

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.

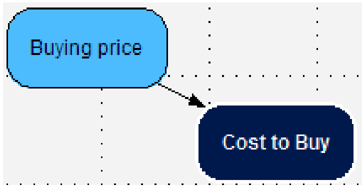


Draw arrow between variable and module

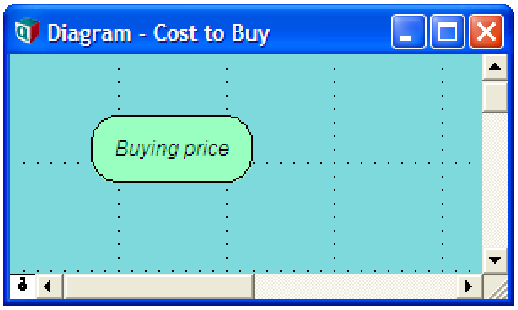
Draw an arrow from the original node to a module node, or from a module node to the original node. This creates an alias in the module. For example, draw an arrow from the variable **Buying price** to the module **Cost to Buy**.



It displays an arrow between the nodes.

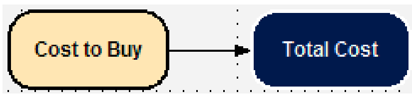


Open up the module **Cost to Buy** to see the new alias.

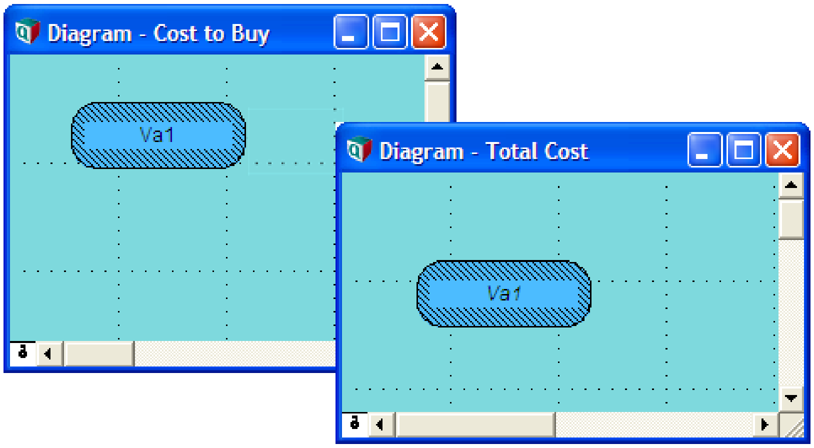


Draw arrow between two modules

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



This 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 it **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 — can 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 using the **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 nodes (page 120) and **output nodes** (page 122) are kinds of alias nodes that have special style properties.

To edit an attribute

You can edit most attributes of an object directly in the **Attribute panel** (page 24) or in the **Object window** (page 23). 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

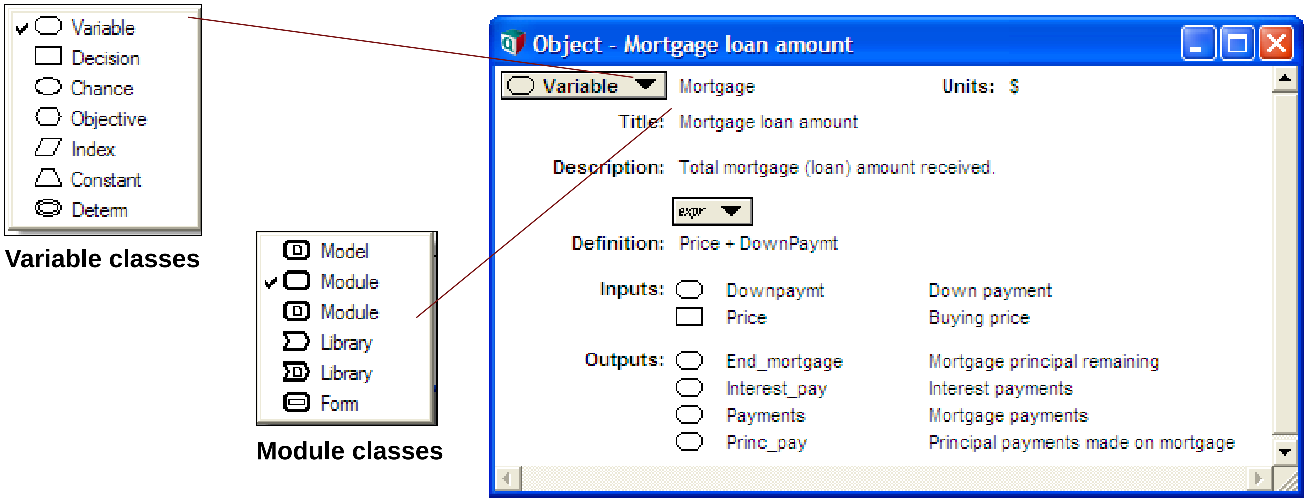
To cancel changes while editing an attribute, press the *Esc* (escape) key to revert to the previous version. Except when editing a definition, click to cancel changes. To cancel changes *after* you have just made and accepted them, select **Undo** from the **Edit** menu (or press *Control+z*).

Attribute changes

All displays of an object use its same attributes, so any change to an attribute affects 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 causes the redrawing of arrows to reflect any changes in dependencies.

To change the class of an object

You can 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.



Variable classes

Module classes

To change class, just select another option from the menu. The shape of the node and other class-dependent properties 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"page 20.

Module Subclasses

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

Model: Usually the top module in a module hierarchy, saved as a separate file (document with extension **.ana**). Any nondefault preferences (see "Preferences dialog"page 58), uncertainty options (see "Uncertainty Setup dialog"page 225), and graph style templates are saved with the model, but not other module types.

Module: A collection of nodes 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 page 309.

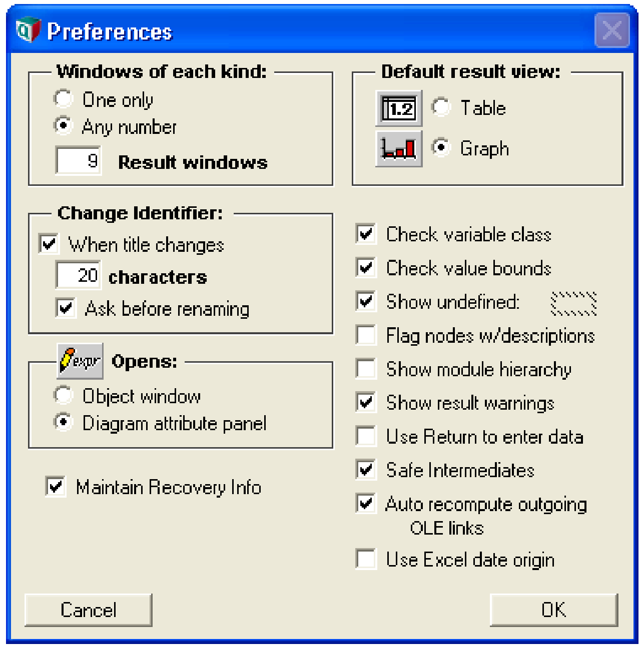
Library: A module that contains functions and sometimes variables. Read-in libraries are listed in the **Definition** menu below the built-in libraries, with a hierarchical submenu listing the functions they contain, giving easy access. See page 323.

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 page 309.

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 Interfaces for End Users."

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”page 313).

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” on page 49 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 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** (page 23) and select the definition text.
Diagram attribute panel	Open the **Attribute panel** (page 24) on the appropriate **Diagram** window and select the definition text.

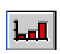
Maintain Recovery Info When this checkbox is checked (the default), Analytica saves each change to a recovery file, starting from the last point at which the model was saved. If the application terminates unexpectedly due to a software or hardware problem, the next time you start Analytica, it detects 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 can 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** (page 30) for a variable.

	Display result as a table.
	Display result as a graph.

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

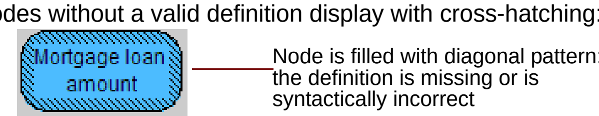
Checkboxes

Check variable class Display a warning if:

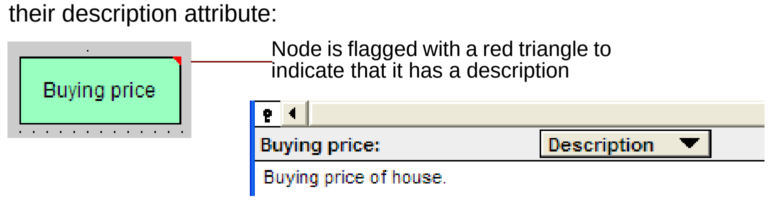
- A variable whose class is not **Chance** contains a probability distribution.
- A constant depends on another variable (other than indexes to an edit table).
- An index has a value that is not a one-dimensional array, or is an array with another index.

Check value bounds Evaluate check attributes for variables that have them. See page 115.

Show undefined Nodes without a valid definition display with cross-hatching:



Flag nodes w/descriptions Show a red triangle in the upper-right corner of nodes that have text in their description attribute:



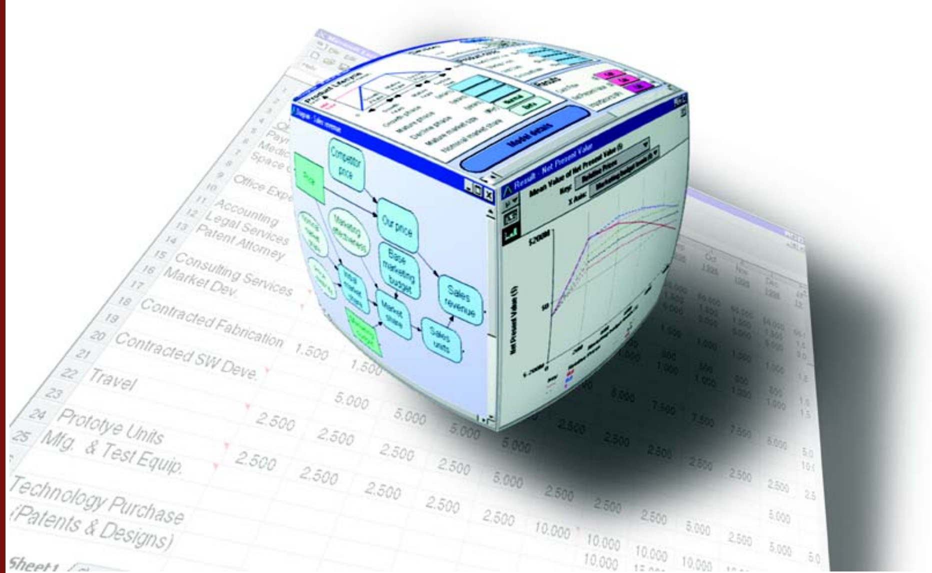
Show module hierarchy Show a hierarchy bar at the top of each **Diagram** window showing its nesting level. See page 304.

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 page 291.
Use Excel date origin	When this is unchecked, Analytica represents dates as a number indicating the number of days since January 1, 1904. When this is checked, is uses January 1, 1900, the same as Excel for Windows.
Maintain recovery info	When checked, Analytica keeps a log of all changes since the last time you saved your model. In the event of an application or system crash, or power outage, Analytica can usually recover all your changes since the model was saved. Having this on can slow things down if you are making changes to really large tables or images.

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.

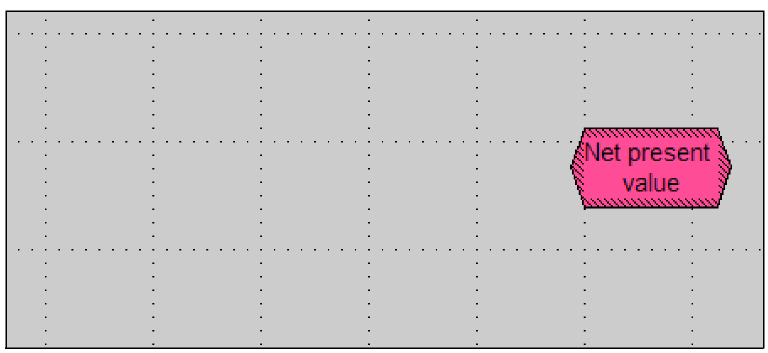
Creating a model

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 is 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 some 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 might 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 might 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 might want to consider the quality of life if you survive the treatment, and not just the probability of survival.

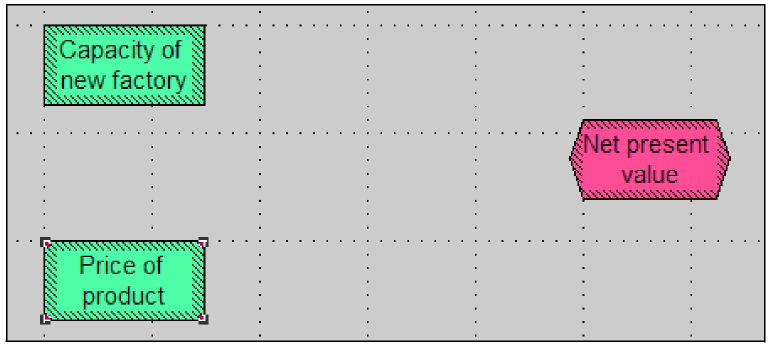
For an excellent guide on how to identify and structure objectives, see *Value-Focused Thinking* by Ralph Keeney (see “Appendix G: Bibliography”page 390).

Identify the decisions

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

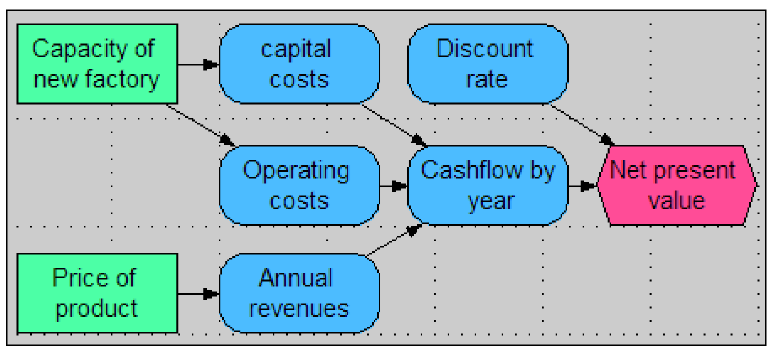
A ***decision variable*** is one that the decision maker can affect directly — which computer to buy, how much to bid on the contract, which medical treatment to choose, when to start construction, and so on. Occasionally, people want to build a model just for the sake of furthering understanding, without explicitly considering any decisions. Most often, however, the ultimate purpose is to make a better decision. In these 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. When 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 might 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 in model construction, to keep the focus on what matters. There can be a bewildering variety of variables in the situation that might 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 initially on the qualitative structure 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 might 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 might 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 might 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 might not express what you really intended, or might 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 might 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 are 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 might contain several errors that interact in confusing ways. Then you must search through much larger sections of the model to track them down. But if you have already tested the model components independently, you've already removed most of the errors, and it is usually 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 is 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 can be due to errors in the new model or the old model. When 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 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 might 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 can 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 might 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 **`NRI23`**.

It's also a good idea to document your model as you construct it by filling in the **Description** and **Units** attributes for each variable and module. You might find that entering a description for each variable and explaining clearly what the variable represents helps 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 run out of time, or 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 might 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 can 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 propagates 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**, page 268) 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 might 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.

When 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.

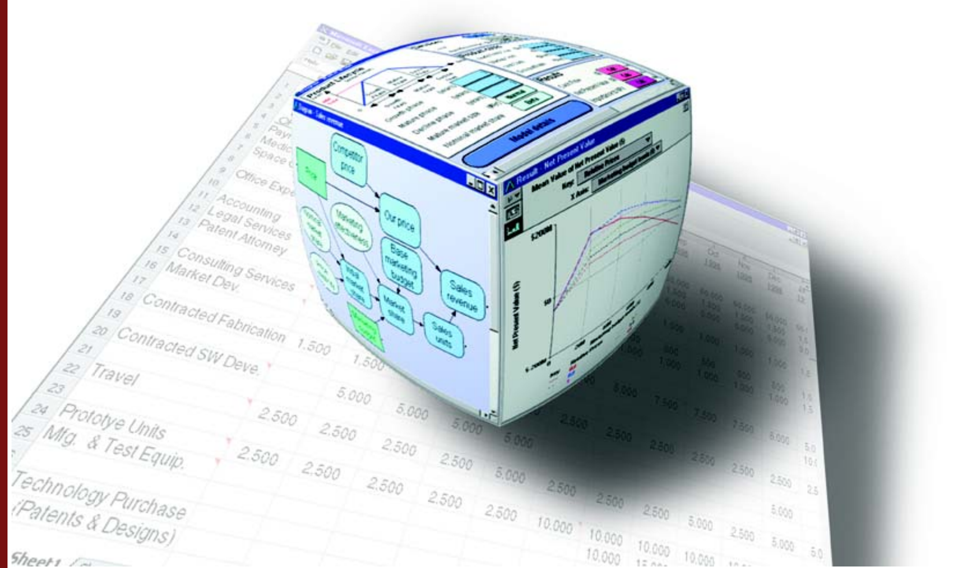
Simplify where possible

There's no reason that a model must grow successively more complex as you develop it. Sensitivity analysis might reveal that an uncertainty or submodel is just not very important to the results. In this case, consider eliminating it. You might 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.

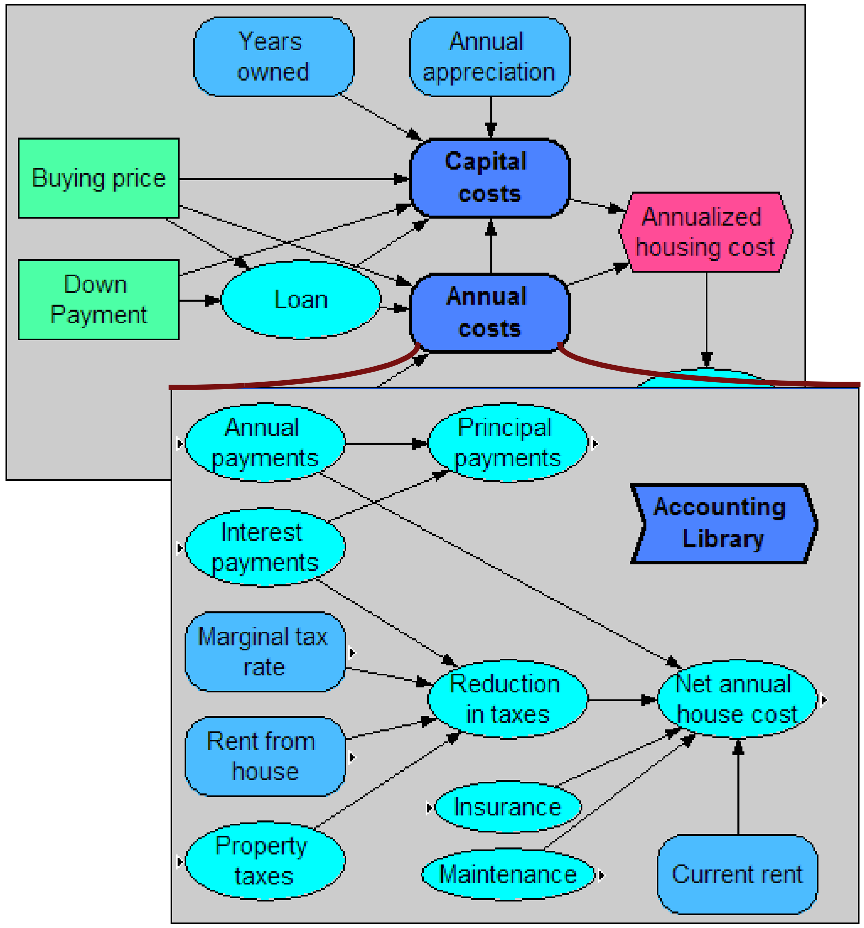
Simplifying a model has many benefits. It becomes easier to understand and explain, faster to run, and cheaper to maintain. These savings can afford you the opportunity to extend parts of the model that are more important.

Chapter 6 *Creating Lucid Influence Diagrams*

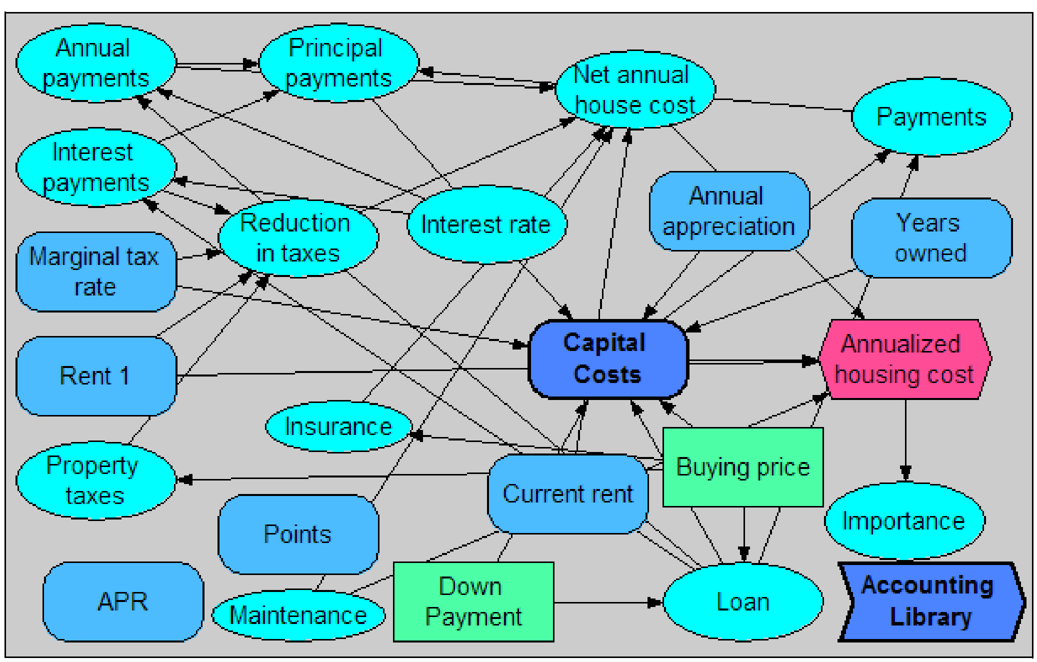
This chapter offers guidelines for creating influence diagrams that are clear and comprehensible by careful arrangement of nodes, well-designed module hierarchies, and judicious use of color. It also describes how to adjust and align nodes, and customize styles for nodes and diagrams. Options include 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!

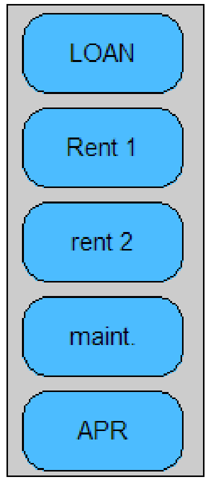


Guidelines for creating lucid and elegant diagrams

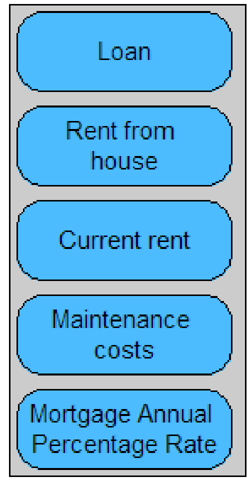
When aesthetics are involved, rules cannot be hard and fast. You can adapt and modify these guidelines to suit your particular applications and preferences.

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 lowercase) is clearer than all letters uppercase.



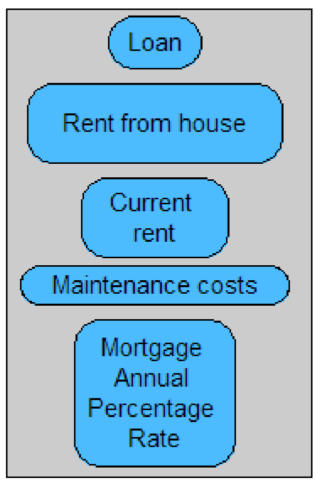
Poor object titles



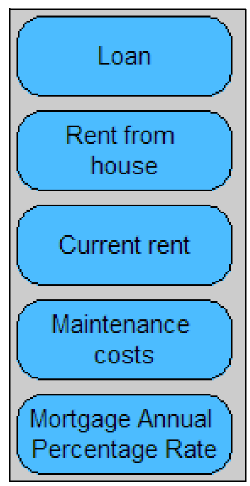
Good object titles

Use consistent node sizes

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



Inconsistent node sizes



Consistent node sizes

Node sizes will be uniform if you set the default minimum node size in the **Diagram Style dialog** (page 78) to be 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”page 72.

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

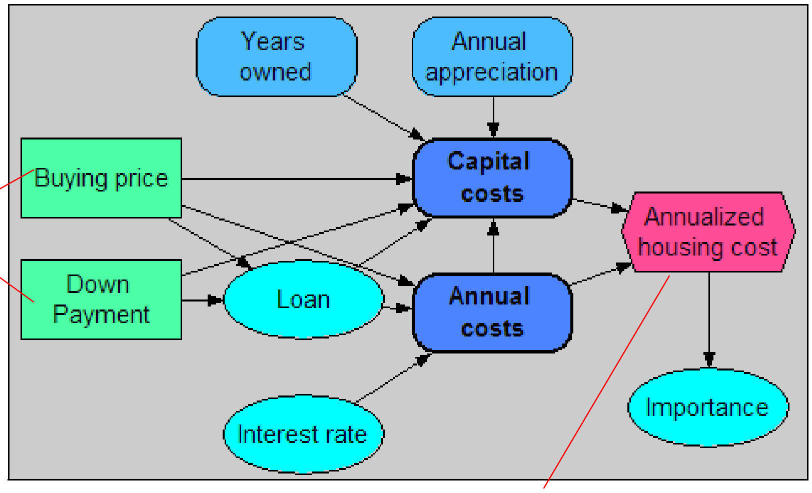
Sometimes, it is effective to make a few special nodes extra large or small. For example, start and end nodes, which can link to other models, often look best when they are very small. Or you can make a few nodes containing large input tables or modules containing the “guts” of a model larger to convey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

Decision variables on the left



Objective variable on the right

Tolerate spaghetti at first…

It can be difficult 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 by clicking the maximize box to have the diagram fill your screen. You might 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 can 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 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:

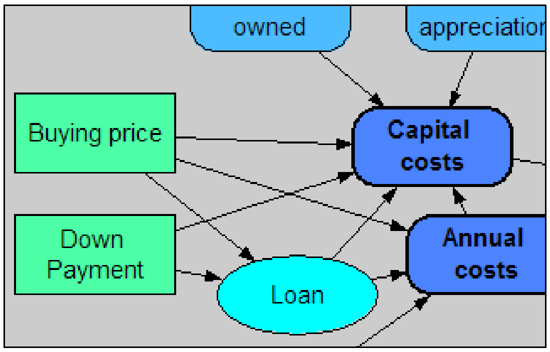
- **Make Same Size Width** — Sets all the selected nodes to the width of the widest node.
- **Make Same Size Height** — Sets all the selected nodes to the height of the tallest node.
- **Make Same Size Both** — Sets all the selected nodes to the width of the widest node and the height of the tallest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

---

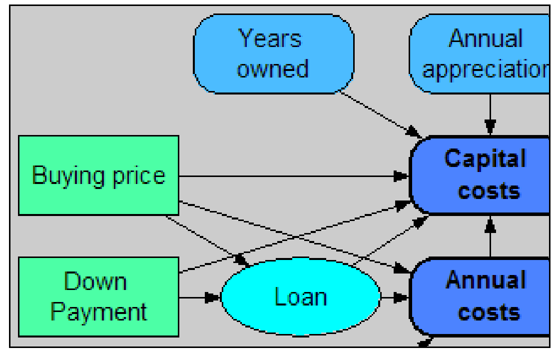
1. Or right to left for models in Arabic or Hebrew.

Align to the grid It usually looks best when the centers of the nodes are aligned along a common horizontal or vertical line, 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 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.



Poor alignment



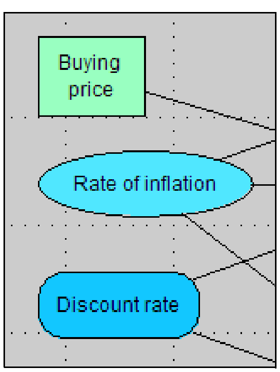
Good alignment

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

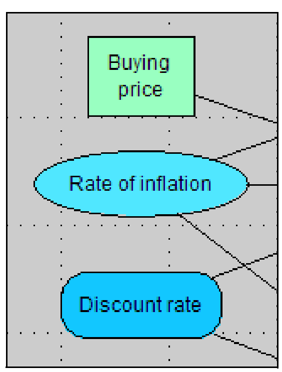
1. Select all nodes in the diagram with the **Select All** (*Control+a*) command from the **Edit** menu.
2. Select **Align Selection To Grid** from the **Diagram** menu (*Control+j*).

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

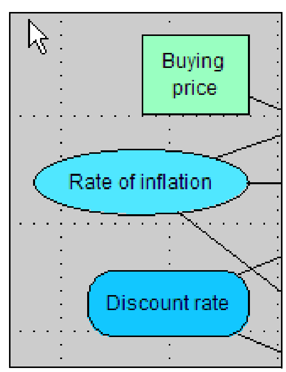
- Align the left edges.
- Align the centers left and right — this aligns the centers horizontally.
- Align the right edges.
- Align the left and right edges — this makes all the selected nodes the same width and aligns them so that their left and right edges match up. All nodes are set to the width of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



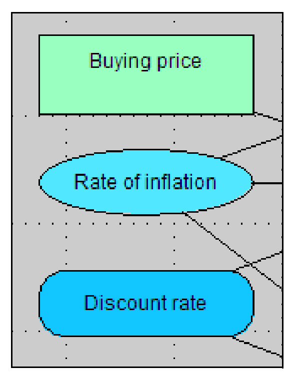
Align left edges



Align centers left and right



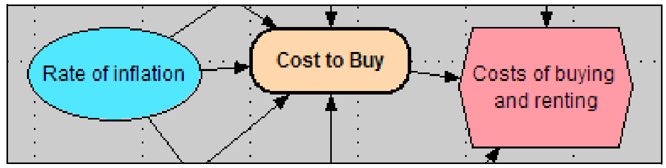
Align right edges



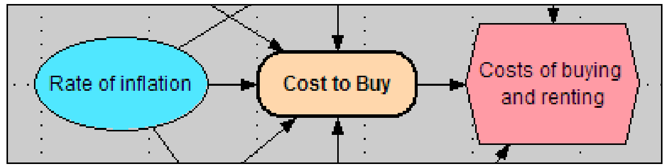
Align left and right edges

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

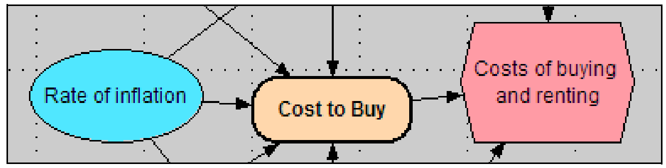
Align top edges



Align centers up and down



Align 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 the centers are evenly spaced vertically (**Space Evenly Across**) or horizontally (**Space Evenly Down**).

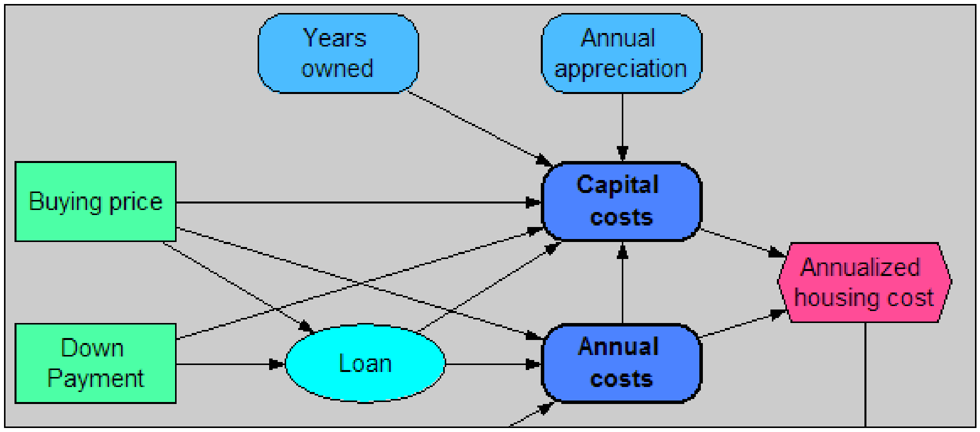
Choosing which node is in front By default, text and picture nodes are behind arrows, and arrows are behind all other types of nodes (decision, chance, variable, etc.). If nodes overlap, the more recently created node is on top of the older node. You can change this order by selecting a node(s) and using 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 might 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.

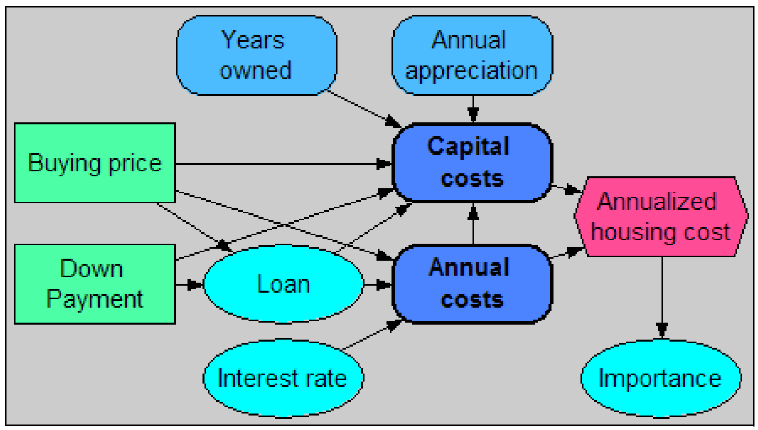
You can hide all of the arrows linking indexes, functions, or modules, or the grayed feedback arrows in dynamic models, using the **Set Diagram Style** command from the **Diagram** menu in the **Diagram Style dialog** (page 78). You can also hide the input or output arrows from each node individually, using the **Set Node Style** command in the **Node Style dialog** (page 79).

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



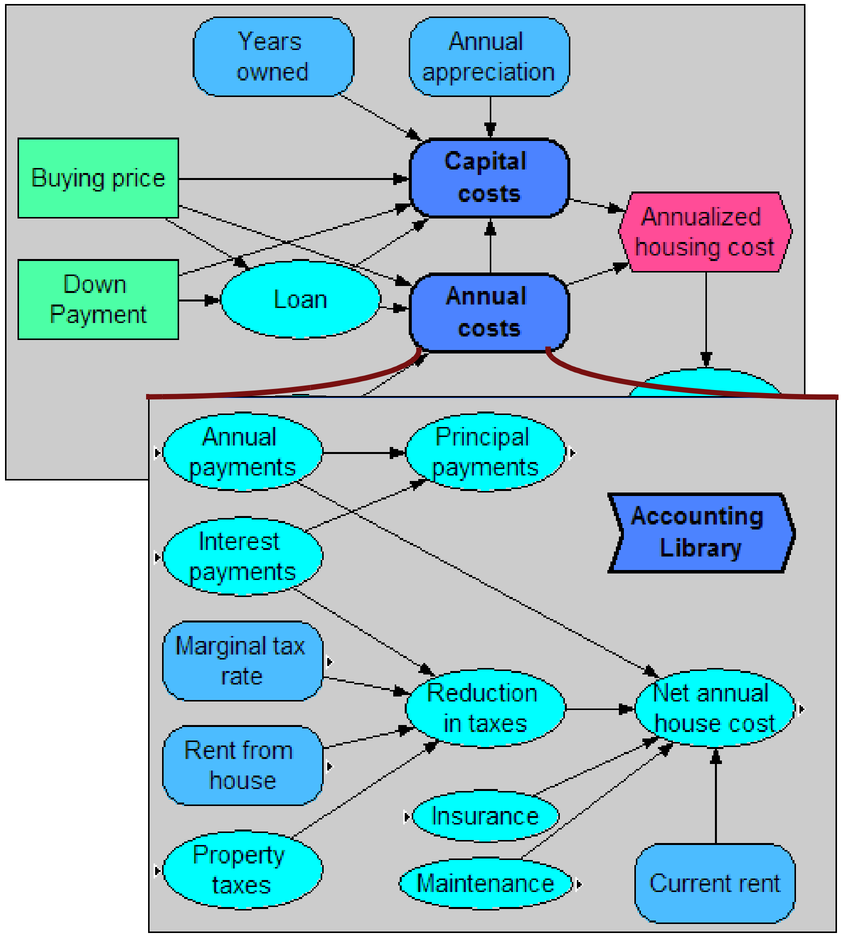
A compact diagram

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 might not be obvious which groupings work best. It is then often best to create all the nodes in a single large diagram. After drawing all the arrows, you might 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 **simplified model** (page 75) with the **spaghetti** (page 70). 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 than in the complicated model with 24 top-level nodes.

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

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 preferences.

Use colors judiciously

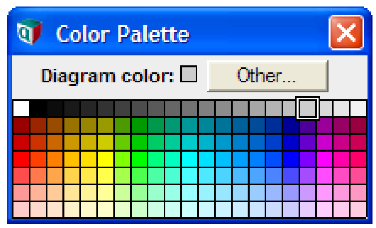
Light colors work best because its easier to see the black arrows and text over light backgrounds. Analytica's default colors provide a light neutral color for the background and a slightly stronger color for the nodes.

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.

Recoloring nodes or background

To apply colors to nodes or the background:

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

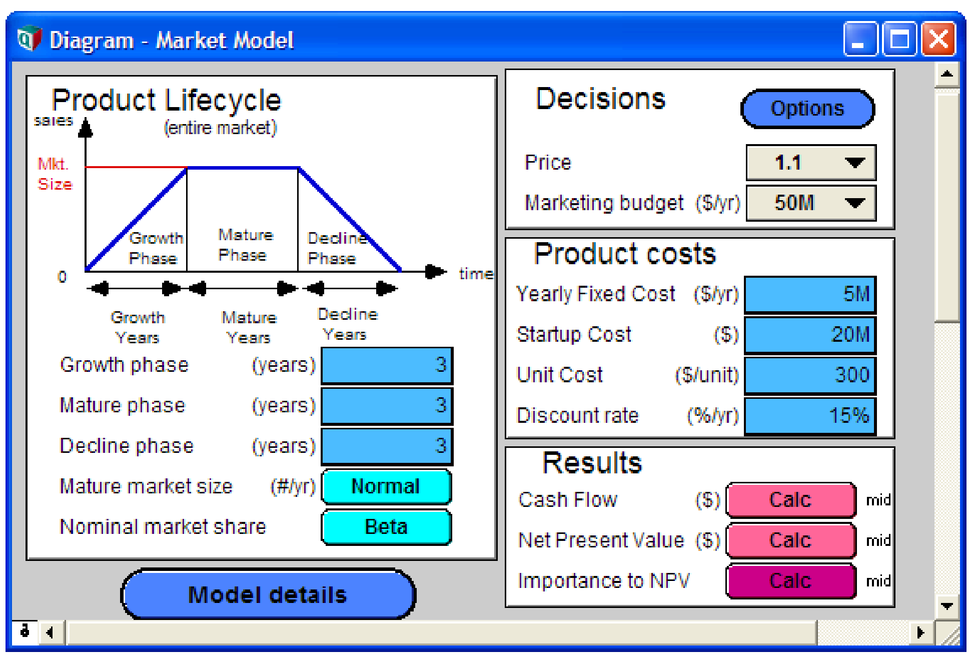


2. Select the node or nodes you want to recolor, or to recolor the background, just click the background. The current color of the node(s) or background appears in the single square at the top of the color palette.
3. Click a color square to apply the new color to the nodes or background.

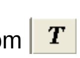
For a wider range of colors, click **Other** to display a full color chart.

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.



To create a grouping rectangle using a text box:

1. With the diagram in edit mode, create a ***text node*** by dragging from  on the node toolbar onto the diagram.
2. Type a title into the text node, or leave it blank as desired.

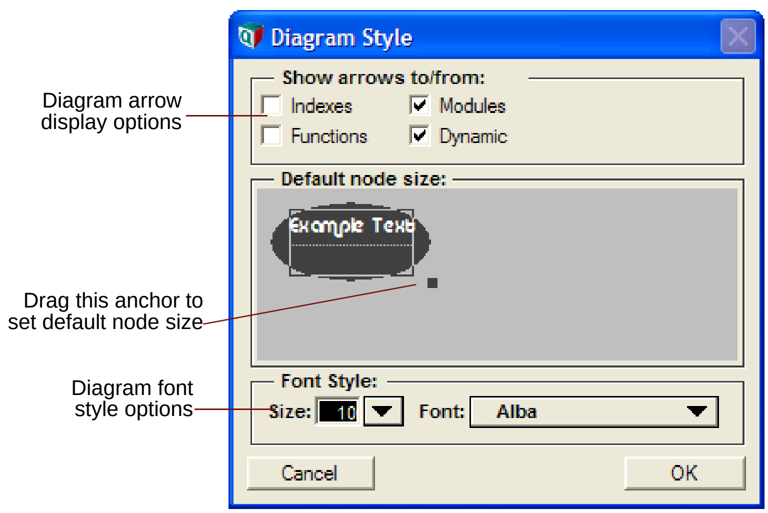
3. Move and resize the node to enclose the group of inputs or outputs. You might find it convenient to deselect the **Resize centered** option from the **Diagram** menu.
4. With the node selected, open the **Set Node Style** dialog from the **Diagram** menu, check the *Border* and *Fill color* options (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 also applies 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 display or hide arrows for specified node classes, set the node size, and customize the font size and typeface for nodes. 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 (page 282) controls the display of time-lagged dependencies to variables defined with *Dynamic*, usually displayed as gray arrows.

By default, diagrams show arrows to and from modules and dynamic, but not indexes and functions. Showing more arrows can clutter some diagrams 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

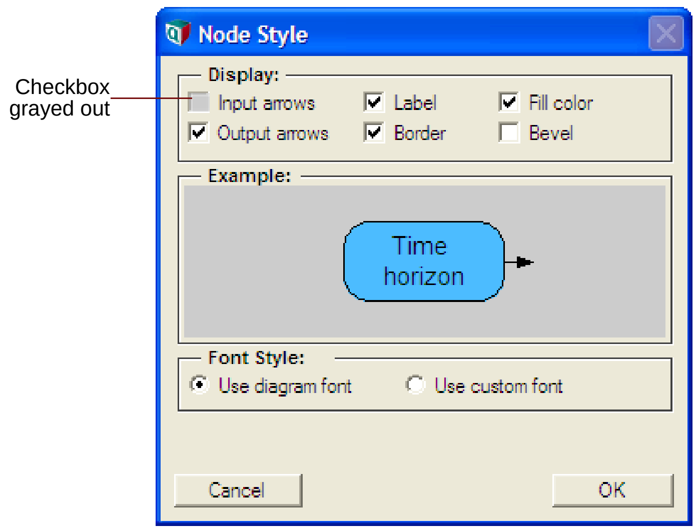
The **Diagram Style** dialog sets defaults for the diagram and for any modules contained in that diagram. You can override these defaults for particular nodes with the **Node Style** dialog (below), or for a submodule by using the **Diagram style** dialog for the submodule.

Node Style dialog

The **Node Style** dialog lets you customize the display of one or more nodes in a diagram. You can display or hide incoming and outgoing arrows, the text label, border, fill color, and bevel, and change the typeface and font size. 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 are 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 checkbox 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** (page 58) from the **Edit** menu.

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 allows you to leave out the background color, if any.

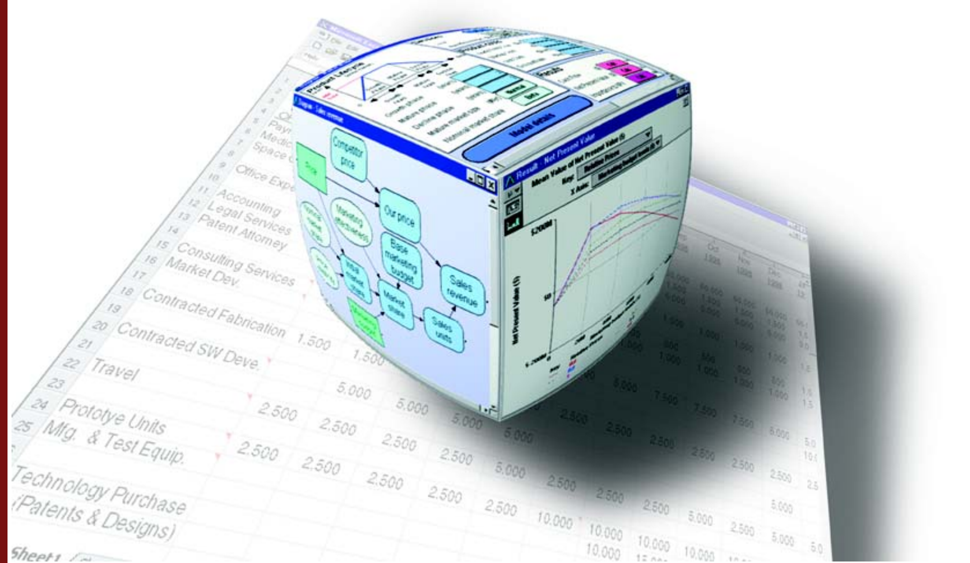
Exporting diagrams as images

To create an image file of your influence diagram, select **Export** from the **File** menu. The image can be stored in a variety of formats such as BMP, JPEG, TIFF, PNG, and EMF.

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 graphs.
- Select styles and options for graphs.



Number formats

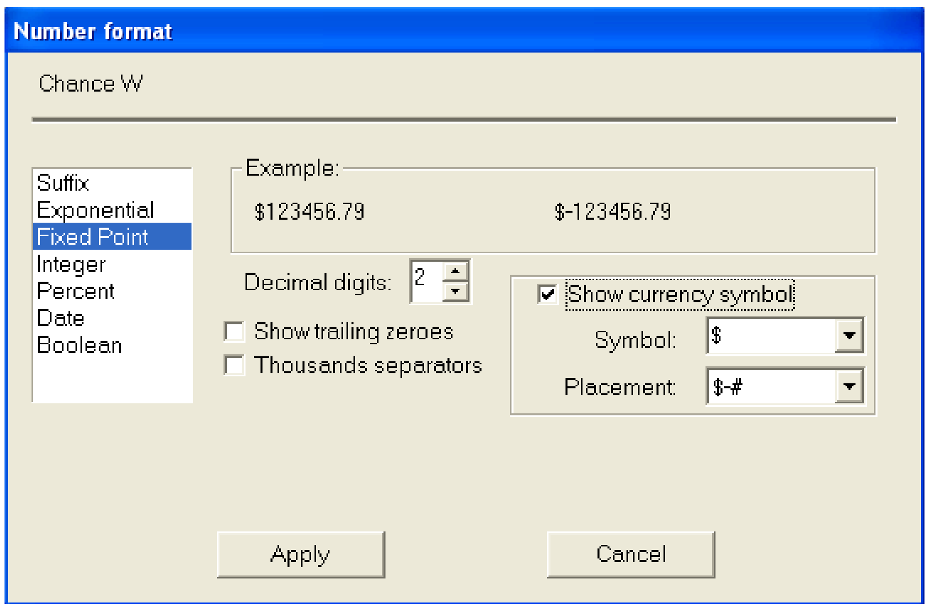
The **Number format** dialog lets you control the format of numbers to display in result tables and graphs — including dates and Booleans. You can select options like the number of decimal digits, currency signs, and commas to separate thousands. The default number format is *suffix*, which uses a letter following the number, such 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 field, 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 can 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 interprets 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 several 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 type you want from the list on the left (see “Format types” on page 82).
4. Select options you want, such as *Decimal digits*, *Show trailing zeroes*, *Thousands separators*, or *Show currency symbol*, from checkboxes, menus, and fields on the right. The options available depend on which format you selected.
5. View the example at the top of the dialog to see if the format is 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, where 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

Format	Description	Example
Integer	A whole number with no decimals	12346
Percent	A percentage	12%
Date	Date and date-times (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 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	Quadrillion	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 uppercase 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 — e.g., 2010, not 2.010K — which works better for years. 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 precision is 15 digits (9 for integers).

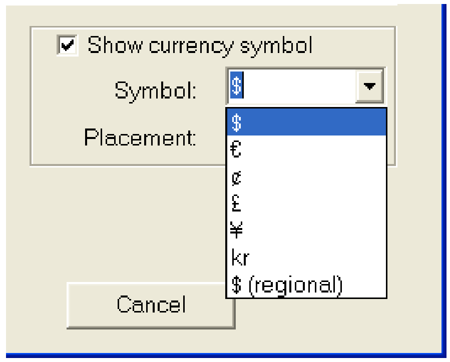
Number format options

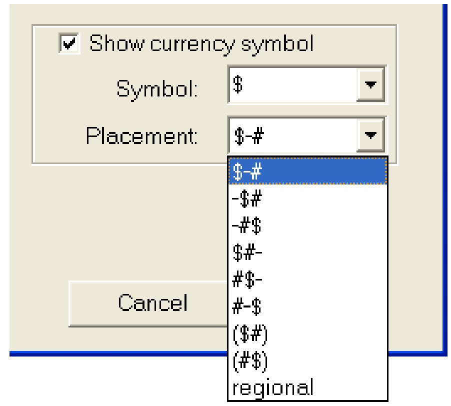
Decimal digits The number of digits to show after the decimal point.

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.



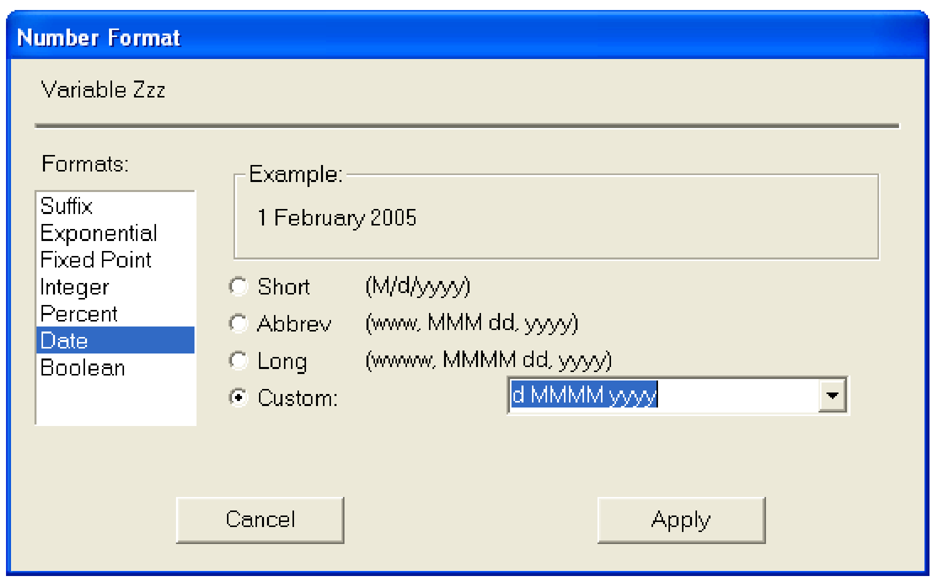


Placement controls the relative location of the currency symbol, e.g., **`$200`** or **`200DM`**, and whether to use a minus sign **`-$200`** or parentheses **`($200)`** to indicate a negative number.

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 fractional part, if any, represents the time-of-day as a fraction of a 24-hour day.



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, as shown in the table below.

Date format	Displays as
dd-MM-yy	05-08-08
'Q'Q YYYY	Q2 2008
www, d MMM yyyy	Thu, 5 Aug 2008
wwww, d of MMMM, yyyy	Thursday, 5 of August, 2008
d-MMM-yyyy hh:mm:ss tt	5-Aug-2008 03:45:22 PM
MM/dd/yy H:m:s	08/05/08 15:45:22

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
wwww	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
h	hour on a 12-hour clock	1, 2, ... 12
H	hour on a 24-hour clock	0, 1, ..., 23
hh	hour on a 12-hour clock as two digits	01, 02, ... 12
HH	hour on a 24-hour clock as two digits	00, 01, ... 23
m	minutes	0, 1, ... 59
mm	minutes as two digits	00, 01, ... 59
s	seconds	0, 1, ... 59
ss	seconds as two digits	00, 01, ... 59
tt	AM or PM	AM, PM

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 **1/5/2008** as **5 January, 2008** on a computer set to **USA region** or **1 May, 2008** elsewhere. Without the date format, it would interpret **1/5/2008** as (1 divided by 5) divided by 2008! A date-time can be entered as e.g., **1-May-2008 15:30:20** or **May 1, 2008 3:30:20 PM**.

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 9/12/2008, but in Denmark you might see 12.9.2008. 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 2008
```

where

```
Variable StartDate := MakeDate(2008, 2, 3)
```

Date numbers and the date origin

Analytica represents a date or date-time 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.

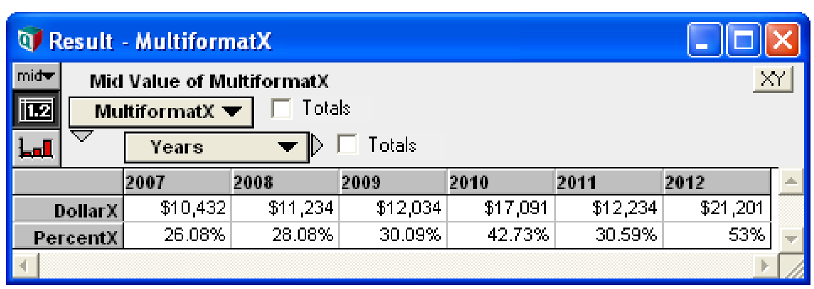
Date arithmetic and functions

You can simply add an integer *n* to a date to get the date *n* days ahead using the **MakeDate()**, **MakeTime()**, **DatePart()**, **DateAdd()**, and **Today()** functions (page 210).

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 might 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 MultiformatX := [DollarX, PercentX]
MultiformatX →
```



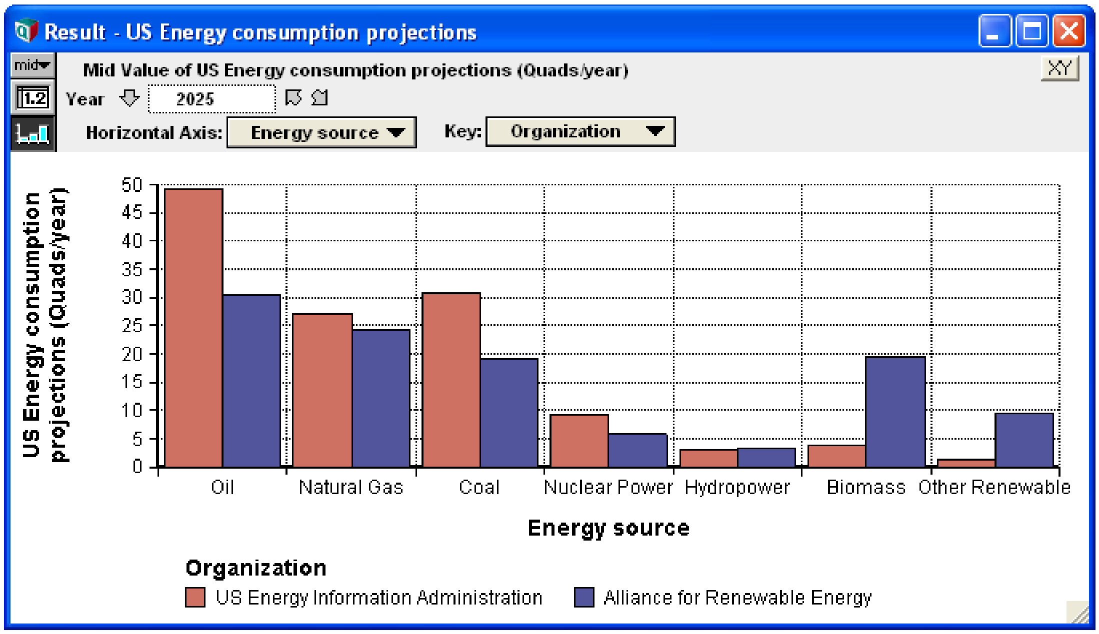
	2007	2008	2009	2010	2011	2012
DollarX	$10,432	$11,234	$12,034	$17,091	$12,234	$21,201
PercentX	26.08%	28.08%	30.09%	42.73%	30.59%	53%

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`.

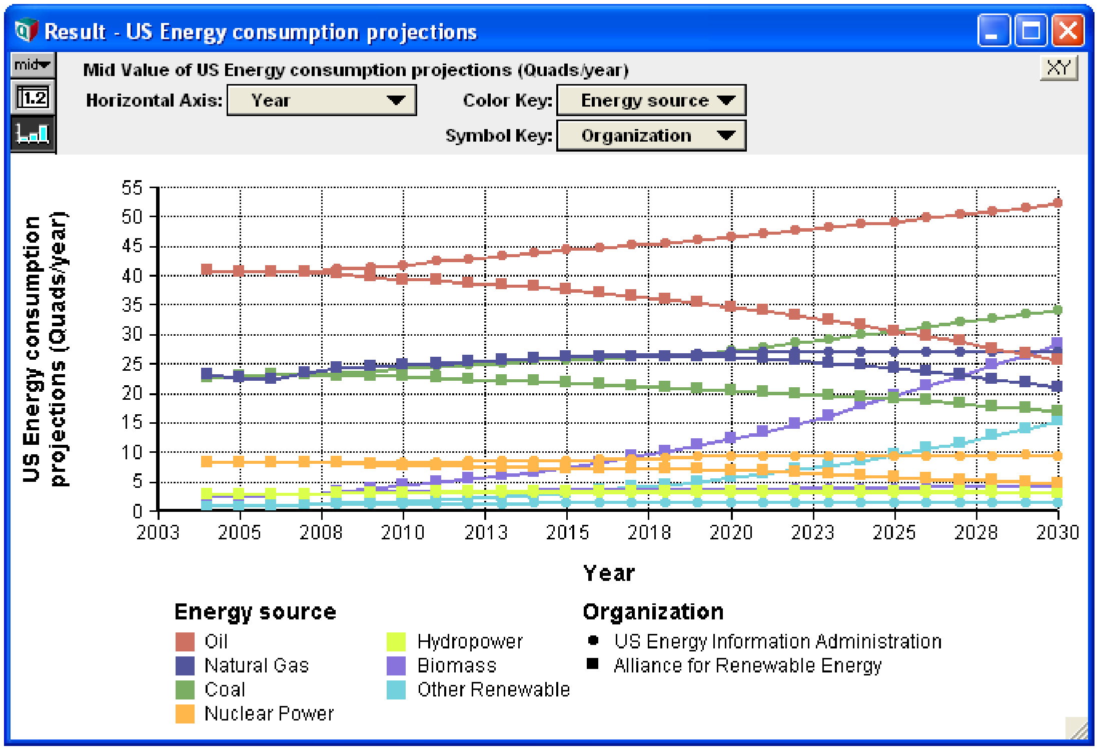
Graphing roles

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 numerical scale or a discrete numerical or categorical scale — except for symbol size, which must be numerical.

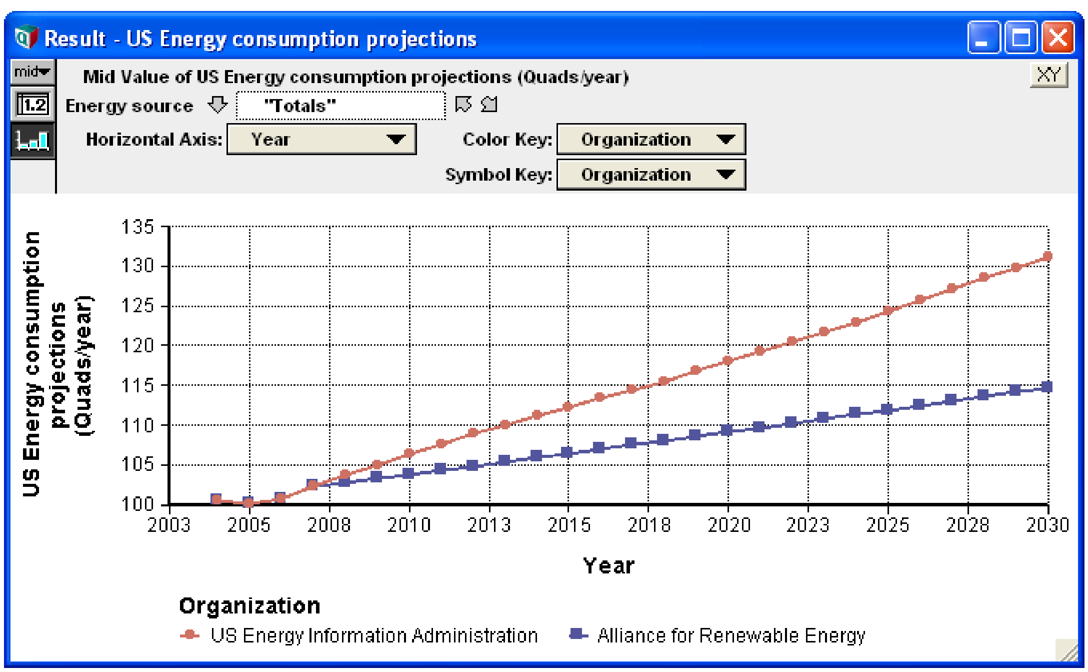
This example shows projections of U.S. energy consumption made by two organizations, the U.S. Energy Information Administration (actual) and the Alliance for Renewable Energy (fictional). The horizontal axis is set to **Energy source**, the key (color) is set to **Organization**, leaving the **Year** 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 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 89) or for **XY graphs** (page 98).

Horizontal axis The horizontal direction, labeled 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 the **Graph setup dialog** (page 89). In the table view, it becomes the row headers.

Color key and symbol key If you check *Use separate color/symbol keys* in the **Graph setup dialog** (page 89) (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 graphing 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

The **Graph setup** dialog 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:

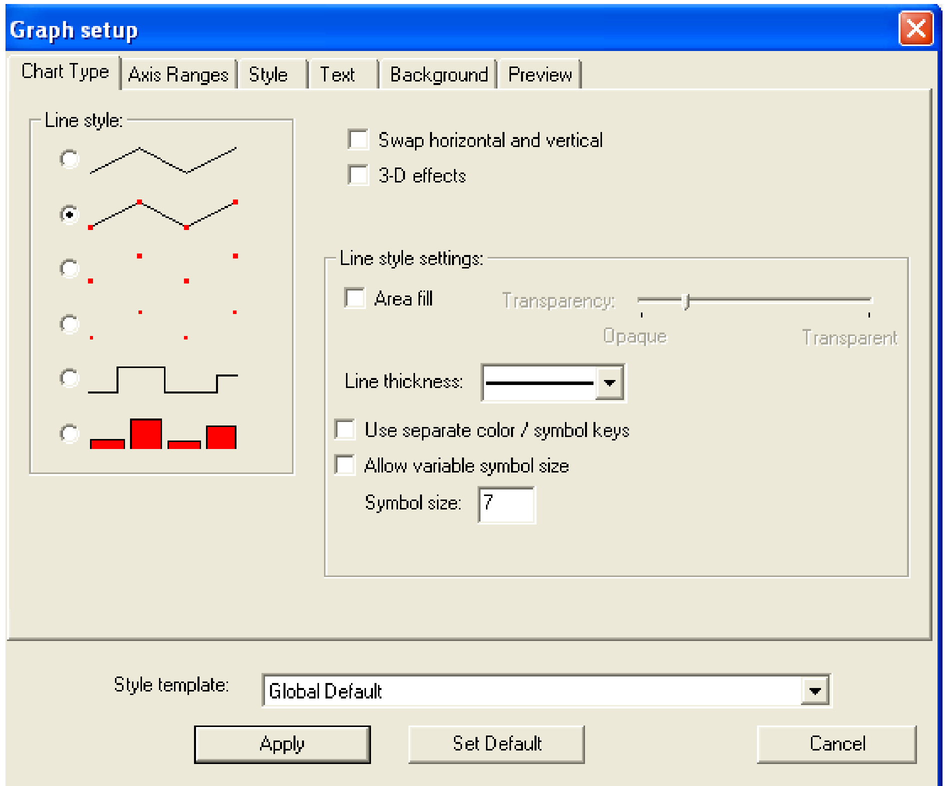
- Select **Graph Setup** from the **Result** menu.
- Select **Graph Setup** from the right mouse button menu.
- Double-click the graph in the **Result** window.

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.

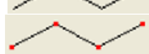
Chart Type tab

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.

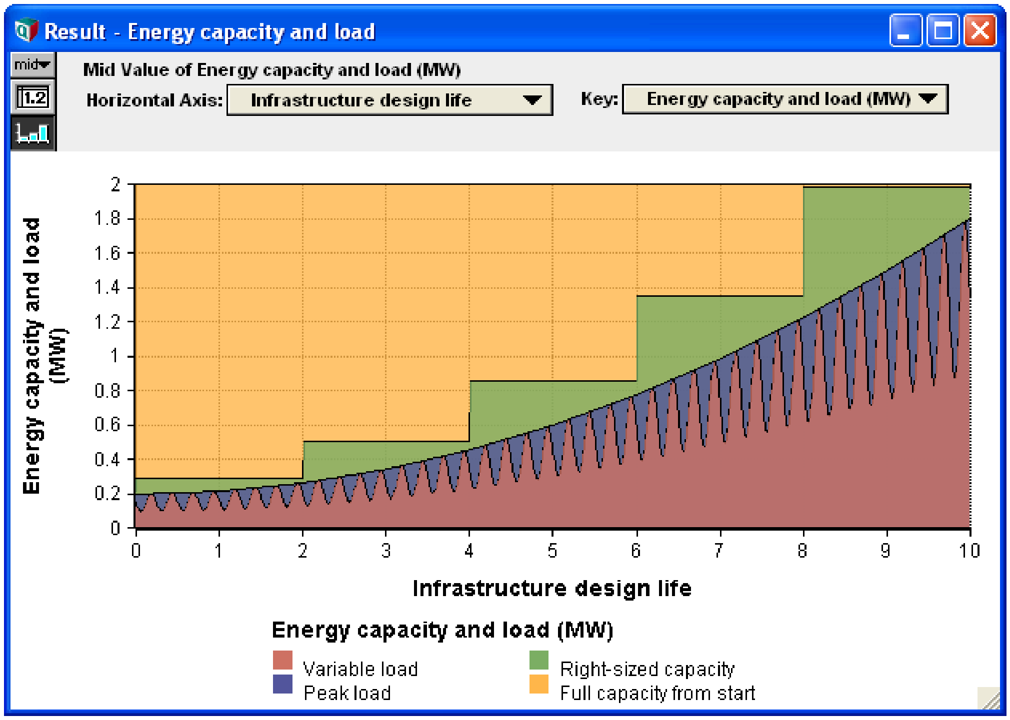
 A bar centered on each x value, with height showing the y value. Forces the graph to be discrete.

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.

3D 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 has 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 obscure earlier ones. Here's an example.



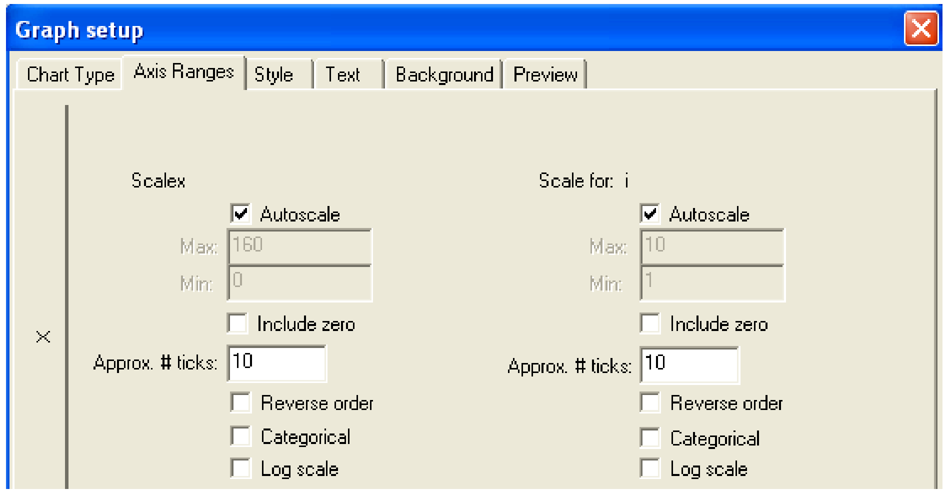
- *Transparency:* Drag the cursor to change transparency of fill colors between opaque and transparent. Transparency lets you see fill lines and areas that would otherwise be obscured behind others.
- *Line thickness:* Select the thickness of lines to display. (Only for styles that show lines.)
- *Use 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 then displays 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.

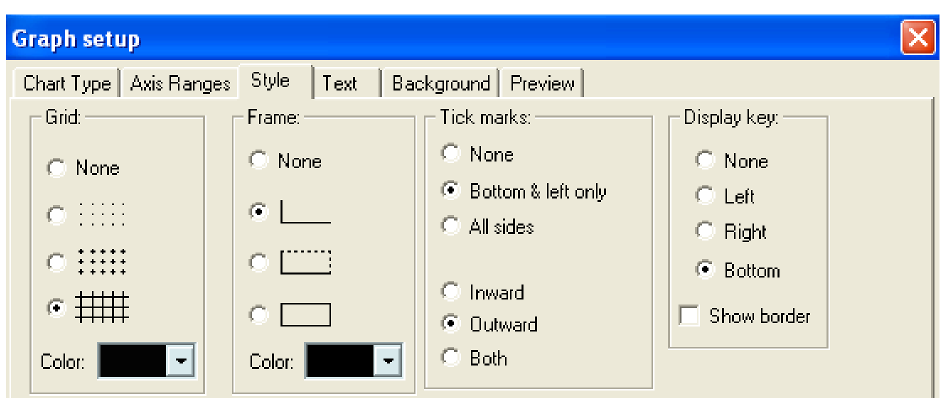
Categorical Treat this axis as categorical. Usually, Analytica figures out the quantity is categorical without help. Occasionally, if the values are numerical, you might want to control it yourself. See “Probability density and mass graphs”page 234.

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.

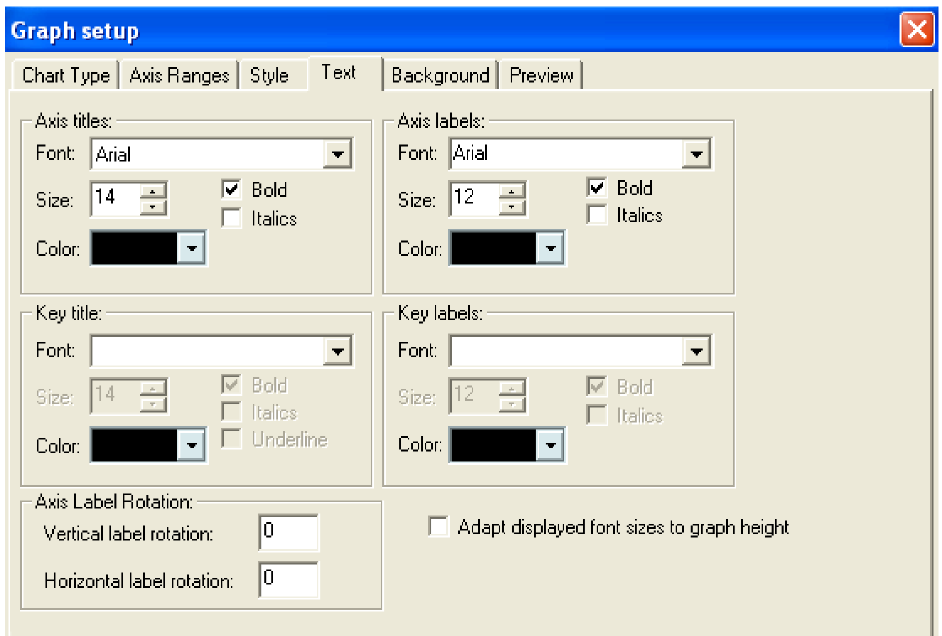
Frame Select the radio button to control the display of the lines framing the graphing area. You can also select the color for the frame. It is usually best to make the frame the same color as the grid, or a darker shade of the same color.

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* checkbox 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).

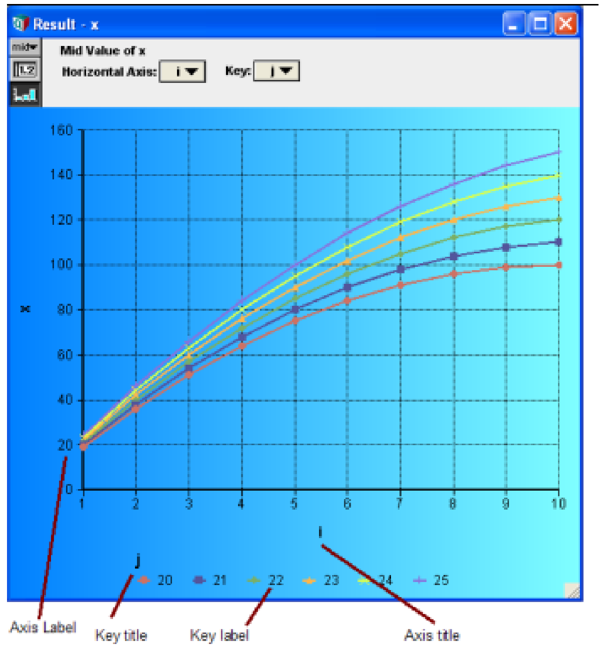


Font Select the font family. Graphic designers recommend using the same font for all text, which you can easily do by leaving all except axis titles as "(Same as axis titles)."

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.

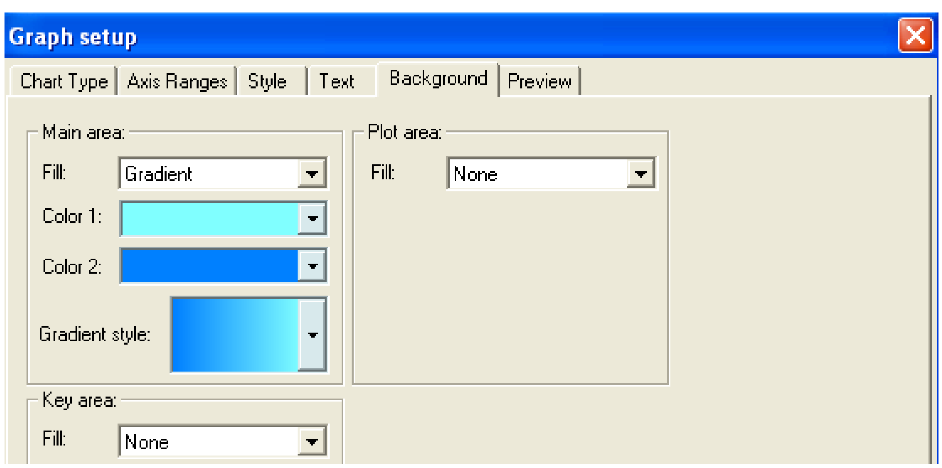


Axis Label Rotation Enter a number from -90 to 90 degrees to rotate the labels for each axis. For example, for a bar graph with many long labels along the horizontal axis, they won't all fit. By rotating them by 45 or 90 degrees, you can make them all fit without getting truncated.

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 show the same fill settings (if any) as the **Main area**.



Fill Select from:

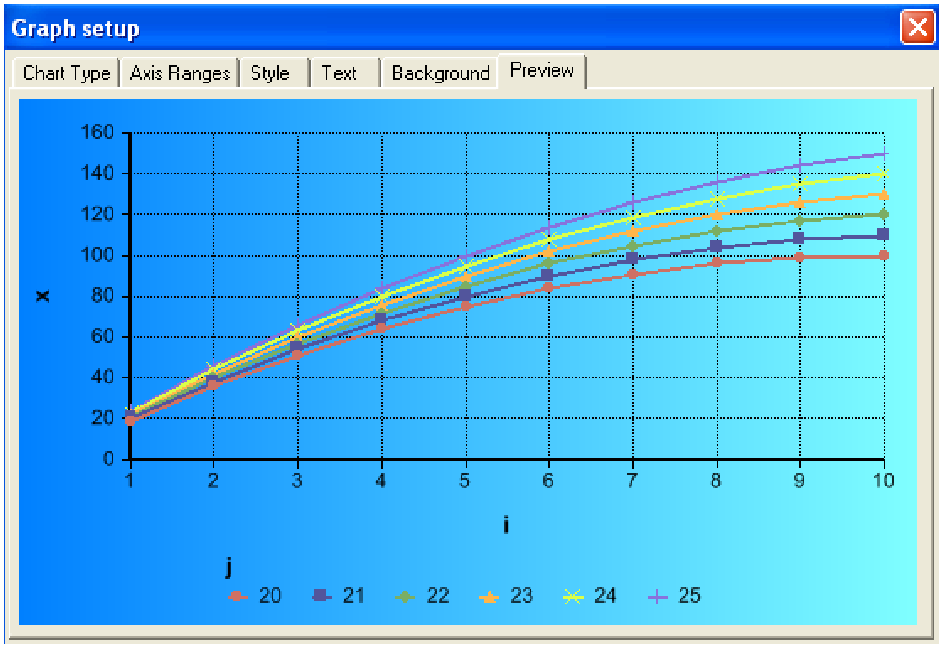
- **None:** No fill. Default to blank (white) background.
- **Solid:** Use a solid fill with the selected **Color 1**.

- **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

Distinctions regarding whether your results are treated as being ***categorical***, ***continuous***, or **discrete** impact how the data is plotted. Analytica usually infers the appropriate distinctions, but occasionally you might need to provide explicit setting information.

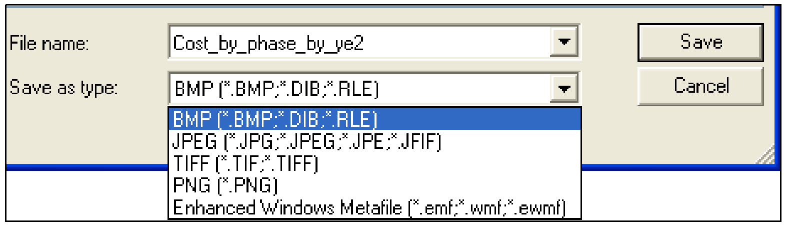
The discrete vs. continuous distinction is determined by the domain attribute, and determines whether probability plots are density and cumulative density plots (continuous) or probability mass and cumulative probability (discrete) plots.

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, TIFF, 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 4.1 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 are 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 want 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 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 this template 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**.

Tip Any changes you make to a template 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 setting changes 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 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 applies to any sample plot, but not to other uncertainty views **Mid**, **Mean**, **Statistics**, **PDF**, or **CDF**. Thus, you can 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

Analytica 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 can change the plot from a bar graph to line graph, and uses 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 has 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 defaults are 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 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.*

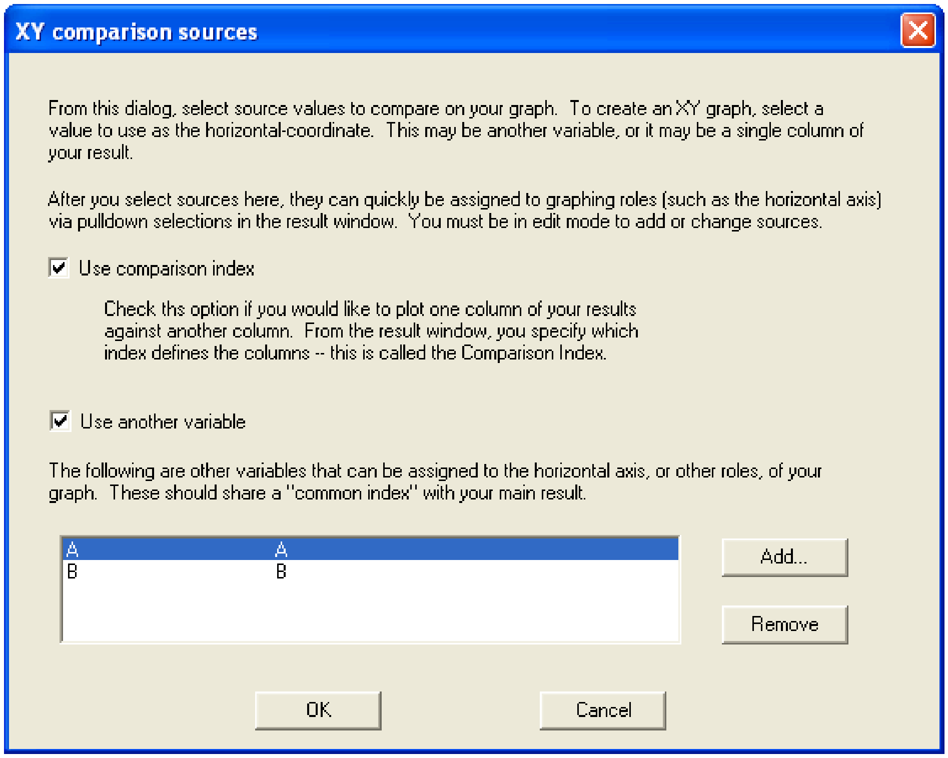
XY comparison

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. In contrast, with

XY comparison 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”page 277 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 shows the **Comparison Index** menu from which you can select the index you want. The **Vertical Axis** and **Horizontal Axis** menus then offer slices from the comparison index so that you can choose which two slices to plot against each other.

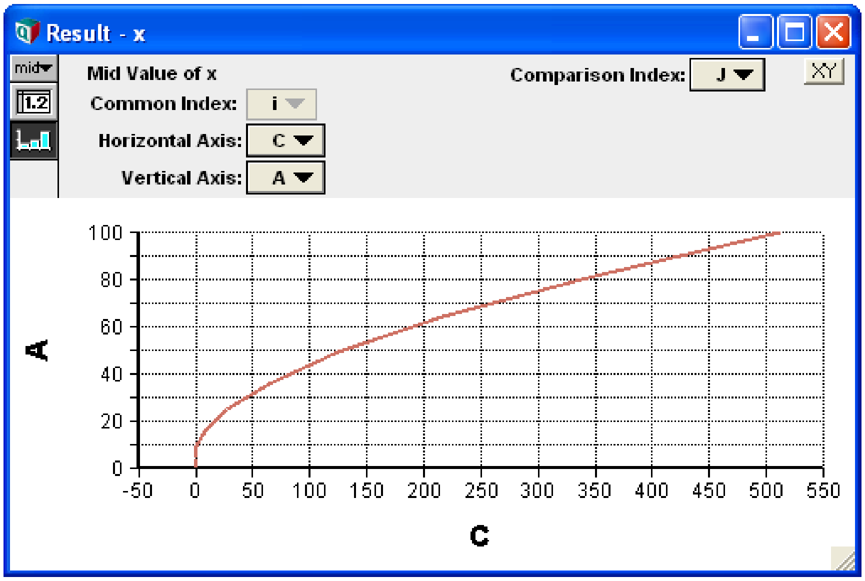
Use another variable Check this box if you would like to compare the base variable by plotting it against one or more other variables (or simple expressions). When you check it, the following items appear:

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 This 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, **Common Index** is an index in common to the base variable and other variable(s). If the variables have more than one index in common, **Common Index** is 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** shows the index(es) of the base variable not selected for **Comparison Index**. **Common Index** is a menu if the variable has more than two indexes — leaving more than one for **Common Index**.

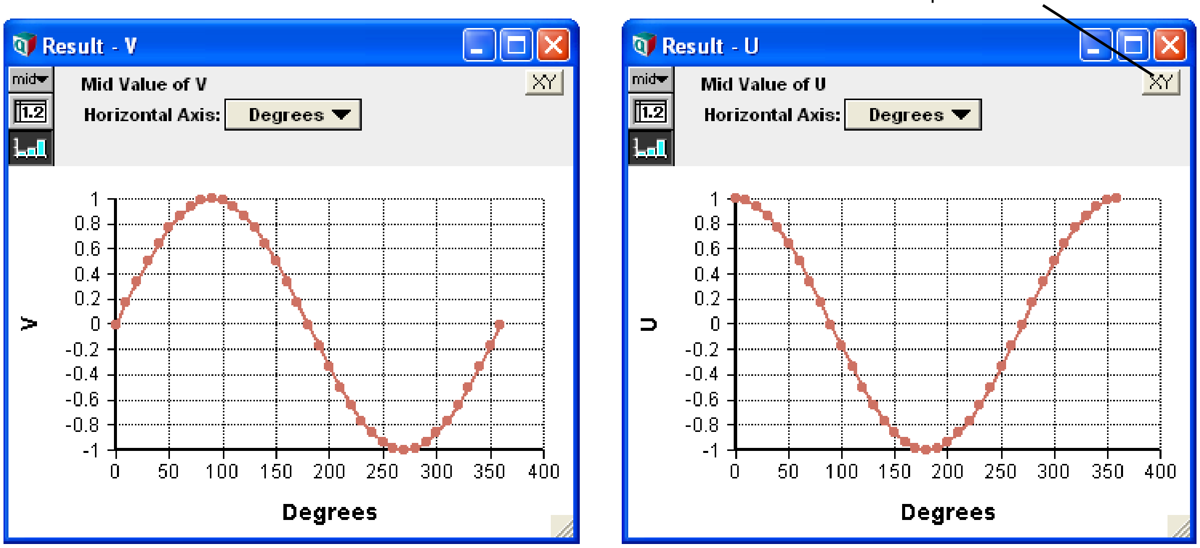
Example: Plot one variable against another

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**.

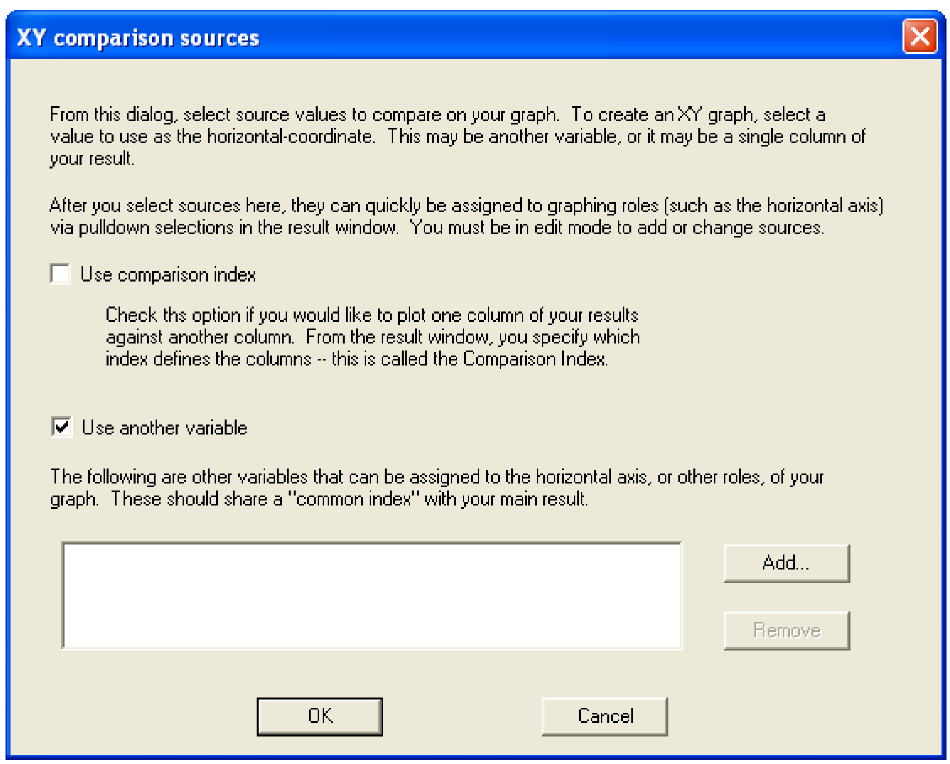
XY comparison button



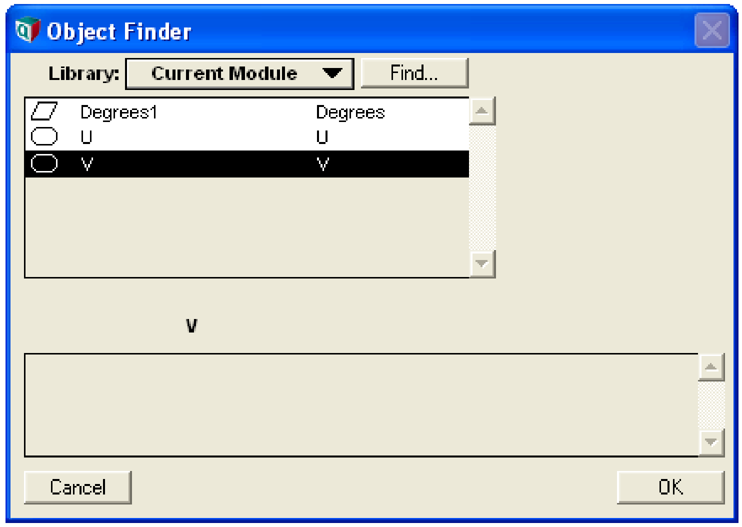
For these graphs, we selected the *symbol plus line* **line style** (page 90) from **Graph setup** to show the data points for each value of **Degrees**.

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

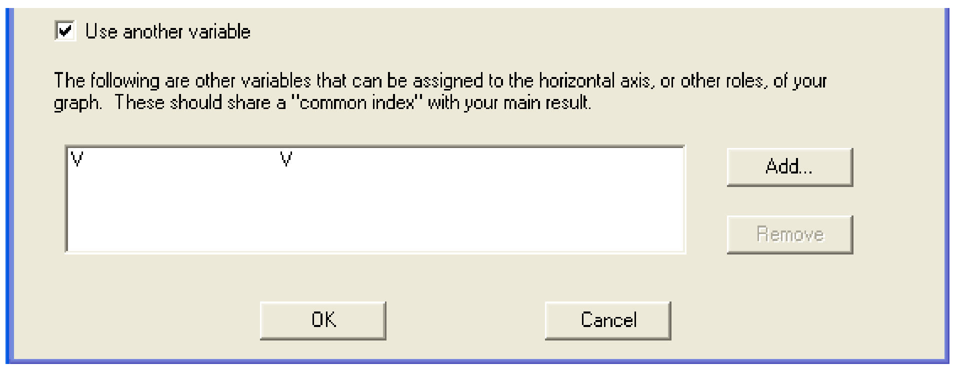
1. Change to *edit mode*. In the **Graph** window for **U**, click the **XY** button in the top-right corner (above) to open the **XY Comparison sources** dialog.



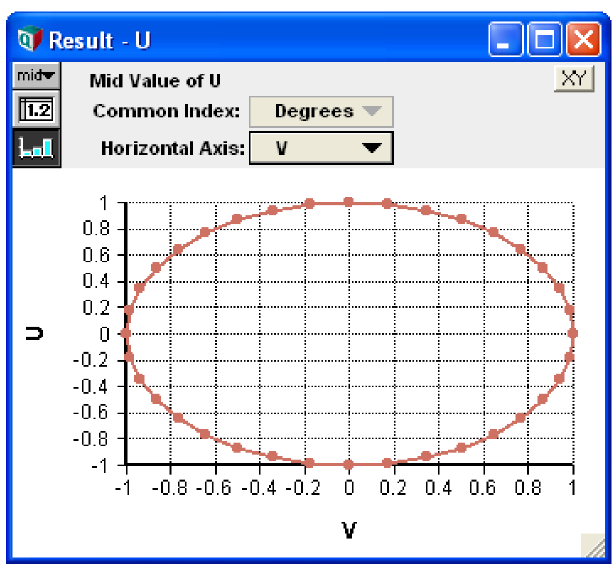
2. Select the checkbox *Use another variable*.
3. Click the **Add** button to open the **Object Finder** dialog.



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



5. Click **OK**.



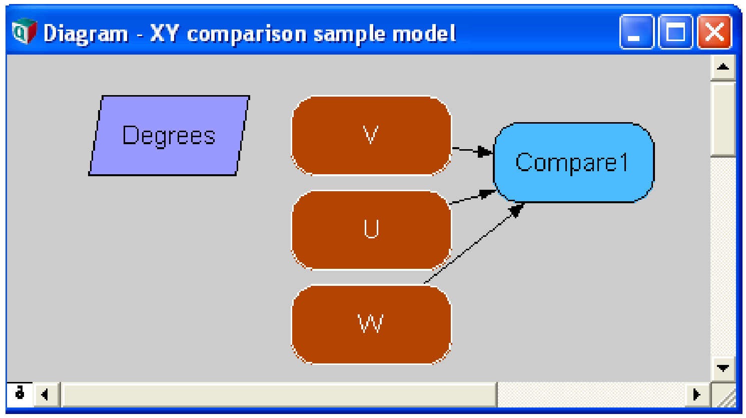
The graph of **U** now plots the values of **U** on the vertical (y) axis against corresponding values of **V** on the horizontal (x) axis. By "corresponding" we mean for each value of **Degrees**, in the **Common Index**. If **U** and **V** had more than one index in common, it would show a menu from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

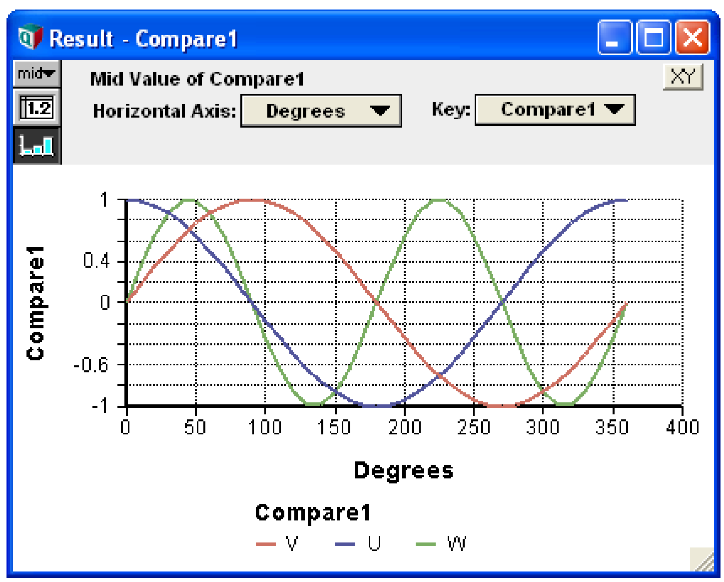
```
Variable W := Sin(2*Degrees)
```

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.

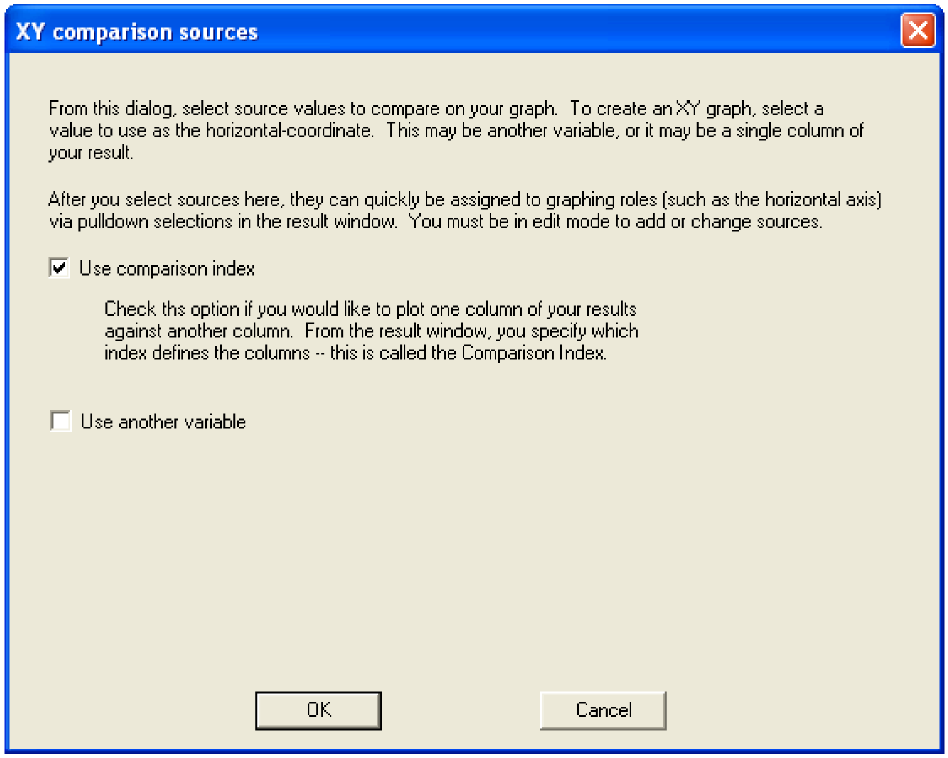


1. Select the three nodes for the variables to compare, **U**, **V**, and **W**, and click **Result** (*Control+r*).
2. When it prompts *"Do you want to compare more than one result?"* click **OK**.

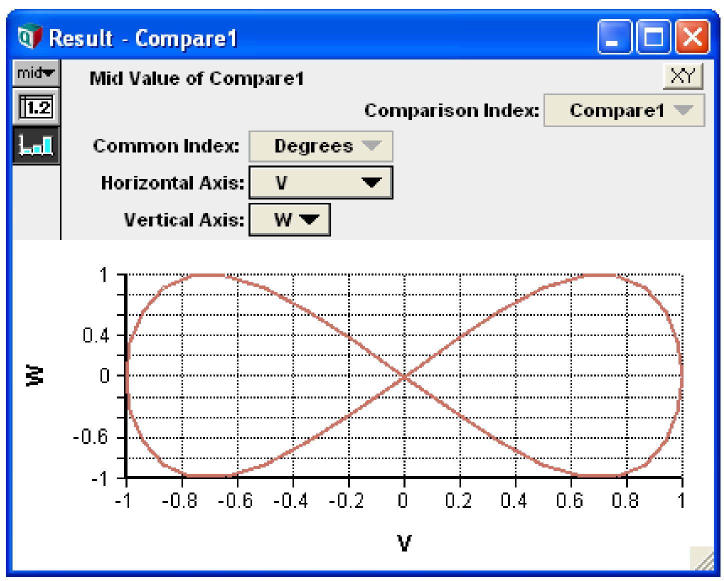
   It creates a new variable **Compare1**, and shows the standard (not XY) graph comparing **U**, **V**, and **W** against index **Degrees**.



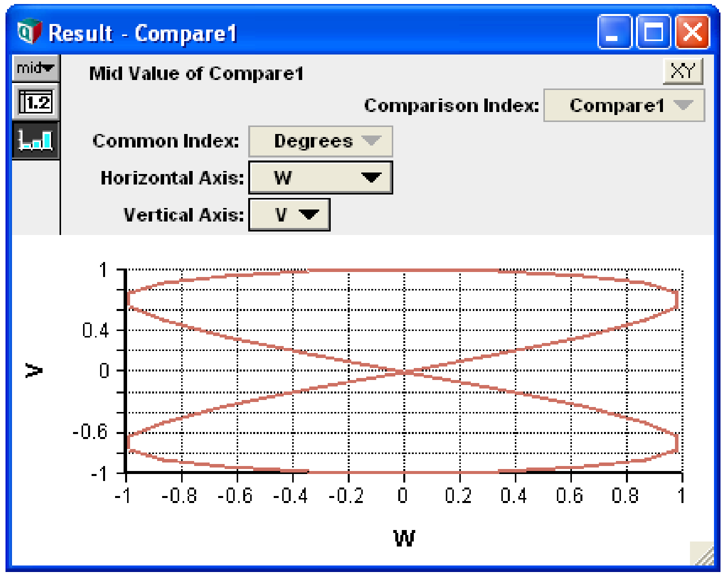
3. Make sure you are in *edit mode*. In the graph window for **Compare1**, click the **XY** button in the top-right corner to open the **XY comparison sources** dialog.



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



This sideways figure 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 8 the right way up.

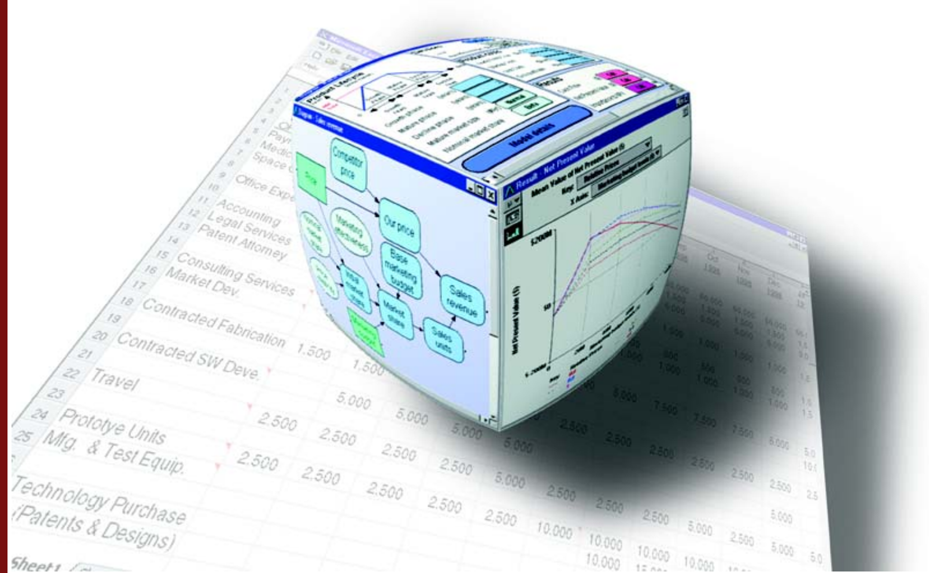


You can also select **Degrees** from the **Horizontal Axis** menu to revert to a standard (non XY) graph of the selected variable against **Degrees**.

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



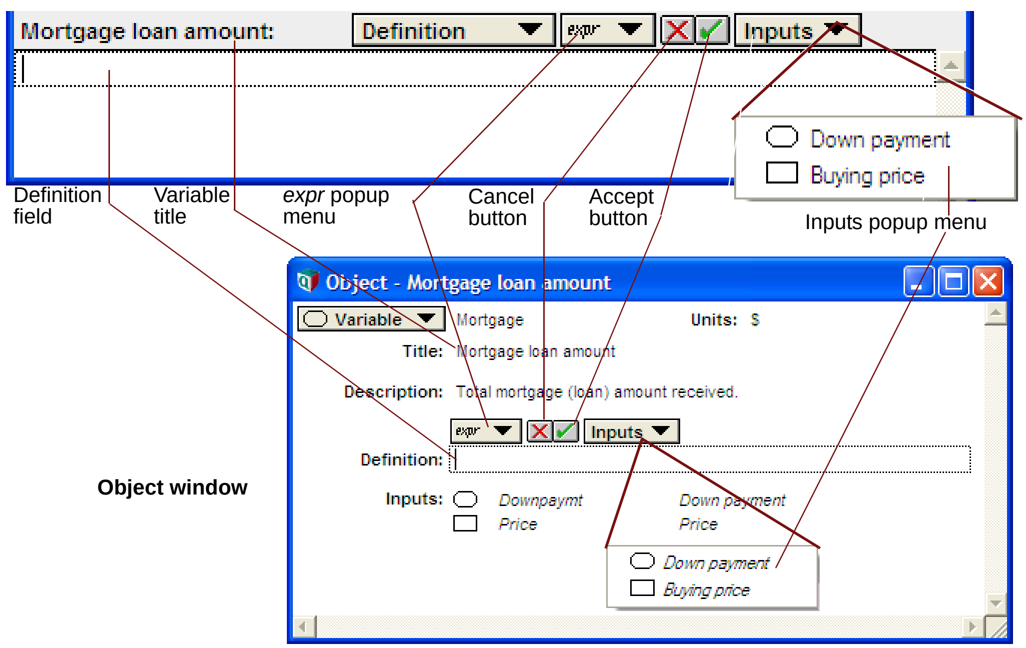
This chapter introduces the tools for creating and editing mathematical models by giving each variable a formula that defines how to compute its value in its ***definition***. The definition of a variable can be a simple number, text, a probability distribution, or a more complicated expression. It can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

Attribute panel



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. This 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** (page 111).

Special editing key combinations

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

Key or key combination	Action
double-click	Selects the entire identifier containing the cursor.
option-click a node	Inserts identifier of the node at the cursor position.
left-arrow ←, *right-arrow* →	Moves cursor one character left or right.
up-arrow, *down-arrow*	Moves one line up or down.
Control+left-arrow, *Control+right-arrow*	Moves to the beginning or end of the next word or identifier.
Alt+Control+left-arrow, *Alt+Control+right-arrow*	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(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 can 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.

Tip Analytica 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 in the **Object Finder dialog** (page 112).
- 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 is inserted at the caret position. This shortcut isn't available from the **Object** window or for nodes is different modules.

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

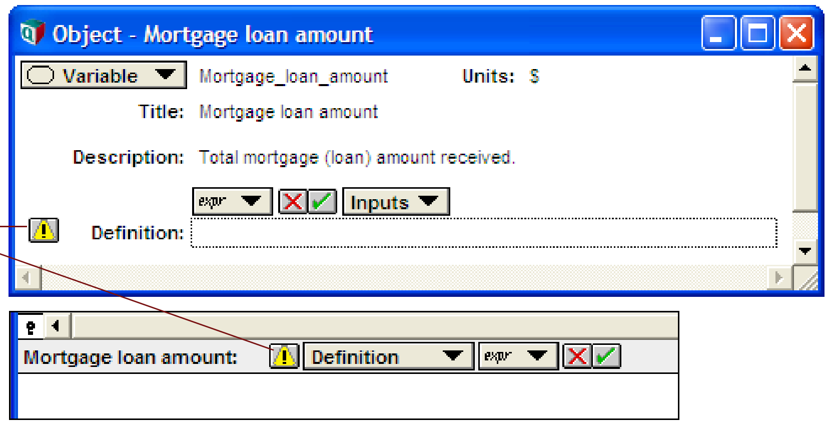
- Select **Paste Identifier** from the **Definition** menu to open the **Object Finder dialog** (page 112).
- Select the function from its library in the **Definition menu** (page 114).

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.

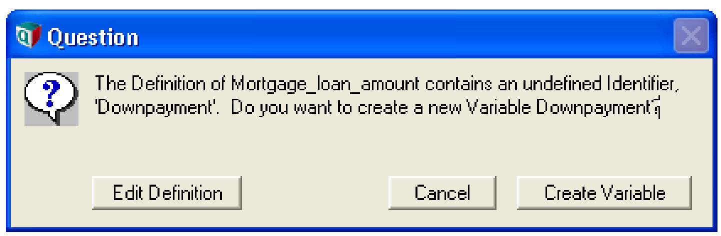
Click the cancel button to cancel your changes.

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

Definition Warning icon



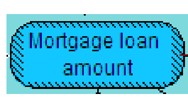
A definition's syntax check can reveal **syntax errors** (page 387). For example, if a definition contains text that is not an identifier, the following dialog appears.



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.



Cross-hatch pattern: the definition is missing or is syntactically incorrect

After you enter a valid definition, the cross-hatching disappears and the node becomes clear.

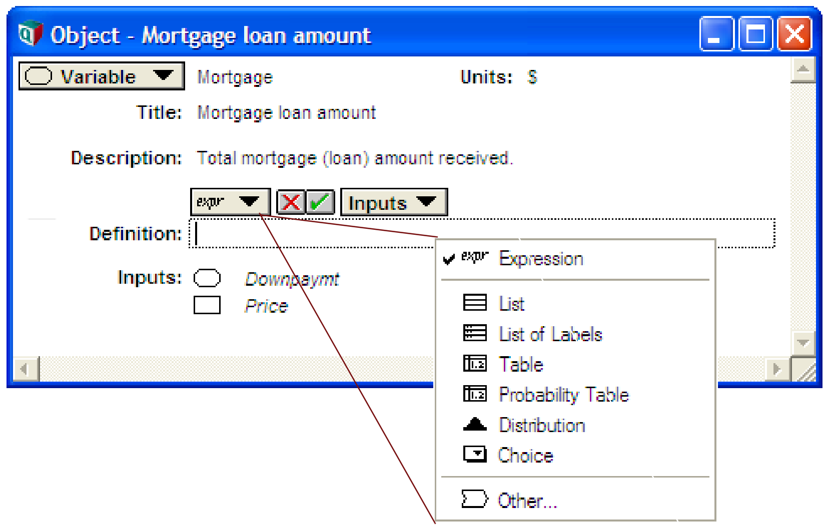
Node is clear: the definition is syntactically correct

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, and removes an arrow from any variable you didn't use in the definition.

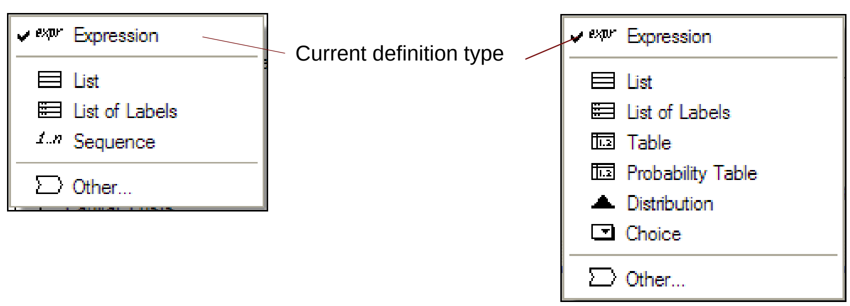
The Expression popup menu

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



expr popup menu

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



Expression Shows the definition as a mathematical expression, even if it was defined using the other expression types in this popup menu. See page 131.

List Creates an ordered set of expressions or numbers. See page 165.

List of Labels Creates an ordered set of text labels. See page 164.

Sequence Creates a list of numerical values. See page 163.

Table Creates an array of numbers or expressions. See page 164.

Probability Table Creates an array defining probabilities (numbers or expressions) across the domain of a discrete (chance) variable. See page 238.

Distribution Creates an uncertain definition by selecting a function from the Distribution System library. See page 222.

Choice Creates a popup menu for choosing one or all elements from a list. See page 121.

Other — Opens the **Object Finder** dialog, which is described in the next section. Changes the definition to the function or variable that you select from the **Object Finder**. See page 112.

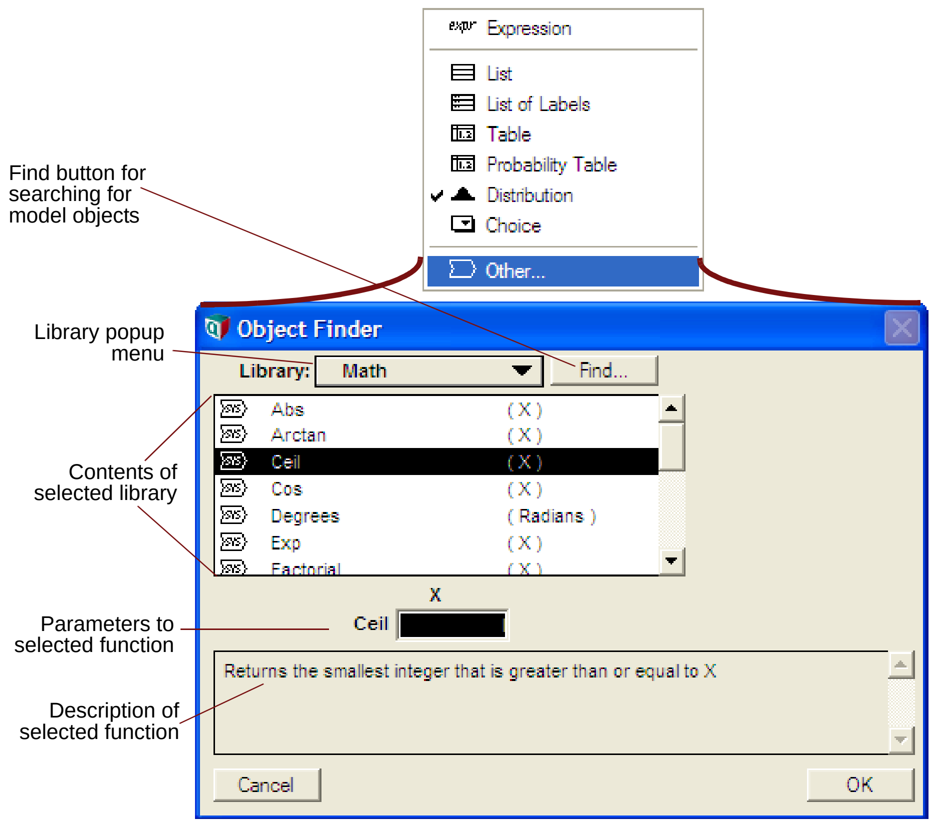
Object Finder dialog

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** dialog in two ways:

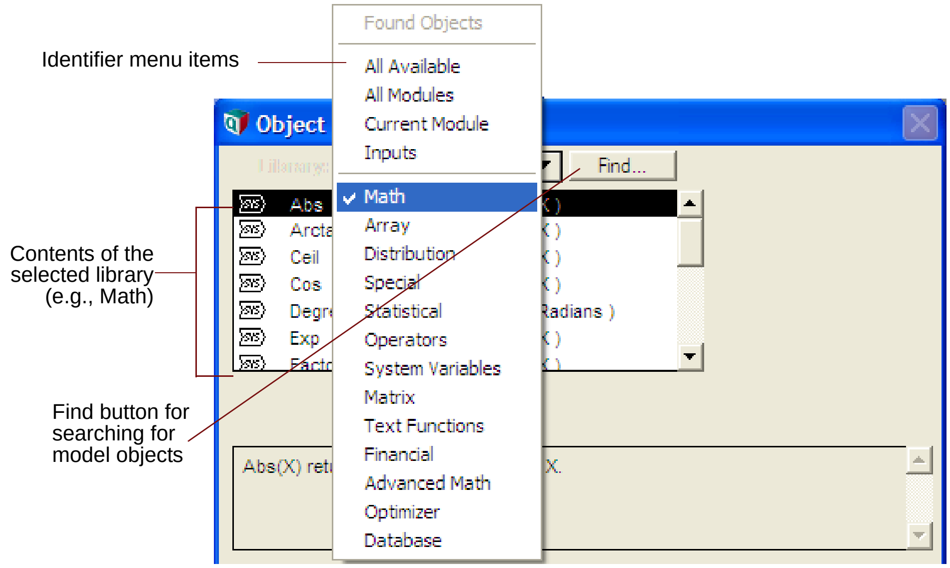
- To insert the desired function or identifier at the cursor position in the definition, select **Paste Identifier** from the **Definition** menu.

  or

- To replace the entire definition with the desired function, select **Other** from the ***expr*** menu.



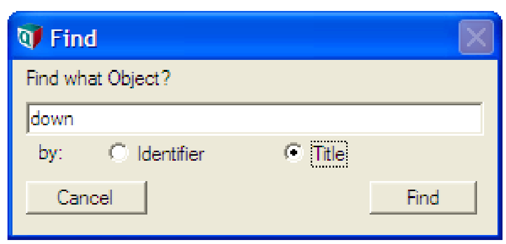
Select the desired set or library from the **Library** menu.



These are the top items in the **Library** menu:

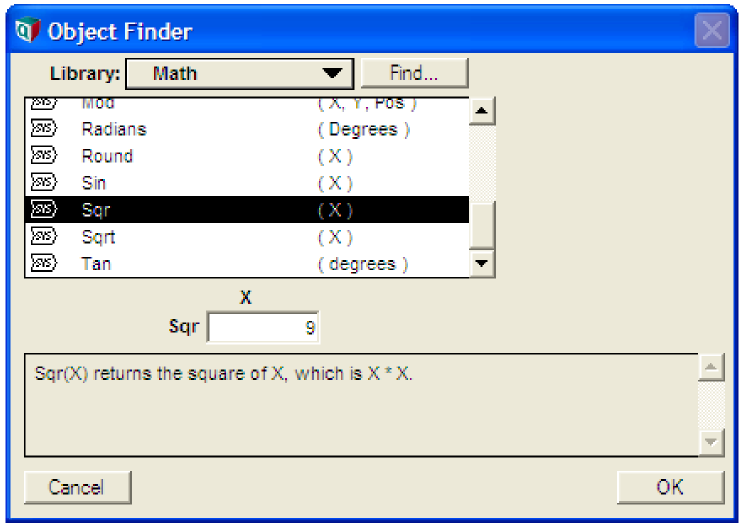
Found Objects	Objects found from **Find** button (see below)
All Available	All objects and functions, from model and built-in
All Modules	Objects from all module in the models
Current Module	Objects in the current module
Inputs	Inputs to the selected node

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

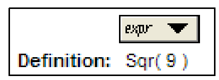


The **Found objects** library in the **Object Finder** dialog then lists all objects whose identifier or title matches in their first ***n*** characters (the ***n*** characters you type into the search box).

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



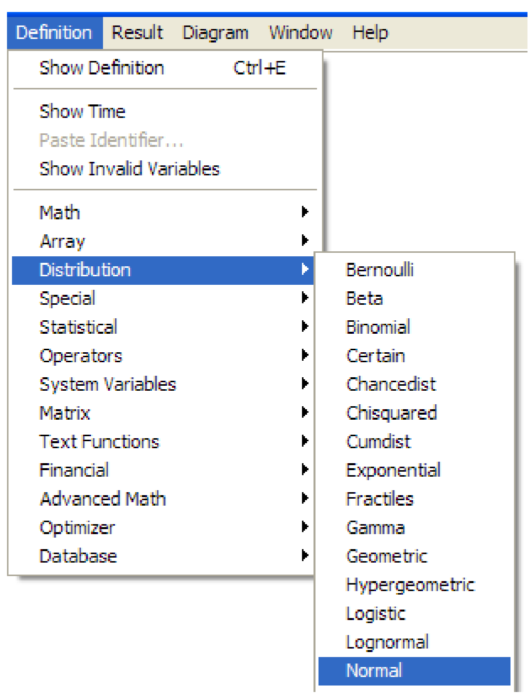
Click **OK** to place the function, identifier, or expression in the definition.



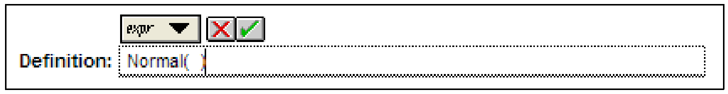
Using a function or variable from the Definition menu

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 angle brackets << >>.



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

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:

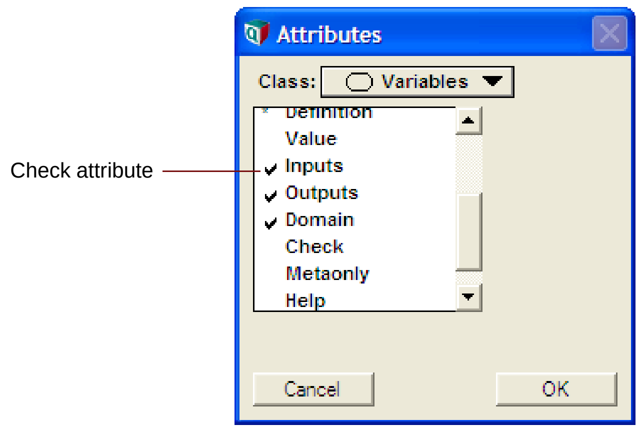
```
Check:= Percent_damage>=0 AND Percent_damage<=100
```

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 (see “Managing attributes”page 306).

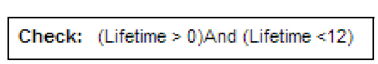


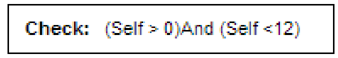
2. Scroll down the attribute list and find **Check**.
3. Click **Check** once to select it, and a second time to add a checkmark next to it. The checkmark indicates that the attribute is displayed in the **Object** window.
4. Click **OK**.

Now the **check** attribute appears 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 returns 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 to match one of the following samples.





If the **Check** expression refers to another variable, it makes a dependency from the variable being checked to the variable mentioned. It usually shows an arrow from that variable.

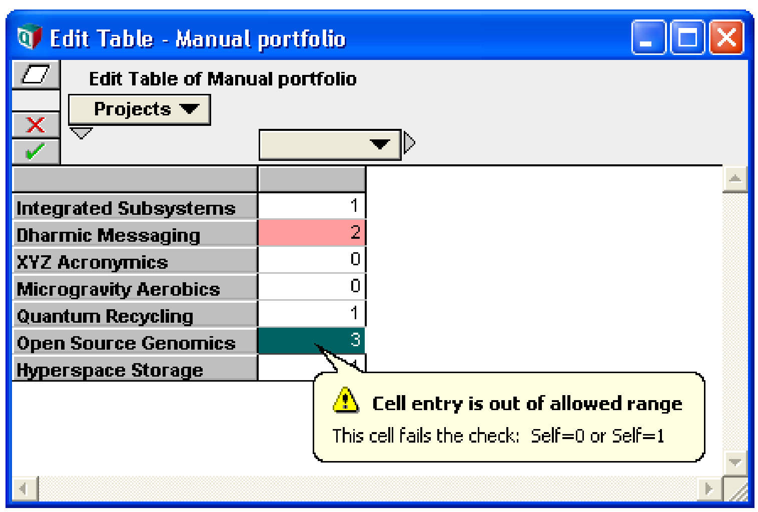
Triggering a check

If a variable **X** depends on no other variables, or if it is defined as an edit table and does not operate over the indexes of the table, it performs the check whenever you change its input value or a cell of the edit table. 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 warns if any sample value of **X** fails the check. More generally, if the value of the **Check** expression is an array, it fails if any atom in the array is false (0). If you compute first its mid value of **X** and then its prob value, it causes two evaluations, one check on the mid value and a second on the prob value.

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

Cell-by-cell validation in edit tables

When you define a check attribute for a variable defined as an edit table, Analytica will test and flag each cell individually as long as the check attribute does not operate over any of the table indexes and the values in the edit table cells do not have the potential of triggering a lengthy evaluation. Cells that fail the validation are displayed with a red background when viewing the edit table, a message balloon appears with a tail pointing to the bad cell when an out-of-range entry is first entered. If the check expression operates over a table index, this feature is disabled and the check is performed only after the final entries are stored.



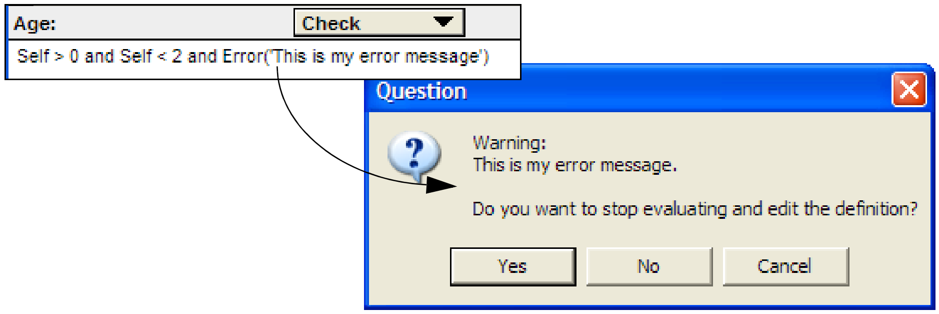
If any cell in the table contains a general expression that references other variables, then the cell-by-cell checking is disabled. This is to prevent the possibility of a delay for the user if a large part of the model must be evaluated; therefore, the cell-by-cell checking is only appropriate for tables where expressions would not be entered. If the check expression operates over any table index, such as **`Sum(Self,Projects)<5`**, then this would indicate that the check is a validation on the table as a whole, rather than on individual cells, and in this case the cell-by-cell checking is again disabled. When disabled, checks are validated at evaluation time as would occur with non-edit table variables.

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 does not perform the check again unless you change the definition of the variable or a variable it depends on.

Custom error messages

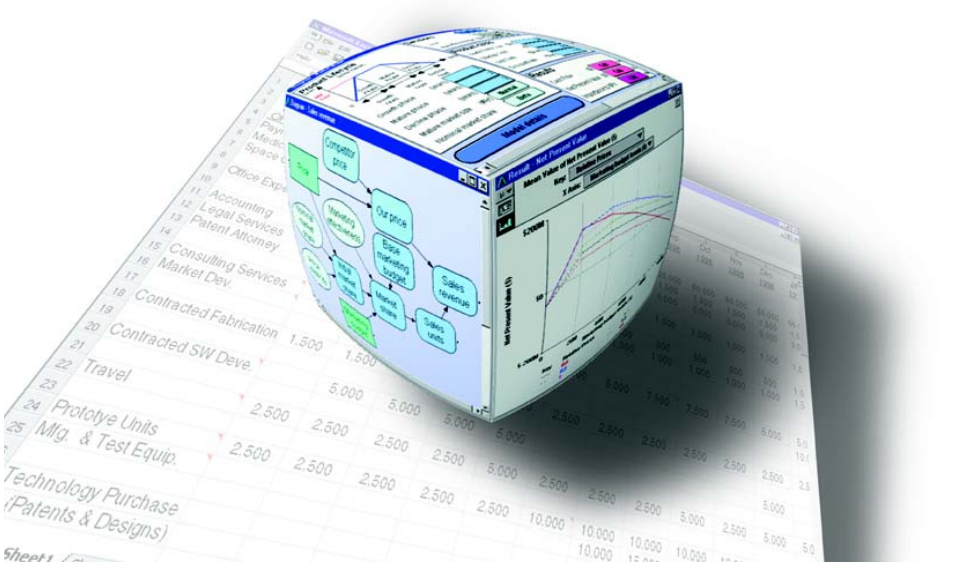
The default warning when a check fails shows the **Check** expression. This is OK for modelers, but might be obscure for end users. If you call the **Error()** function (page 348) in the check, it displays the message you pass to **Error()** instead of the default warning. Using this, you can craft a 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** (page 58) from the **Edit** menu. This checkbox is checked by default.

Chapter 9 *Creating Interfaces for End Users*

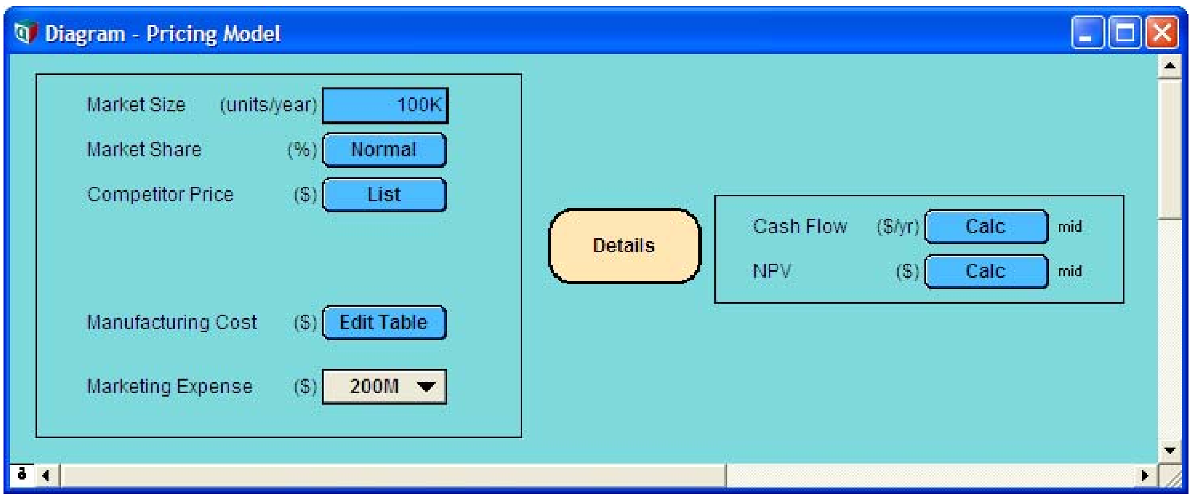
This chapter shows you how to create a user interface containing input and output nodes for easy access for other people who might use your model. It also describes how to design a clear user interface, apply icons and graphics, and include hyperlinks to web pages.



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 can 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 are 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 can be a field to enter a number or text value, a button that opens an edit table or probability distribution, or a pull-down 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. These 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

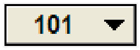
An ***input node*** lets you, or your end user, see and easily change the value of a variable directly in the diagram, without opening an *Attribute* view or **Object** window (see “Browsing with input and output nodes”page 22). In browse mode you can change only the values and definitions of input nodes.

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”page 55).

The type of definition of the original variable determines the appearance of the input node. 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 **expr menu** (page 222).

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** (page 115); 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” on page 121.

List

A list or list of labels displays as a **List** button. See “Creating an index” on page 163.

Edit table

An edit table displays as an **Edit Table** button. See “Defining a variable as an edit table” on page 169.

Probability distribution

A probability distribution displays a button with the name of the distribution. See “Defining a variable as a distribution” on page 220.

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 appears 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**.

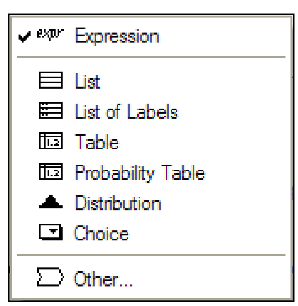
Creating a choice menu

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

Creating a menu from a list

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

1. Show the definition of the variable as a list, either in the *Attribute* view or the **Object** window.
2. Click the ***expr*** menu and select the **Choice** option. Click **OK** to the question “*Replace current definition with a Choice?*” and click **OK** again to “*Replace current definition?*” when prompted.



3. The **Object Finder** dialog displays with parameter **I=Self** and **n=0**. Click **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 (page 176).

Tip To define *Var1* as a popup menu of another variable *Var2*, that is defined as a list, select **Choice** from the ***expr*** menu, and set the first parameter to **`I=Var2`** in the **Object Finder** dialog).

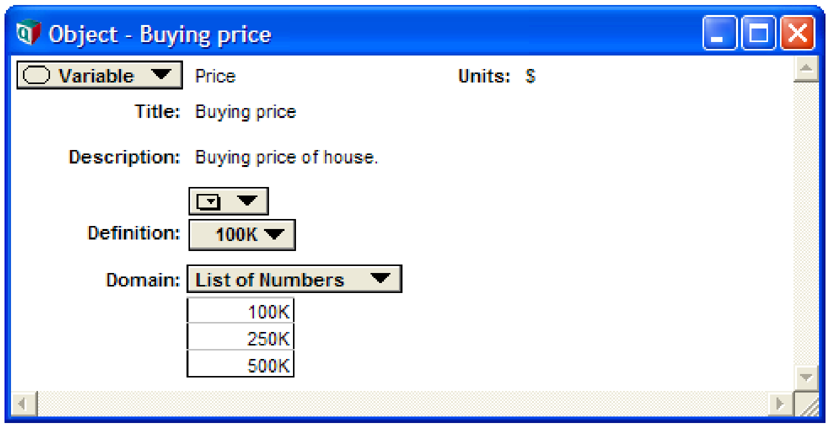
Tip To hide the **All** option on the popup, enter **`inclAll=False`** as the third parameter in the **Object Finder** dialog.

Creating a new definition

If a variable has no previous definition, when you select **Choice** from the ***expr*** 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 page 165).



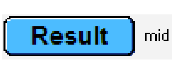
***Note:** The values in the domain are evaluated deterministically.*

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” on page 33).

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



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.

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.

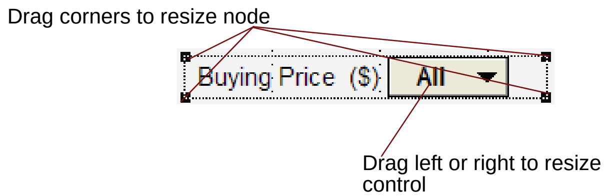
Creating an output node

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 appears 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) is the format you last set for it (see “Formatting Numbers, Tables, and Graphs” on page 81).

Resizing controls



You can resize input and output nodes by dragging their corner handles, just like other nodes. But for these, its 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 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 might want 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 page 56). 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.

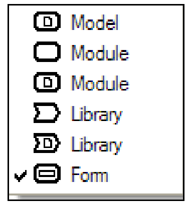
To change the appearance of an input or output node alone, use the **Set Node Style** and **Show Color Palette** options from the **Diagram** menu (see “Node Style dialog” on page 79 and “Recoloring nodes or background” on page 77). When you use these options to change the appearance of an input or output 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.

Creating 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 node** (page 54). 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 might 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 **Y** 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 might wish to organize them into additional modules (which need not be forms) to enclose related groups of inputs and outputs.

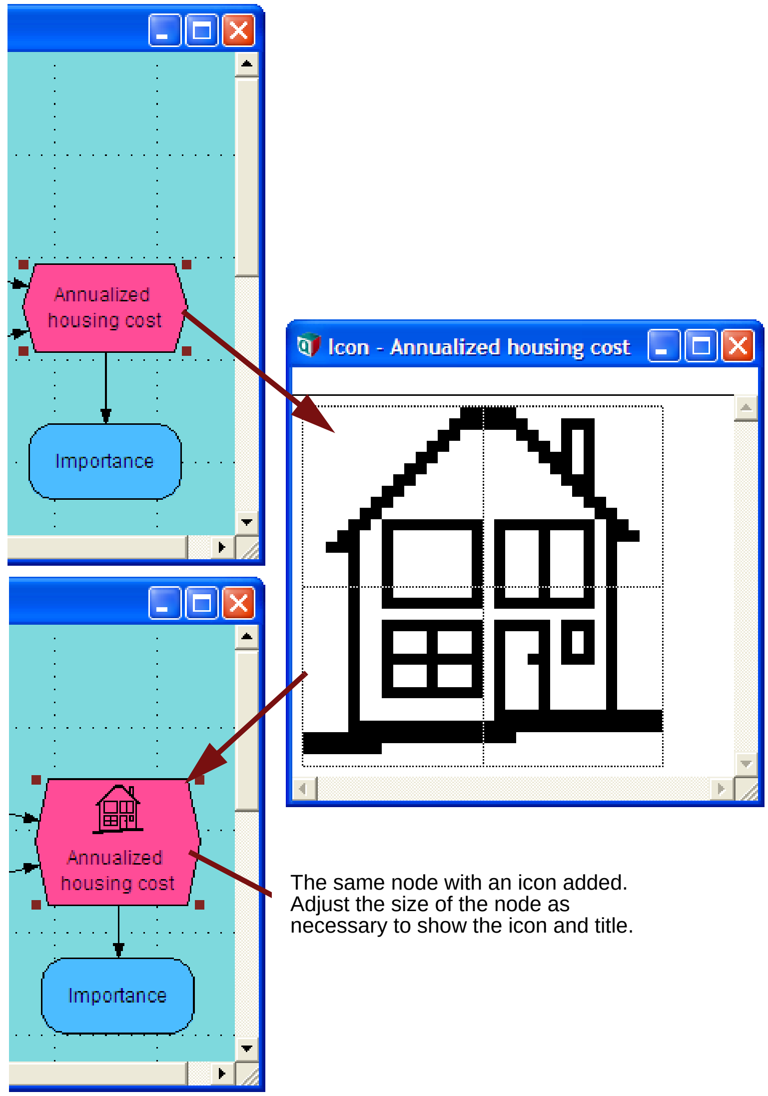
Adding icons to nodes

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 is black; if the starting pixel of the line or curve is white the line or curve is 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.

Converting image formats

Some applications post bitmap graphics on the system clipboard in compressed image formats such as PNG or JPEG. When Analytica recognizes a compressed format, it imports the image and stores it internally in that format. Unfortunately, most applications post images only as full uncompressed bitmaps. Large uncompressed bitmaps can consume a lot of space and result in very large model files; therefore, when Analytica 4.1 pastes an uncompressed bitmap, it converts it and stores it internally as compressed (lossless) PNG format. Any transparency and alpha blending present in the original image are preserved by this conversion.

Earlier releases of Analytica do not recognize these compressed bitmap formats. If someone else loads your model in Analytica release 4.0 or earlier, these images will not display. If you want your bitmap images to display when your model is loaded into Analytica 4.0 or earlier, you must convert them back into the *Legacy Bitmap* format after it has been pasted into your model. To do this:

- Make sure the edit tool is selected.
- Select the image node to convert.
- Select **Change Picture Format** from the **Diagram** menu.
- In the **Change Picture Format** dialog, select the new format to use.

These steps can be used to convert any image into any desired internal image format. In some cases, certain conversions can further reduce the amount of memory (and thus model file size) consumed by the image. Legacy Bitmap files might lose some information in the image (such as transparency and alpha blending), and might consume much more space.

Images that are stored in the ***Mac PICT*** format do not display from Analytica Web Publisher (AWP) and cannot be rendered by the Analytica Decision Engine (ADE). Images in this format might be present in older Analytica models. Using the above steps, these images should be converted to EMF if you intend to post your model on AWP or render them from ADE.

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 (page 79) 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 (page 79) 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 are "under" the framing decision node; they are visible, but you cannot 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 (page 79) 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 (page 79) to change the font or to change the background from transparent to filled.

Default and XML model file formats

Analytica supports two formats for saving models — the default format and an XML format. Both formats are fairly easy-to-read text files, which you can view and edit in standard text editors and word-processor applications. (See examples below.)

Analytica normally saves a new model in the default format. You can change to the XML format in by checking **Save in XML Format** in the **Save as** dialog when you first select **Save** from the **File** menu, or whenever you select **Save as**. It remembers and reuses the format you select in future sessions.

Sample default file format

The default format lists each object with each attribute on a separate line. The first line gives its class and identifier. Subsequent lines give each attribute name, followed by ":" followed by the attribute value. Here is part of a sample model file in the default format:

```
{ From user Richard Morgan, Model Sample_old_file_format ~~
at Jun 1, 2007 3:56 PM}
Softwareversion 4.0.0

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

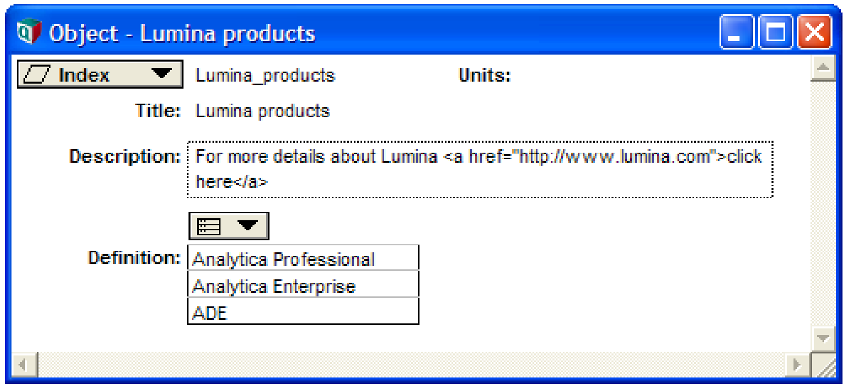
A ***description***, or other text attribute of a variable or other object, can contain a hyperlink to any web page. This is useful for linking to detailed explanations, data, or references for a model, or even to related downloadable Analytica models. In browse mode, hyperlinks appear conventionally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

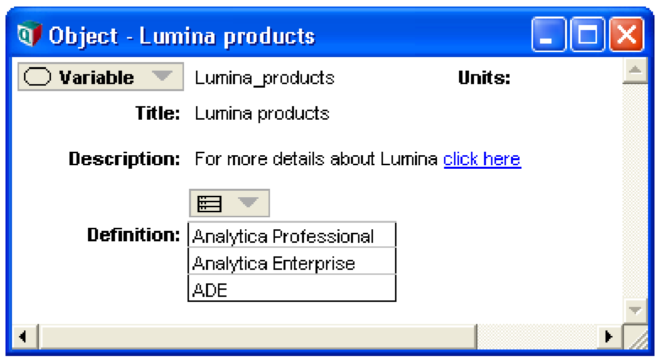
```
<a href="http://www.lumina.com">Click here</a>
```

When you switch to browse mode, the HTML code displays as a hyperlink.

In edit mode



In browse mode



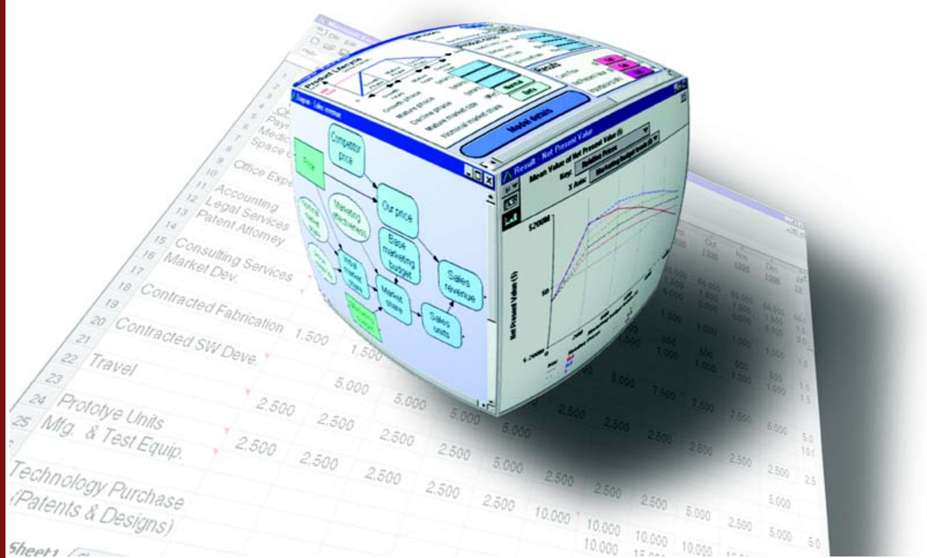
Chapter 10 *Using Expressions*

The definition of each variable is an expression, such as

```
(- B + Sqrt(B^2 - 4*A*B))/(2*A)
```

This chapter describes the elements of an expression, and their syntax, including:

- Literal values, including numbers, Boolean or truth values, and text values
- Arithmetic, comparison, and logical operators, such as + - / * ^ < > = AND OR
- IF a THEN b ELSE c
- Function calls and parameters and math functions
- Exception values INF, NAN, and NULL
- Warnings
- Datatype functions



The definition of a variable or function is an expression, such as:

```
(-B + Sqrt(B^2 - 4*A*B))/(2*A)
```

An expression can consist of or contain a literal number (including Boolean or date), a text value, an identifier of a variable, an arithmetic expression, a comparison or logical expression, **IF THEN ELSE**, or a function call, such as **Sqrt(B)**.

See Chapter 21, "Procedural Programming," for details on more advanced constructs, such as **BEGIN ... END** statements, **For** and **While** loops, local variables and assignments.

Numbers

You can enter a number into an expression using any available number formats in (see "Number formats" on page 82), including:

```
2008, 12.345, 0.00123, 5.3E20, 5.3E-20, $100,000
```

Suffix format uses a letter or symbol suffix to denote a power of ten, such as:

```
25K, 200M, 123p, 20%
```

Suffix format provides a simple, familiar way to specify large or small numbers. See "Suffix characters" (page 83) for details.

You can usually enter numbers using most number format types directly into an expression no matter what number format was specified for the variable defined by the expression. The exceptions are:

1. You may use commas to separate groups of three digits, such as **123,456.00**, only if the expression consists of that single number. If the number is part of an expression with other elements, such as **12*123,456**, you may *not* use comma separators because the syntax would be ambiguous. You should use simply **12*123456**.
2. You may use date formats, such as **10/11/2008** or **11-Oct-2008**, only for a variable specified as having the specified date number format. Otherwise, for example, it would interpret **10/11/2008** as an expression with two divide operators:

   ```
   (10/11)/2008
   ```

Integers Analytica treats integers and real numbers both as floating point numbers internally. Using the default suffix number format, it displays numbers that are very close to integers as integers.

Precision Analytica uses double-precision using 8 bytes to represent each floating point number. This means that the maximum internal precision of numbers is 15 significant digits. Some calculations, especially those that involve small differences between large numbers or large numbers of additions, might result in less precision than this maximum.

Largest and smallest numbers Analytica can represent positive numbers between about 10^{-320} and 1.797 $x10^{+308}$. If a calculation would result in a number smaller than about 10^{-320}, it rounds it down zero:

```
1/10^1000 → 0
```

If the result would be larger than 1.797 $x10^{+308}$ it returns **INF** (infinity):

```
10^1000 → INF
```

For more, see "Exception values INF, NAN, and NULL" on page 138.

Boolean or truth values

A ***Boolean*** or ***truth*** value can be **True** and **False**, or, equivalently, the number 1 or 0. For example:

```
False OR True → True
1 AND 0 → False
1 OR 0 → True
```

It actually treats every nonzero number as **`True`**. For example:

`2 AND True` → **`True`**

Boolean values are represented internally as the numbers 1 and 0. By default, a Boolean result displays as 0 or 1. To display them as **`False`** or **`True`**, change the number format of the variable to Boolean (see “Number formats” on page 82).

Text values

You specify a text value by enclosing text between single quotes, or between 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 contains only one apostrophe. 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”page 163 and “Expression view”page 164) or edit table (see “Defining a variable as an edit table”page 169). Analytica displays text values in results without the enclosing quotes. Also see “Converting number to text” on page 138.

For comparison and sort order for text, see “Alphabetic ordering of text values” on page 134.

For functions that work with text values, see “Text functions” on page 206.

For converting between numbers and text, see “Numbers and text” on page 138.

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
x + y	plus	3+2 → 5
x - y	binary minus	3-2 → 1
-x	unary minus	-2 → -2
x*y	product	3*2 → 6
x/y or x÷y	division	3/2 (= $\frac{3}{2}$) → 1.5
x^y	to the power of	3^2 = 3^2 → 9 4^.5 = $4^{\frac{1}{2}}$ → 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` → `0` `'A'<'B'` → `1`
<=	less than or equal to	`2<=2` → `1` `'ab'<='ab'` → `1`
=	equal to	`100=101` → `0` `'AB'='ab'` → `0`
>=	greater than or equal to	`100>=1` → `1` `'ab'>='cd'` → `0`
>	greater than	`1>2` → `0` `'A'>'a'` → `1`
<>	not equal to	`1<>2` → `1` `'A'<>'B'` → `1`

Alphabetic ordering of text values When applied to text values, the comparison operators, >, >=, >=, and <, use alphabetic ordering based on the numerical ASCII codes of the text values. For example:

`'Analytica' < 'Excel' → 1 (True)`

Using the numerical (ASCII) representation of the characters, means:

1. Digits precede letters:
   `'9' < 'A' → 1 (True)`
2. Uppercase letters precede lowercase letters:
   `'Analytica' > 'excel' → 0 (False)`

If you want to alphabetize without regard to case, use **`TextUppercase`** or **`TextLowerCase`** to convert letters to the same case.

`TextUpperCase('Analytica') < TextUpperCase('excel') → 1 (True)`

3. Letters with accents, umlauts, cedillas, ligatures, and other decoration come after undecorated letters, hence alphabetic ordering might be different from what you expect.

Sortindex(d, i) sorts text values in ***d*** using this ASCII ordering scheme. But, **Rank()** works only on numbers, not text values.

Logical operators Logical operators apply to Boolean values and produce Boolean values.

Operator	Meaning	Examples	
`b1` **`AND`** ***`b2`***	true if both ***b1*** and ***b2*** are true, otherwise false	`5>0 AND 5>10`	→ `False`
`b1` **`OR`** ***`b2`***	true if ***b1*** or ***b2*** or both are true, otherwise false	`5>0 OR 5>10`	→ `True`
`NOT` ***`b`***	true if ***b*** is false, otherwise false	`NOT (5>0)`	→ `False`

Scoping operator (::) It is possible that a model created in a previous release might contain a variable or function with the same identifier as a new built-in variable or function. In this situation, an identifier name appearing in an expression might 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 refers to the user's variable or function. With this convention, existing models are not changed by the introduction of new built-in functions.

Example Suppose a model from an older release of Analytica contains the user-defined function **`Irr(Values, I)`**. Then:

`Irr(Payments, Time)`	User's **`Irr`** function
`::Irr(Payments, Time)`	The built-in function

Operator binding precedence

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

1. parentheses ()
2. function calls
3. Not
4. @I, \A, \[I]A, #R.
5. A.I
6. A[I=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 X := 1M
Variable Y := 1
IF X > Y THEN X ELSE Y → 1M
```

returns the larger of **`X`** and **`Y`**.

It is possible to omit the **`ELSE`** clause:

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

If the condition is false, it gives 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” on page 161.

Function calls and parameters

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” on page 315).

Position-based function calls

The conventional position-based syntax to call a function uses this form:

```
FunctionName(param1, param2, ...)
```

You follow the function name by a comma-separated list of parameters enclosed between parentheses, with the parameters in the specified sequence. 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
Mod(7, 3) → 1
Pmt(8%, 30, -1000) → $88.83
Normal(500, 100)
```

Some functions have optional parameters. In that case, you can simply omit the trailing parameters that will use their default values.

Name-based function calls

Analytica also offers name-based parameter syntax as an alternative for calling most functions: You name each parameter, followed by a colon (:) and the value passed to that parameter. Since the parameters are named, you can list them in any order. For example, this function has four parameters, of which you can provide any two to define the distribution:

```
Lognormal(median, gsdev, mean, stddev)
```

Calling it using name-based syntax:

```
Lognormal(mean: 10, stddev: 1.5)
```

is equivalent to the following using position-based syntax, which uses commas to indicate that the first two parameters are omitted:

```
Lognormal( , , 10, 1.5)
```

because **mean** and **stddev** (standard deviation) are the third and fourth parameters. Name-based syntax is useful for functions with many optional parameters. It’s usually easier to read name-based function calls because you don’t need to remember the ordering of the parameters.

Math functions

These functions can be accessed from the **Math** library from the **Definition** menu.

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**, **e** raised to the power of **x**.

```
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**.

```
Sqrt(25) → 5
Sqrt(-1) → NAN
```

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, it rounds **x** 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 (the inverse of Tan).

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

See also “Arccos(x), Arcsin(x), Arctan2(y, x)” on page 209.

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 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 two 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 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() (page 348) 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**.

An alternative method, for converting text to a number is to use the **Coerce Number** qualifier on a user-defined function (see "Parameter qualifiers" on page 318). For example, you could define a user-defined function such as:

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

Exception values INF, NAN, and NULL

INF, **NAN**, and **Null** are system constants that arise in exceptional cases.

Constant	Meaning
INF	Infinity or a real number larger than can be represented, e.g., **1/0**
NAN	Not a Number: Actually, the result is known to be "number" but not well defined, e.g., **0/0**
Null	The result of an operation where the desired data is not there, such as **X[I = '?']**, where index **I** does not have the value **'?'**

INF (infinity)

INF is the result of a numerical calculation whose absolute value is larger than largest number Analytica can represent. This could be an overflow — that is a valid real number greater than 1.797 $x10^{+308}$:

```
10^1000 → INF
```

or it could be a division by zero or other result that is mathematically infinite:

```
1/0 → INF
```

INF can be positive or negative:

```
-1 * 10^1000 → -INF
```

You can use **INF** as a value in an expression. You can perform useful, mathematically correct arithmetic with **INF**, such as:

```
INF + 10 → INF
INF/0 → INF
```

```
10 - INF → -INF
100/0 = INF → True
```

NAN **`NAN`** is the result of a numerical calculation that is an undetermined or imaginary number, including numerical functions whose parameter is outside their domain:

```
INF - INF → NAN
0/0 → NAN
INF/INF → NAN
Sqrt(-1)→ NAN
ArcSin(2)→ NAN
```

It usually gives a warning if you apply a function to a parameter value outside its range, such as the two examples above — unless you have pressed “Ignore warnings” (see “Warnings” on page 139).

Any arithmetic operation, comparison, or function applied to **`NAN`** returns **`NAN`**:

```
0/0 <> NAN → NAN
```

Analytica’s representation and treatment of **`NAN`** is consistent with IEEE Floating point standards. **`NAN`** stands for “Not A Number,” which is a bit misleading, since **`NAN`** really is a kind of number.

You can detect **`NAN`** in an expression using the **IsNaN()** function (page 140).

Null **`Null`** is a result that is ill-defined, usually indicating that there is nothing at the location requested, for example a subscript using a value that does not match a value of the index:

```
Index I := 1..5
X[I=6] → Null
```

Other operations and functions that can return **`Null`** include **Slice()**, **Subscript()**, **Subindex()**, and **MDTable()**.

You can test for **`Null`** using the standard = or <> operators, such as:

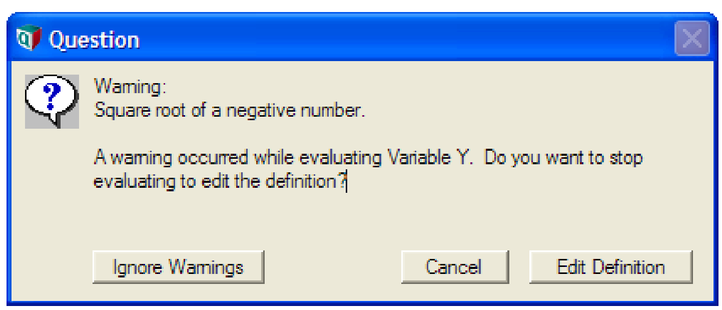
```
X[I=6] = Null → True
```

or you can use **IsUndef(`X[I=6]`)**.

Warnings

Warnings can occur during evaluation, for example when trying to take the square root of a negative number, 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`** values can be propagated further into a model.

Tip If you click the **Ignore warnings** button, it will ignore all warnings from this variable and all other variables in this and future sessions with this model. Ignoring warnings could lead to you getting **`NAN`** or **`NULL`** results for unknown reasons. If this happens, you can switch warnings back on by checking **Show result warnings** in the **Preferences** dialog.

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 **`NAN`** resulting from evaluating **`Sqrt(X)`** for negative **`X`** 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 evaluates **`Sqrt(X)`**, and generates **`NAN`** as for **`Y`** above. But, the conditional means that resulting value for **`Z`** contains no **`NAN`s**, 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 common warning "subscript or slice value out of range" returns **`Null`**, for example:

```
Index I := 1..5
X[I=6] →  Null
```

If you want to ignore warnings for a single variable, you can use the **IgnoreWarnings()** function around the definition.

Datatype functions

A value can be a number, text, **`Null`**, or a reference (see "References and data structures" on page 340 for more on references). Integers, reals, Boolean, and date values, are all represented as numbers. You can use these functions from the **Special** library of **Definition** menu to determine the type.

IsNumber(x) Returns **`True`** if **x** is a number, including a Boolean, date, **`INF`** or **`NAN`**.

```
IsNumber(0) → True
IsNumber(False) → 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('hello') → True
IsText(7) → False
IsText('7') → True
```

IsNaN(x) Returns **`True`** if **x** is "not a number," i.e., **`NAN`**. **`INF`** or regular numbers 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 **`Null`**, otherwise it returns **`False`**.

IsReference(x) Returns **`True`** if **`x`** is a reference to a value.

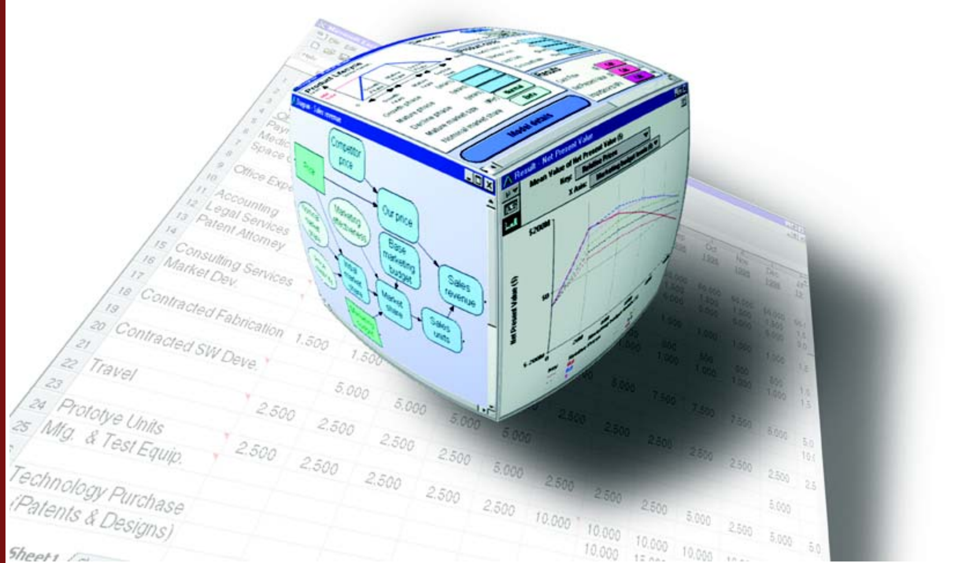
TypeOf(x) Returns the type of expression **`x`** as a text value, usually one of **`"Number"`**, **`"Text"`**, **`"Reference"`**, or **`"Null"`**. **`INF`** and **`NAN`** are both of type **`"Number"`**:

```
TypeOf(2008) → "Number"
TypeOf('2008') → "Text"
TypeOf(INF) → "Number"
TypeOf(0/0) → "Number"
```

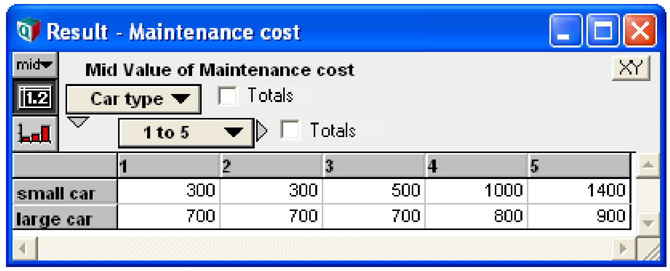
Chapter 11 *Arrays and Indexes*

Analytica offers powerful features for working with indexes and arrays, with one, two, or many dimensions. Collectively, we refer to them as Intelligent Arrays™. This chapter provides an extended introduction to the essential concepts, followed by more details on:

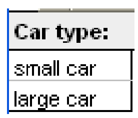
For more, see Chapter 12, “More Array Functions.”

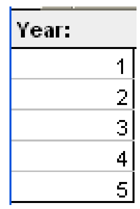


Arrays The value of a variable can be a single number, Boolean, text value, or reference — more generally, an ***atom*** — or it can be an ***array***, a collection of such values, viewable as a table with one or more dimensions. Here's an array with two dimensions.



Indexes The dimensions of the variable **`Maintenance_cost`** are identified by the indexes **`Car_type`** and by **`Year`**.



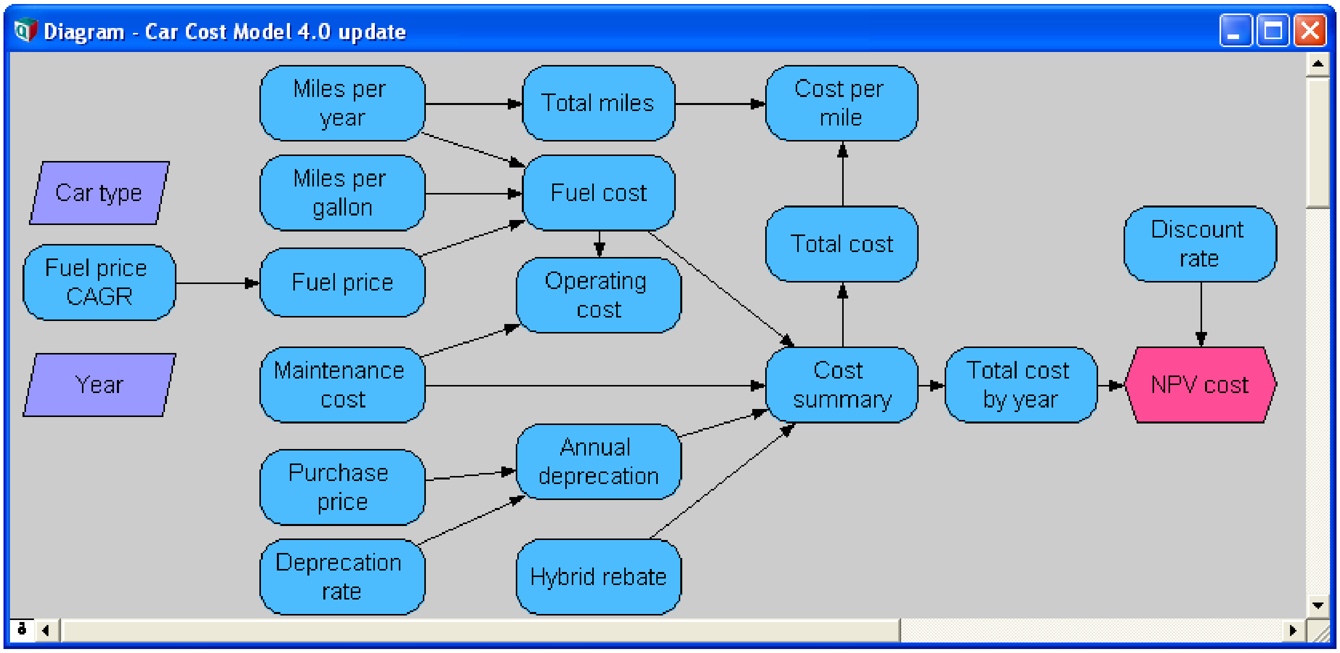


Intelligent Arrays Each index is a separate variable and can be used as a dimension of many arrays. For example, other arrays can be indexed by **Car type** or **Year**. The fact that Analytica identifies each dimension by a named index provides the basis for the ease and flexibility with which you can create, calculate with, and display arrays with one or many dimensions. It lets expressions and functions work with arrays just the same way they work with single numbers. They automatically generalize to work with arrays without you having to bother with subscripts and **For** loops the way you would with other computer languages. We call this set of features ***Intelligent Arrays***™.

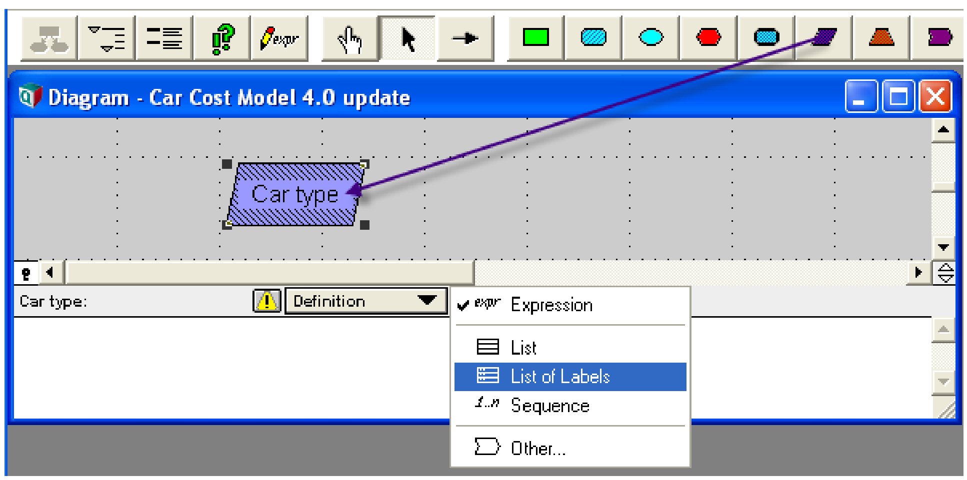
Learning key concepts There are some subtleties to the effective use of Analytica's Intelligent Arrays. To fully appreciate them, you might find you need to let go of some of your past experience with spreadsheets or programming languages. But, once you grasp the key ideas, they will seem quite simple and natural. Many Analytica users end up thinking that these features are what make Analytica most valuable. We recommend that you start by reading through the "Introducing indexes and arrays" below, which illustrates key concepts and features. You can then refer to the rest of this chapter and the next chapter, "More Array Functions" on page 181, as needed for details.

Introducing indexes and arrays

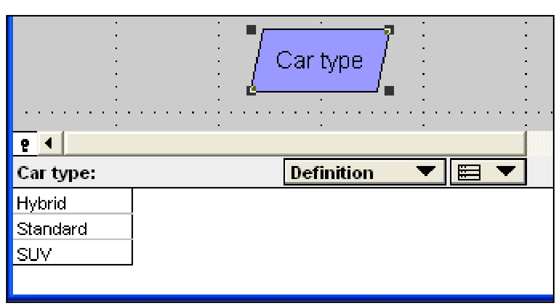
In this section, we demonstrate the concepts and features of indexes and arrays by building a model to compare the costs of three automobiles, including fuel costs, maintenance, depreciation, and a rebate for a hybrid car. We will end up with a model that looks like this.



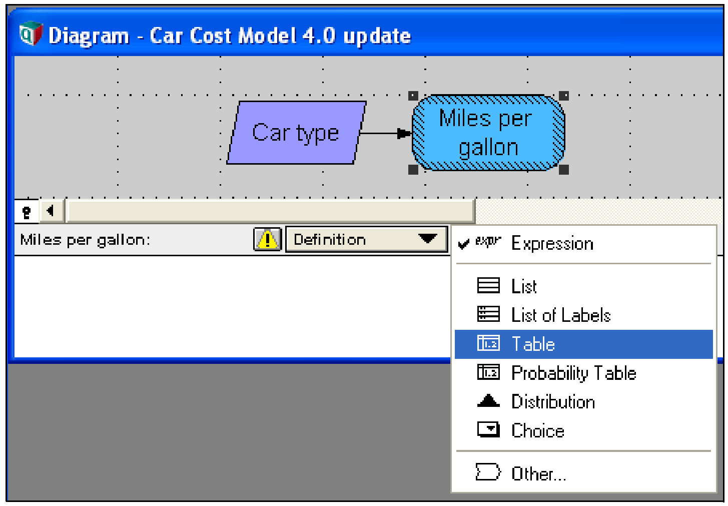
Create an index Suppose you want to compare the fuel cost of three different vehicles, each with different fuel efficiency. First let's define an index `Car_type`, listing the three different types of cars as text values. You create a new index by dragging the index node from the node menu. Type the title `Car type` into the node. In its definition attribute, select **List of Labels** from the ***expr*** menu.



Type the car types **Standard**, **Hybrid**, and **SUV** (Sports Utility Vehicle) into individual cells of the index. Press *Enter* to add the next cell.

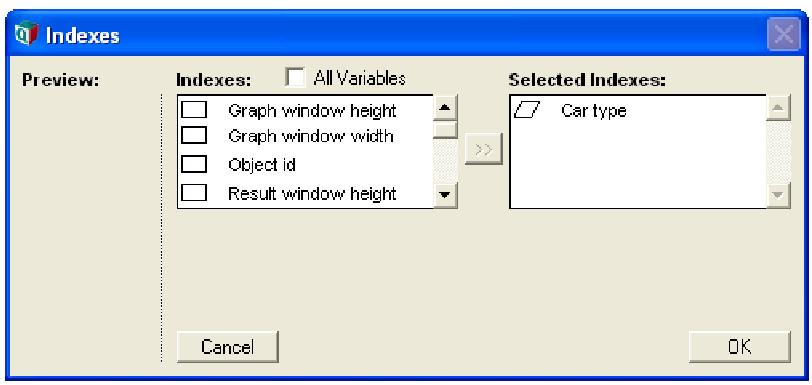


Create an edit table Now we create a new variable by dragging it from the node menu, typing its title **`Miles per gallon`**[1] into the node, and drawing an arrow to it from the index **`Car_type`**.

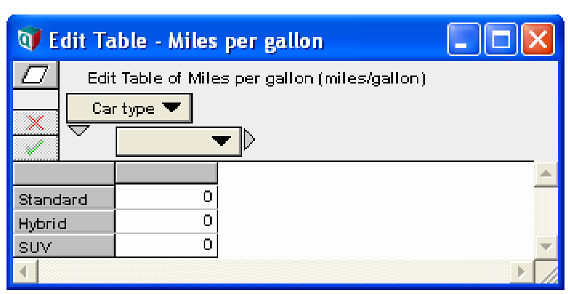


Tip By default, diagrams do not display arrows to or from index nodes after you have drawn them. For clarity, we display them by checking **Show arrows to/from Indexes** in the **Set Diagram style** dialog from the **Diagram** menu. See “Diagram Style dialog” on page 78.

In the attribute panel above, we show the definition of **`Miles_per_gallon`**, and select **Table** from the ***expr*** menu. This opens the **Indexes** dialog to let you choose which index(es) to use for the table dimensions.

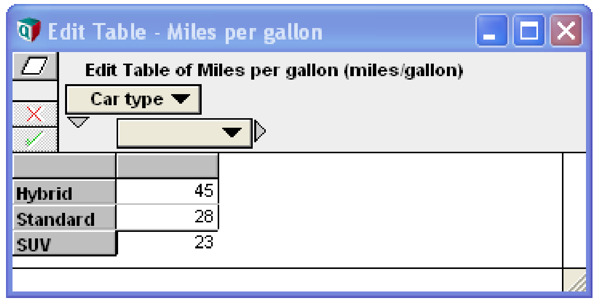


It starts with **`Car_type`** as the selected index because you drew the arrow from it (see “Indexes dialog” on page 170). Click **OK** to accept. An edit table appears, indexed by **`Car_type`**, with cells initialized to **`0`**.



You can now edit the cells of the table. Type in a number for each **`Car_type`**.

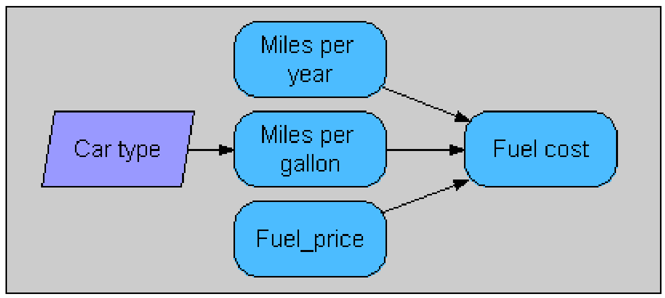
1. We apologize to our readers outside the U.S. for using the archaic units, gallons and miles!



This completes the edit table for **Miles per gallon**.

Combine a scalar (0D) and 1D array

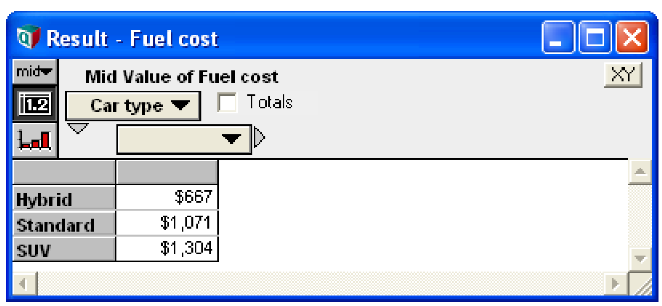
Now let’s calculate the annual fuel cost for each car type. We create three new variables, **`Miles_per_year`**, **`Fuel_price`**, and **`Fuel_cost`** and draw the arrows.



Type these definitions for the new variables:

```
Miles_per_year := 10K
Fuel_price:= 3.00
Fuel_cost := Fuel_price * Miles_per_year / Miles_per_gallon
```

Select **`Fuel_cost`** and click the **Result** button to show this result table.

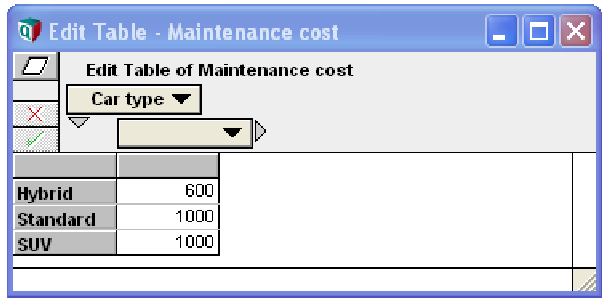


Array abstraction with arithmetic operators

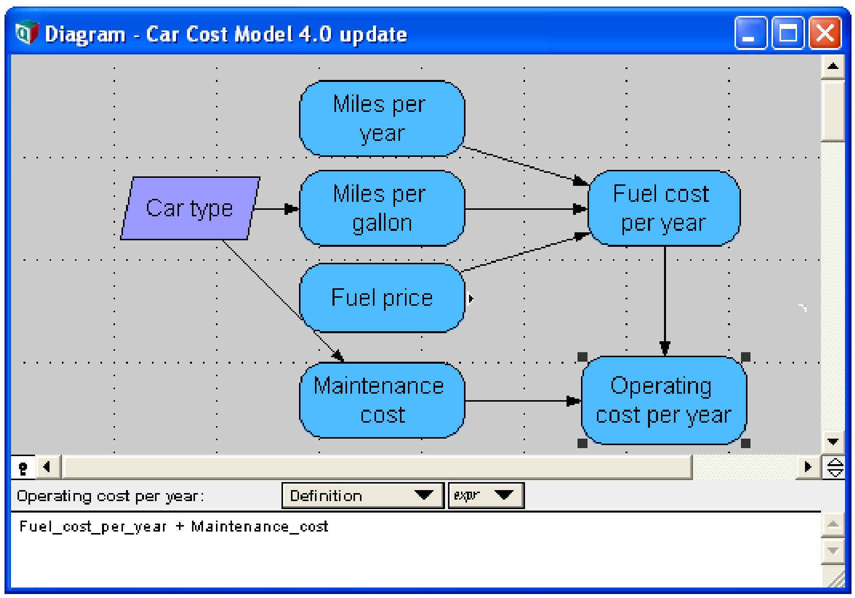
This table for **`Fuel_cost`** was computed using **`Miles_per_gallon`** for each **`Car_type`**, and the single (scalar) numbers, **`3.00`** for **`Fuel_price`** and **`10K`** for **`Miles_per_year`**. The arithmetic operations * and / work equally well when one or both operands is an array as when it is a single number – also known as an ***atom*** or ***scalar*** value. The same is true for +, -, and ^. This is an example of ***array abstraction***, central to Intelligent Arrays™.

Define another edit table

Now let’s add in the maintenance costs. We create a new variable **`Maintenance_cost`**, defined as an edit table, based on the **`Car_type`** index, just as we did for **`Miles_per_gallon`**.

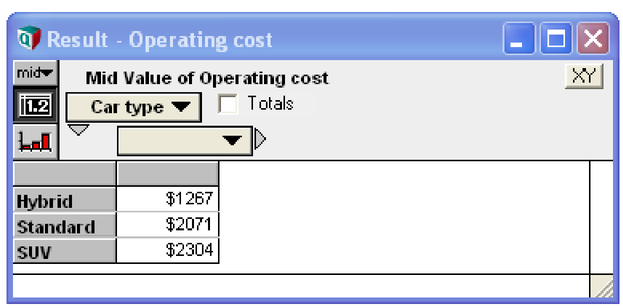


We now create **`Operating_cost`** as the sum of **`Fuel_cost`** and **`Maintenance_cost`**. Here is the diagram showing the definition of the new variable.



Operation on two 1D tables with the same index

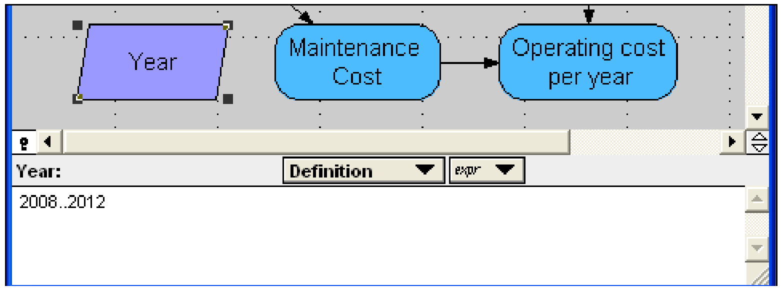
Here is the result.



It is the sum of **`Fuel_cost`** and **`Maintenance_cost`**, both 1D arrays indexed by **`Car_type`**, so the result is also indexed by **`Car_type`**. Each cell of the result is the sum of the corresponding cells of the two input variables.

Make an index as a sequence of numbers

Now let's add another index, **`Year`**, so that we can extend the model to compute the costs for multiple years. We create the new index as before. In its definition we enter **`2008..2012`**, to specify the start and end year.



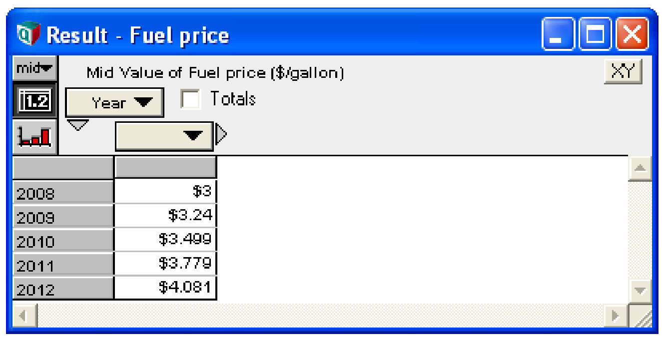
The value of **`Year`** is now the sequence of years from **`2008`** to **`2012`**. (See “Creating an index” on page 163 and “Functions that create indexes” on page 166 for other ways to define indexes.)

Compound annual growth of fuel price by year

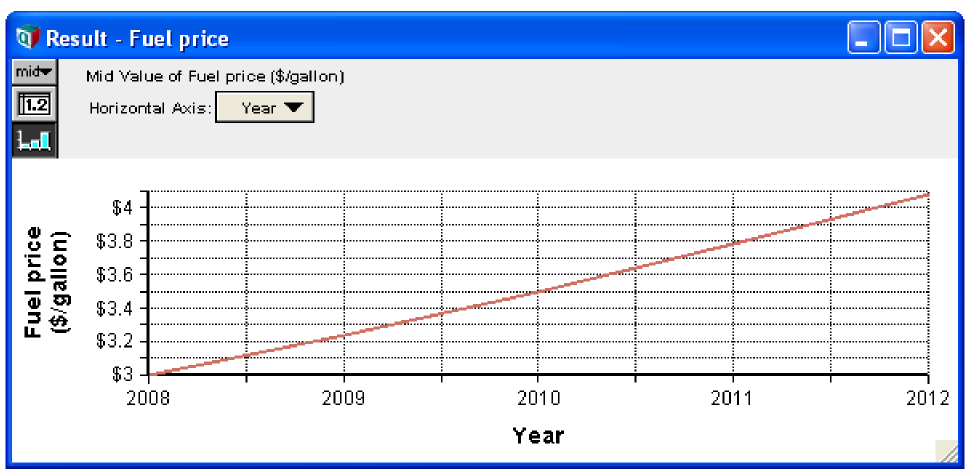
What happens if **`Fuel_price`** changes over time? Let’s model **`Fuel_price`** starting with its current value of **`3.00`** ($/gallon) multiplied by a compound annual growth rate of **`10%`** per year:

```
Fuel_price := 3.00 * (1 + 10%)^(Year – 2008)
```

This expression says that **`Fuel_Price`** starts at **`3.0`** in **`Year 2008`** (when the exponent **`(Year – 2008)`** is zero. For each subsequent year, we raise **`(1 + 10%)`** to the power of the number of years from the start year, **`2008`** — i.e., standard compound growth. Here’s the result.

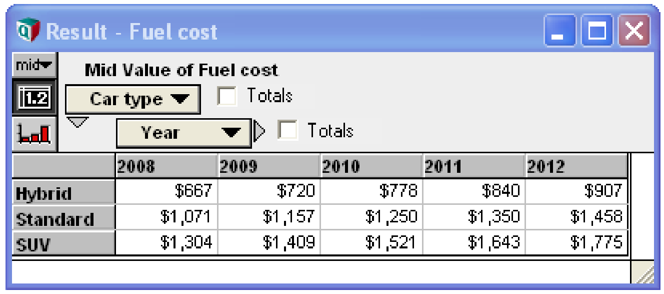


Click the graph icon  to view this as a graph.



Combine two 1D arrays with different indexes

Now look at **`Fuel_cost`**. Its has three inputs, **`Miles_per_year`**, which is still a single number, **`10K`**, **`Miles_per_gallon`**, which is indexed by **`Car_type`**, and **`Fuel_price`**, which is now indexed by **`Year`**. The result is a two-dimensional table indexed by both **`Car_type`** and **`Year`**. It contains every combination of **`Miles_per_gallon`** by **`Car_type`** and **`Fuel_price`** by **`Year`**.



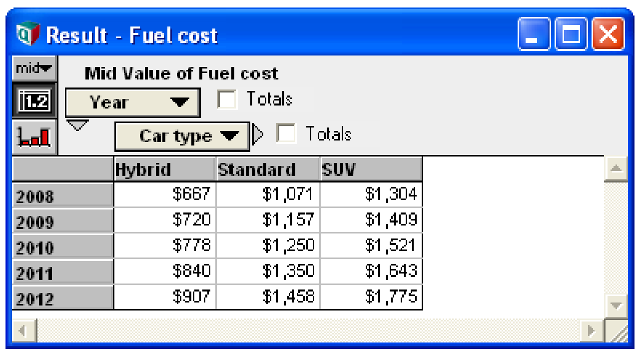
	2008	2009	2010	2011	2012
Hybrid	$667	$720	$778	$840	$907
Standard	$1,071	$1,157	$1,250	$1,350	$1,458
SUV	$1,304	$1,409	$1,521	$1,643	$1,775

Result of operation contains all indexes of operands

This illustrates a general rule for Intelligent Arrays, that the result of an operation contains the union of the sets of indexes of its operands.

Pivot a table, exchanging rows and columns

In the table above, it shows **`Car_type`** down the rows and **`Year`** across the columns. To pivot the table — i.e., exchange rows and columns — select the other index from the menu defining the columns (or the rows).



	Hybrid	Standard	SUV
2008	$667	$1,071	$1,304
2009	$720	$1,157	$1,409
2010	$778	$1,250	$1,521
2011	$840	$1,350	$1,643
2012	$907	$1,458	$1,775

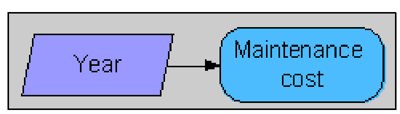
(We expanded the window size so that all rows are visible.)

Rows and columns are just for display of tables

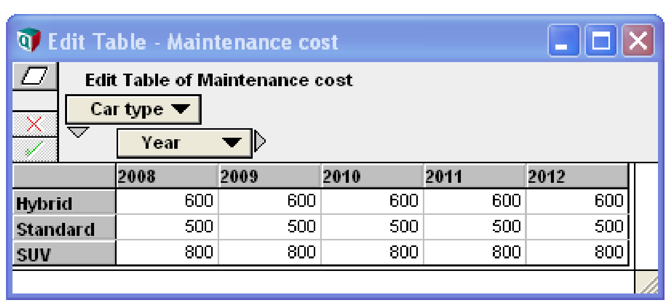
Unlike other computer languages, with Analytica, you don't need to worry about the ordering of the indexes in the table. Rows and columns are simply a question of how you choose to display the table. They are not intrinsic to the internal representation of an array.

Add a dimension to an edit table

Maintenance costs also changes over time, so we need to add **`Year`** as dimension. Simply draw an arrow from **`Year`** to **`Maintenance_cost`**.

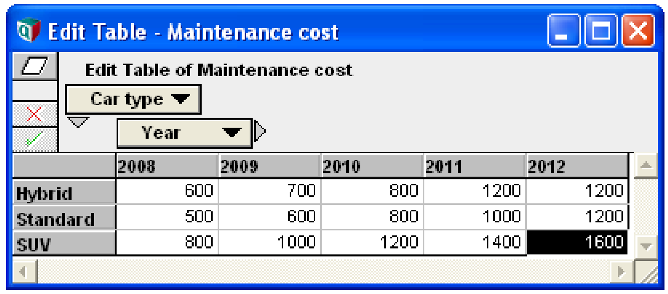


When it prompts *"Do you wish to add Year as a new index of the table in Maintenance_cost?"* click **Yes**. Now open the edit table for **`Maintenance_cost`**. It has added **`Year`** as a second dimension, copying the number for each **`Car_type`** across the years.



	2008	2009	2010	2011	2012
Hybrid	600	600	600	600	600
Standard	500	500	500	500	500
SUV	800	800	800	800	800

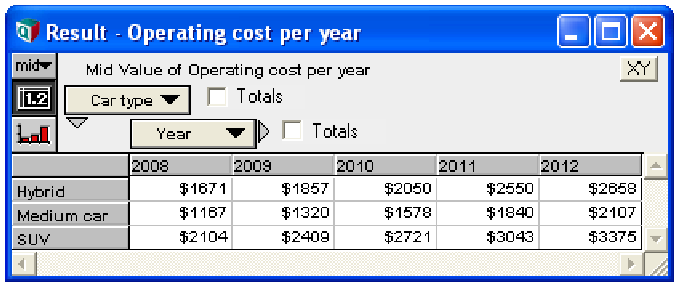
Notice that it shows the same values for each **`Year`**, following the rule that a value is constant over a (previously) unused index. Now you can edit these numbers to reflect how maintenance cost increases over time.



	2008	2009	2010	2011	2012
Hybrid	600	700	800	1200	1200
Standard	500	600	800	1000	1200
SUV	800	1000	1200	1400	1600

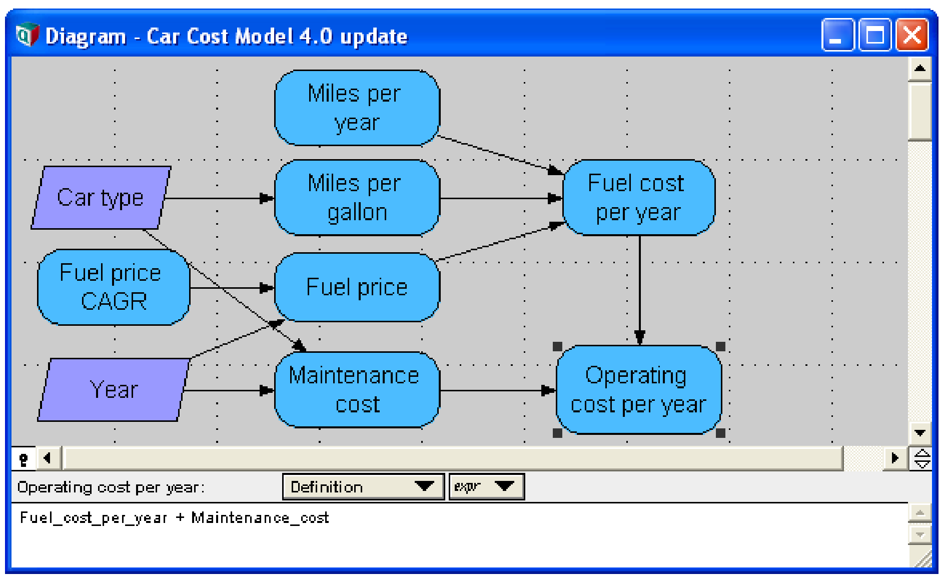
Combine two 2D arrays with the same indexes

Let's look at the value of **`Operating_cost`** again.



	2008	2009	2010	2011	2012
Hybrid	$1671	$1857	$2050	$2550	$2658
Medium car	$1167	$1320	$1578	$1840	$2107
SUV	$2104	$2409	$2721	$3043	$3375

Since its inputs, **`Fuel_cost`** and **`Maintenance_cost`**, are both indexed by **`Car_type`** and **`Year`**, the result is also indexed by those two indexes. Each cell contains the sum of the corresponding cells from the two input variables. The diagram now looks like this.



A list of numbers for parametric sensitivity analysis

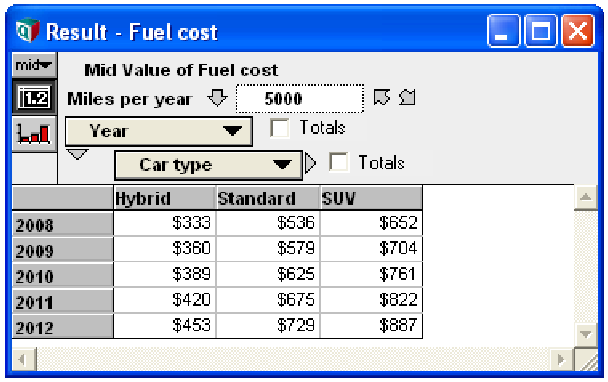
Suppose you're not sure how many miles you drive per year. You want to examine three scenarios. You include three values in **`Miles_per_year`** by specifying a list of numbers enclosed in square brackets:

```
Miles_per_year := [5K, 10K, 15K]
```

Even though **`Miles_per_year`** is not defined as an index node, it becomes an implicit index. This is an example of model behavior analysis, described in "Varying input parameters" on page 42.

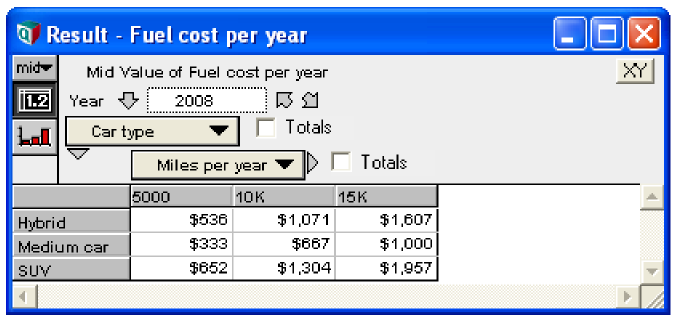
Combine three 1D arrays with different indexes

Now all three inputs to **`Fuel_cost`** are one-dimensional arrays, each with a different index. Its result is a three-dimensional table, computed for each combination of three input variables, so indexed by **`Miles_per_year`**, as well as **`Year`** and **`Car_type`**.

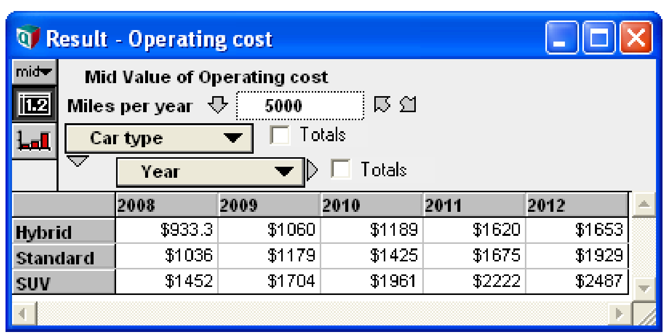


The new third index, **`Miles_per_year`**, appears as a slicer index, initially showing the slice for **`5000`** miles/year. You can click the *down-arrow* for a menu to choose another value, or click the diagonal arrows or to step through the values for miles/year. See "Index selection" on page 30.

Pivot a 3D table You can also pivot a table to display, for example, the **`Car_type`** down the rows and **`Miles_per_year`** across the columns, for a selected **`Year`** in the slicer.



Combine a 2D and 3D array with two common indexes When we look at **`Operating_cost`** again, it also now has three dimensions. Again the result has the union of the indexes of its operands.



It is the sum of fuel cost and maintenance cost, each of which is indexed by **`Car_type`** and **`Year`** as before, but now **`Fuel_cost`** has the third index, **`Miles_per_year`**. The result contains all three dimensions.

Propagation of indexes without changing downstream definitions Note how each time we add an index to an input variable, or change a variable, e.g., **`Miles_per_year`**, to be a list of values, the new dimensions automatically propagate through the downstream variables. The results have the desired dimensions (the union of the input dimensions) without any need to modify their definitions to mention those indexes explicitly.

Sum over an index If we want to sum over **`Year`** to get the total cost, we define a new variable:

```
Variable Total_operating_cost := Sum(Operating_cost, Year)
```

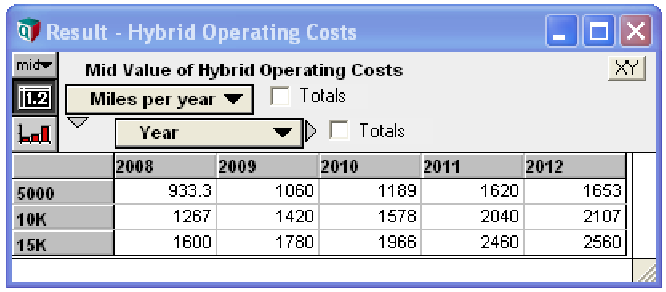
We mention the index **`Year`**, over which we want to calculate the sum. But, we do not need to mention any of the other indexes of the parameter **`Operating_cost_by_year`**.

The built-in function **Sum(x, i)** is called an ***array-reducing function***, because it reduces its parameter **x** by one dimension, namely **i**. There are a variety of other reducing functions, including **Max(x, i)**, **Min(x, i)**, and **Product(x, i)** (see “Array-reducing functions” on page 185). These functions explicitly specify the index over which they operate. Since they mention it by name, you don’t need to know or worry about any ordering of dimension in the array.

X[i = v]: subscript

The subscript construct lets you extract a slice or subarray from an array, say the values for the **`Hybrid Car_type`**:

```
Total_operating_cost[Car_type = 'Hybrid'] →
```



You can also select multiple subscripts in one expression:

```
Fuel_cost[year = 2012, Car_type = 'SUV', Miles_per_year = 10K] →
3375
```

For more, see “x[i=v]: Subscript construct” on page 174.

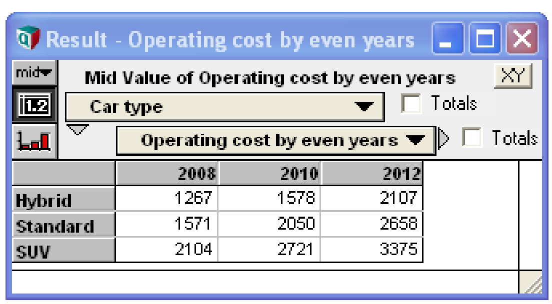
Name-based subscripting

You can list the indexes in any order since you identify them by name. Again you don’t need to remember which dimension is which. This is called ***name-based subscripting syntax***, in contrast to the more conventional sequence-based subscripting. In addition to absolving you from having to remember the ordering, name-based subscripting generalizes flexibly as you add or remove dimensions of the model.

When the subscripting value v is an array

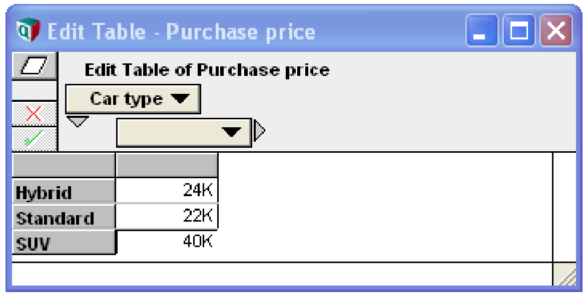
The value **v** in **x[i=v]** can itself be an array. For example, if you wanted to get the operating cost only for even years:

```
Operating_cost[Miles_per_year = 10K, Year = [2008, 2010, 2012]]
```



Purchase price and depreciation

To complete the model, let’s add the **`Purchase_price`**, an edit table indexed by **`Car_type`** (just as we created **`Miles_per_gallon`**).

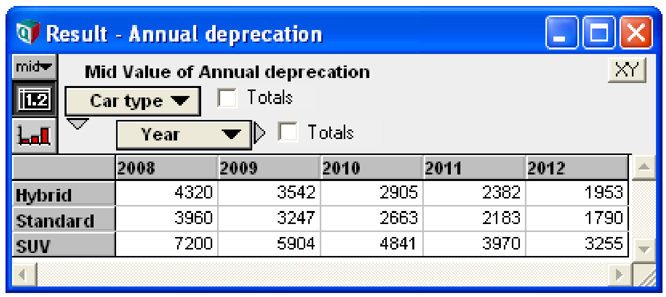


Hybrid	24K
Standard	22K
SUV	40K

To annualize this, we compute the annual depreciation, using a depreciation rate of 18% per year — typical for an automobile:

```
Variable Depreciation_rate := 18%
Variable Annual_depreciation := Purchase_price * Depreciation_rate *
                          (1 - Depreciation_rate) ^ (Year - 2008)
```

It calculates this formula for each **Year**, raising **(1 - Depreciation_rate)** to the power of the number of years from 2008.



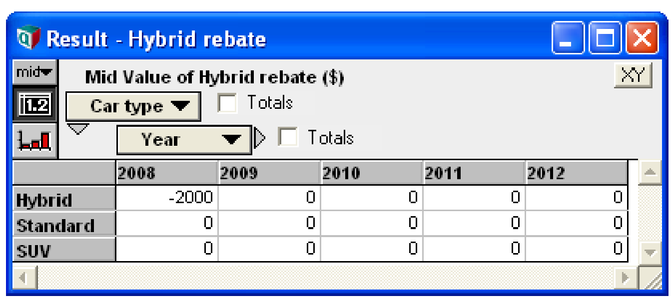
	2008	2009	2010	2011	2012
Hybrid	4320	3542	2905	2382	1953
Standard	3960	3247	2663	2183	1790
SUV	7200	5904	4841	3970	3255

IF THEN ELSE with arrays

Suppose that there is a government rebate of $2000 when you purchase a hybrid. You could create an edit table by **Car_type** and **Year** with **-$2000** for Hybrid in 2008 and **$0** in all other cells. (The rebate is negative because we are treating the numbers as costs.) A more elegant method is to define it as a conditional expression based on **Year** and **Car_type**:

```
Variable Hybrid_rebate := IF Year = 2008 AND Car_type = 'Hybrid'
                          THEN -2000 ELSE 0
```

It calculates the expression for each value of the indexes, in this case **Year** and **Car_type**, with this result.



	2008	2009	2010	2011	2012
Hybrid	-2000	0	0	0	0
Standard	0	0	0	0	0
SUV	0	0	0	0	0

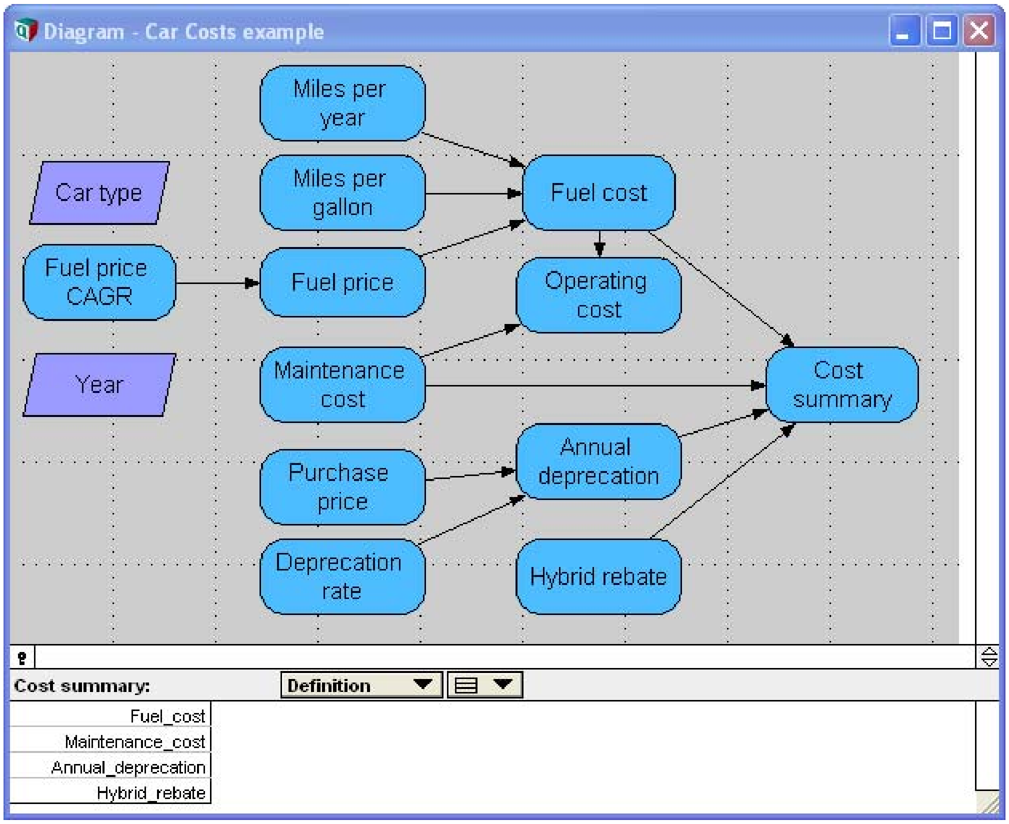
The subexpression **Year = 2008** returns an array indexed by **Year** containing 1 (true) for 2008 and 0 (false) for the other years. Subexpression **Car_type = 'Hybrid'** returns an array indexed by **Car_type**, containing 1 (true) for **'Hybrid'** and 0 (False) for the other **Car_type**. Therefore, the expression **Year = 2008 AND Car_type = 'Hybrid'** returns an array indexed by both **Year** and **Car_type**, containing 1 (true) only when both subexpressions are true, that is 1 for Hybrid in 2008 and 0 for the other cells. The entire **IF** expression therefore returns **-2000** for the corresponding top-left cell and **0** for the others. (See “IF a THEN b ELSE c with arrays” on page 161 for more.)

Compare a list of variables

To summarize the results, it is useful to compare the four types of cost, **`Fuel_cost`**, **`Maintenance_cost`**, **`Purchase_price`**, and **`Hybrid_rebate`**, in one table. Let's make a variable **`Cost_summary`**, and first define it as an empty list, i.e., square brackets with nothing between them yet:

```
Variable Cost_summary := []
```

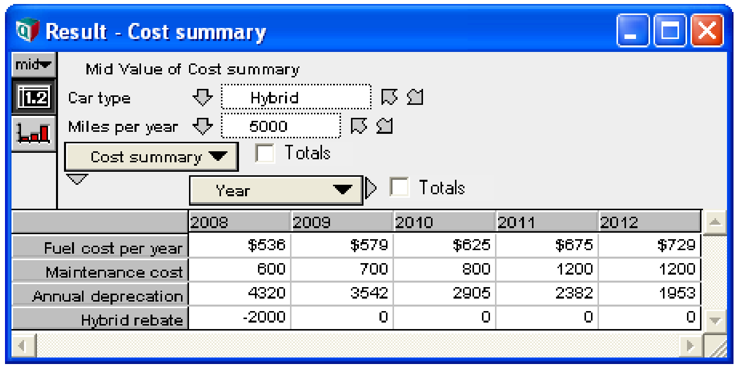
Now draw an arrow from each of the four variables you want to view to **`Cost_summary`**, in the sequence you want them to appear. Each time you draw an arrow into a variable defined as a list, it automatically adds that variable into the list. (If the origin variable was already in the list, it removes it again.) Here is the diagram showing the resulting definition for **`Cost_summary`**.



Tip This diagram does not display arrows from index nodes to avoid confusion with crossing arrows. We switched these off by restoring **Show arrows to/from** Indexes to unchecked (the default) in the **Diagram style** dialog from the **Diagram** menu.

The resulting definition is a list of variables (see "List of variables" on page 167).

The result for **`Cost_summary`** is four-dimensional, adding a new index, also labeled **`Cost_summary`**, showing the variables in the list.



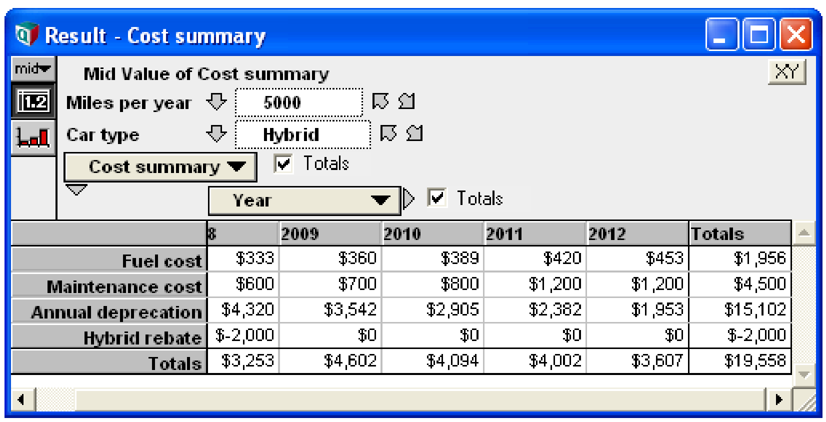
	2008	2009	2010	2011	2012
Fuel cost per year	$536	$579	$625	$675	$729
Maintenance cost	600	700	800	1200	1200
Annual deprecation	4320	3542	2905	2382	1953
Hybrid rebate	-2000	0	0	0	0

Constant value over an index not in array

Note that only **`Fuel_cost`** depends on **`Miles_per_year`**. The other three quantities, maintenance, depreciation, and rebate, are expanded over that index in the table, using the same number for each value of **`Miles_per_year`**. This is an example of a general principle: *An array that does not contain index **i** as a dimension is treated as though it has the same value over each element of **i** when there is a need to expand it to include **i** as a dimension.*

Totals in a table

To see the total over the costs and over the **`Years`**, check the two **`Totals`** boxes next to the row and column menus.



	8	2009	2010	2011	2012	Totals
Fuel cost	$333	$360	$389	$420	$453	$1,956
Maintenance cost	$600	$700	$800	$1,200	$1,200	$4,500
Annual deprecation	$4,320	$3,542	$2,905	$2,382	$1,953	$15,102
Hybrid rebate	$-2,000	$0	$0	$0	$0	$-2,000
Totals	$3,253	$4,602	$4,094	$4,002	$3,607	$19,558

Self index

The new index containing the titles of the four cost variables in the list is also called **`Cost_summary`**. Thus, the identifier **`Cost_summary`** serves double-duty as an index for itself. This is known as a ***self index***, and can be accessed using the **IndexVals()** function (see “IndexVals” on page 14).

If we want to compute the sum of the four costs, we can use **Sum(x, i)** to sum array **x** over index **i**. In this case, we sum **`Cost_summary`** over its self index, also **`Cost_summary`**:

```
Variable Total_cost_by_year := Sum(Cost_summary, Cost_summary)
```

Sum(x, i)

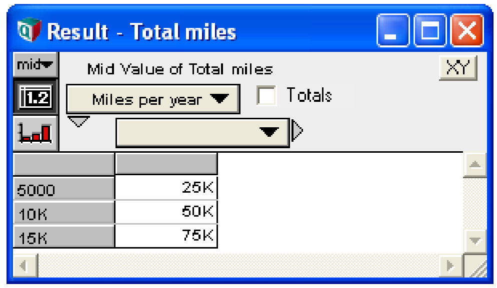
We also want to compute the average cost per mile over all the years. First we compute total cost over time, using the **Sum()** function:

```
Variable Total_cost := Sum(Cost_summary, Year)
```

As before, we need to specify the index over which we are summing, **`Year`**, but we don’t need to mention any other indexes, such as **`Car_type`** and **`Miles_per_year`**, which are irrelevant to this summation.

Next we calculate the **`Total_miles`** over **`Year`**:

```
Variable Total_miles := Sum(Miles_per_year, Year)
```



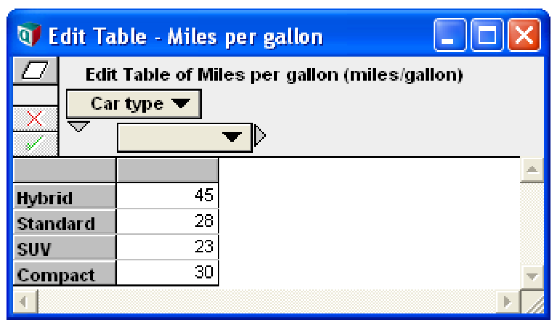
Note that **Miles_per_year** is not indexed by **Year**. The principle of *Constant value over unused indexes* implies that **Miles_per_year** has the same value for each **Year**. Hence, the result is the miles per year multiplied the number of years, in this case 5.

Finally, we define:

```
Variable Cost_per_mile := Total_cost/Total_miles
```

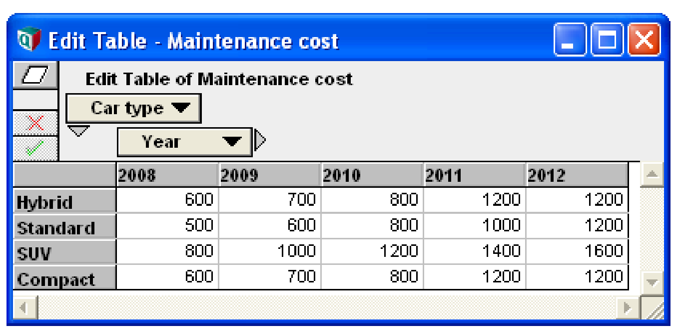
Add a new item to an index

What if you want to extend this model to include **Compact** as a fourth **Car_type**? Open one of the edit tables indexed by **Car_type**, say **Miles_per_gallon**. Click the last **Car_type**, **SUV**, to select that row (or column), and press *Enter* or the *down-arrow* key ↓. It says *"Changing the size of this index will affect table definitions of other variables. Change data in tables indexed by Car_Type?"* This warns that adding a new **Car_type** will affect all the edit tables indexed by **Car_type**. Click **OK**, and it adds a new bottom row, with the same label **SUV** as the previous bottom row, and with value **0**. Double-click the index label in this bottom row, and type the new **Car_type**, **Compact**, to replace it. Then enter its value, say **30** (miles/gallon).



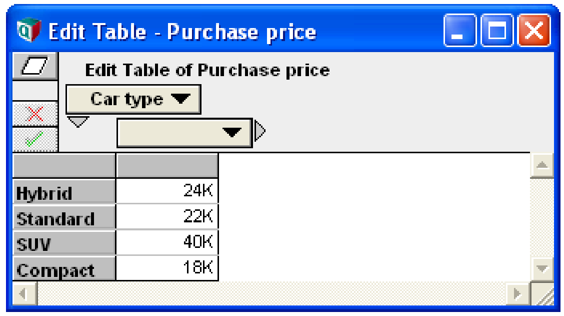
Expanding index for other edit tables

Now open the edit table for **Maintenance_cost**, and you will see a new row for **Compact** already added, initialized to **0** in each cell. You just need to enter numbers for **Maintenance_cost** for the **Compact** car, as shown here.



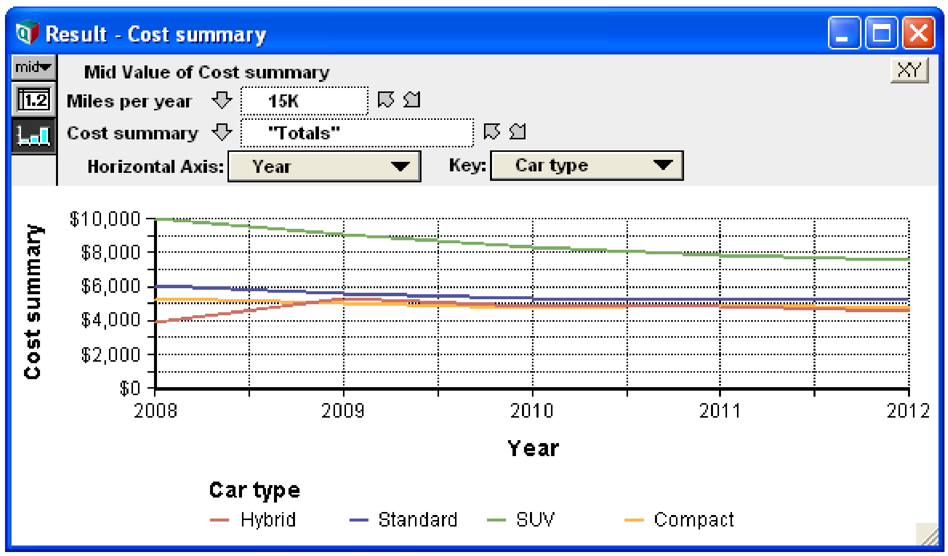
Next enter numbers for the **Maintenance_cost** for the **Compact** car.

Then enter a purchase price for the **Compact** car.



Automatic propagation of changes to index

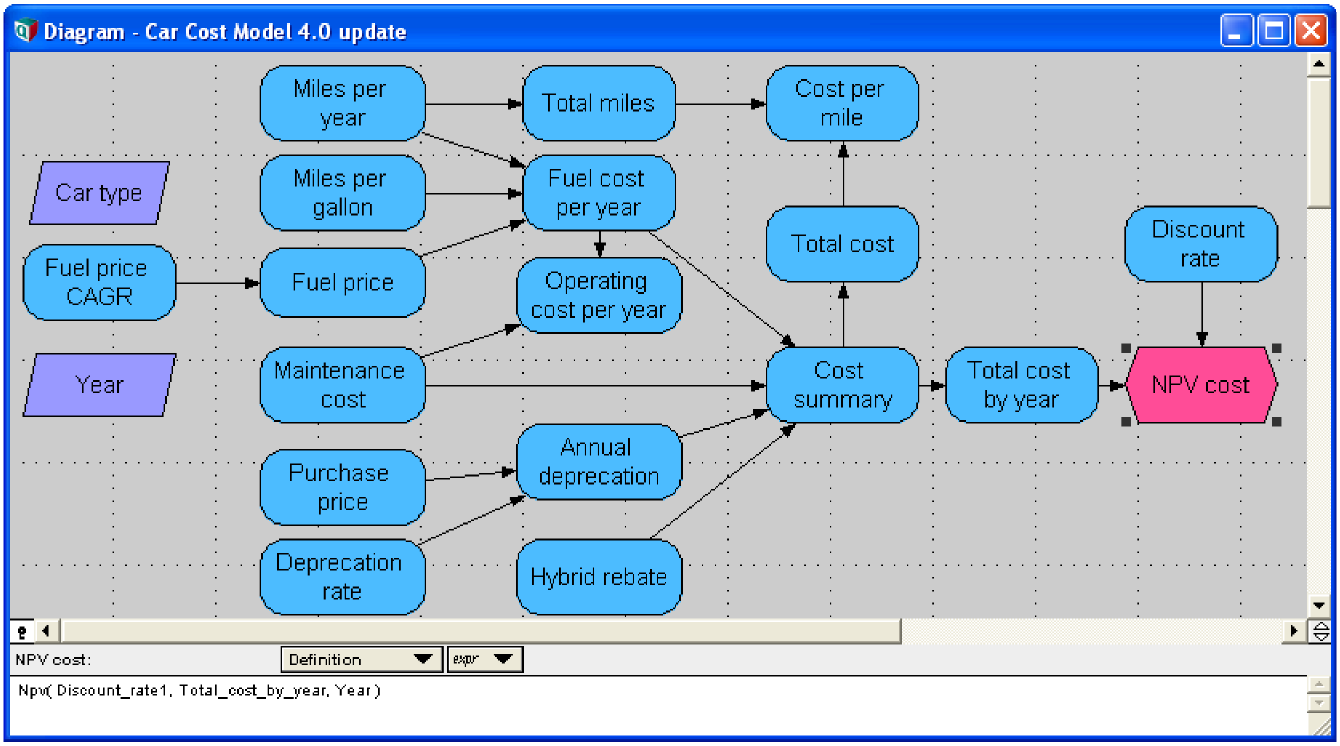
Now you've entered the data for **`Compact Car_type`** into the three edit tables, and you're done. All the computed tables automatically inherit the expanded index and do the right thing — without you needing to make any change to their definitions. For example, **`Cost_summary`** now looks like this.

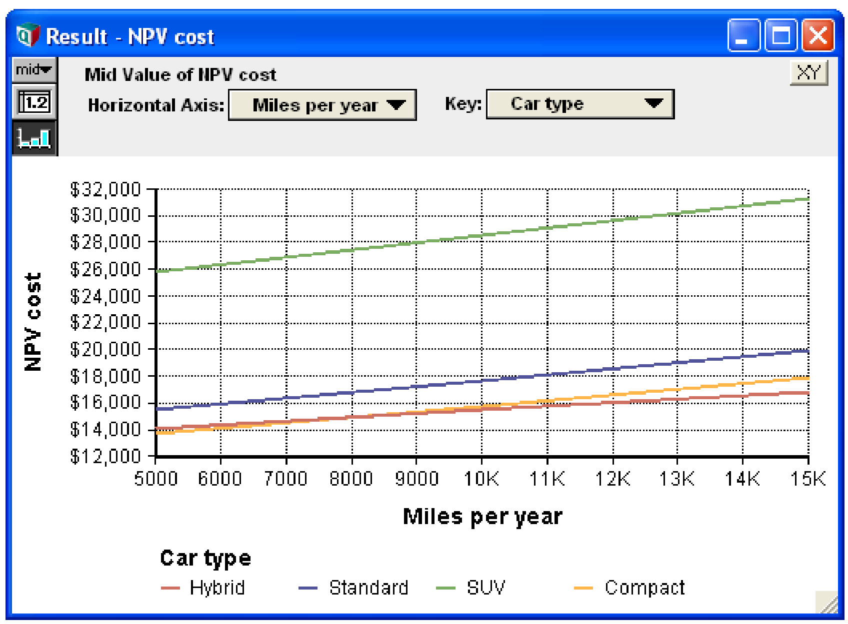


Finally, let's compute the net present value cost as the objective, using the reducing function **Npv(discount, x, i)**. (See "Npv(discountRate, values, i)" on page 212.) We define:

```
Variable Discount_rate := 12%
Objective NPV_cost := NPV(Discount_rate, Total_cost_by_year, year)
```

Here is the final diagram, showing **`NPV_Cost`**.





Monte Carlo sampling and Intelligent Arrays

Almost any variable in Analytica can be uncertain — that is, probabilistic. Each probabilistic quantity is represented by a random sample of values, generated using Monte Carlo (or Latin hypercube) simulation. Each random sample is an array indexed by a special system variable **Run**. The value of **Run** is a sequence of integers from 1 to **`Sample_size`**, a system variable specifying the sample size for simulation. See "Appendix A: Selecting the Sample Size" on page 372. For most operations and functions, **Run** is just another index, and so is handled just like other indexes by the Intelligent Arrays. You can see it when you choose the **Sample** uncertainty view. In other uncertainty views, such as **Mean** or **CDF**, the values displayed are computed from the underlying sample. See "Uncertainty views" on page 33.

Progressive refinement of a simple model

As we developed this simple model, we refined it by adding indexes progressively. First, we defined **`Car_type`**, then **`Year`**, and finally we changed **`Miles_per_year`** from a single value to a list of values for parametric analysis. Creating **`Cost_summary`** added a fourth index, consisting of the four cost categories. It is often a good idea to build a model like this — starting with a simple version of a model with no or few indexes, and then extending or disaggregating it by adding indexes — and also sometimes removing indexes if they don't seem important.

This approach to development is sometimes called ***progressive refinement***. By starting simple, you get something working quickly. Then you expand it in steps, adding refinements where they seem to be most useful in improving the representation. A more conventional approach, trying to implement the full detail from the start, risks finding that it's just too complicated, so it takes a long time to get anything that works. Or, you might find that some of the details are excessive — they just weren't worth the effort.

Progressive refinement is a much easier in Analytica than in a spreadsheet and most other computer languages — where extending or adding a dimension requires major surgery to the model to add subscripting and loops. With Intelligent Arrays, to extend or add an index, you only need to change edit tables or definitions that actually do something with the new index. The vast majority of formulas generalize appropriately to handle a modified or new dimension without needing any changes.

Summary of Intelligent Arrays and array abstraction

Analytica's Intelligent Arrays make quite easy what would be very challenging in a spreadsheet or in a conventional computer language which would force you to add loops and subscripts to every array variable every time you add a dimension.

If you find yourself using a lot of subscripts or **For** loops (see "For and While loops and recursion" on page 331), you are probably not using Intelligent Arrays properly. Take the time to understand them, and you should find that you can greatly simplify your model.

Almost every operator, construct, and function in Analytica supports array abstraction, automatically generalizing as you add or remove dimensions to their operands or parameters. (See "Ensuring array abstraction" on page 336 for the few exceptions and how to handle them if you want to make sure that your model fully supports this array abstraction.)

General principles of Intelligent Arrays™

Omit irrelevant indexes: An expression need not mention any index that it does not operate over.

A value is constant over unused index: A value (atom or array) that does not have **i** as a index is treated as constant over each value of the unused index **i** (has the same value over all values of **i**) by any construct or function that operates over that index.

Rows and columns are features of displayed tables, not arrays: You can choose which index to display over the rows or columns. You (almost) never need to care about the order in which indexes are used in an array.

The indexes of a result of an expression contain the union of the indexes of its component arrays: The result of an operation or expression contains the union of the indexes of any arrays that it uses — that is, all indexes from the arrays, without duplicating any index that is in more than one array. There are two unsurprising exceptions:

- When the expression contains an *array-reducing function or construct*, such as **Sum(x, i)** or **x[i=v]**, the result will not contain the index **i** over which it is reduced.
- When the expression creates an index, the result will also contain the new index.

To be more precise, we can define the behavior of Intelligent Arrays thus: For any expression or function **F(x)** that takes a parameter or operand **x** that might be an array indexed by **i**, for all values **v** in index **i**:

```
F(x[i=v]) = F(x)[i=v]
```

In this way, Analytica combines arrays without requiring explicit iteration over each index.

Exceptions to array abstraction

The vast majority of operators, constructs, and functions fully support Intelligent Arrays — that is, they generalize appropriately when their operands or parameters are arrays. However, very few do not accept parameters that are arrays, notably the sequence operator (**`..`**), **Sequence()** function, and **While** loop. When you use these, you need to take special care to ensure that your models perform array abstraction conveniently when you add or modify dimensions. See "Ensuring array abstraction" on page 336 for details.

IF a THEN b ELSE c with arrays

The IF a THEN b ELSE c (page 135) construct 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

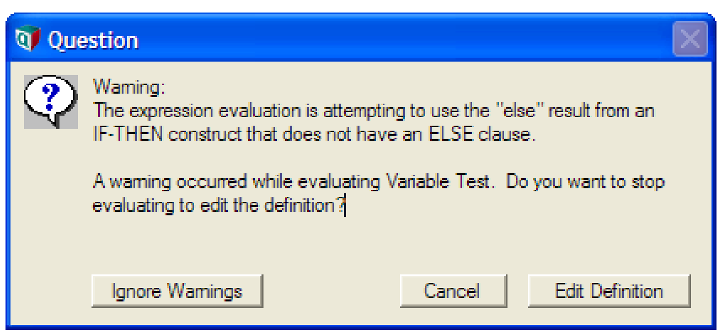
In this case, the expression **`Sqrt(X)`** is also indexed by **`X`**. The **`IF`** expression evaluates **`Sqrt(X)`** for each value of **`X`**, even the negative ones, which return **`NAN`**, even though they are replaced by **`Imaginary`** in the result.

To avoid evaluating all of b or c

When **`If a Then b Else c`** is evaluated, the expression **a** is first evaluated completely. If its result is true, or if it results in an array where any value in the array is true, then the expression **b** is evaluated completely as an array operation, meaning the expression is evaluated for all values of all indexes contained within **b**. Similarly, if **a** is false or **a** is an array and any element is false, the expression is **c** is evaluated in its entirety. Once both are computed, the appropriate values are picked out of these results according to the result of **a**. 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 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" on page 328.

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 does 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” on page 328), 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, as shown in this example:

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

Caveats of conditional side effects

In the expression above, the empty brackets following **A** define **A** as array with no indexes (i.e., as atomic). Analytica will ensure that within the body of the expression where **A** is used, **A** will always be atomic, even if **X** or **Y** are array-valued. To do this, Analytica might need to iterate the expression. If you feel compelled to embed an assignment inside a **THEN** or **ELSE** clause, you should always make sure that the condition being tested is a *scalar* and not an array. In this case, because **A** has been declared to be 0-dimensional, the expression **A<0** is guaranteed to be scalar. If you cannot guaranteed that the **IF** clause will always be scalar, even if other indexes are added to your model in the future, then you should avoid using assignment from within a **THEN** or **ELSE** clause since Analytica evaluates IF-THEN-ELSE and an array operation. Without the brackets declaring A to be scalar, the **IF** clause would say “IF any value of A is less than zero THEN evaluate the assignment”, so the result would be an array of zeroes even if there is only a single negative number in **X** and **Y**. A better way to encode a conditional assignment, which properly array abstracts and has the intended effect, is as follows:

```
BEGIN
    VAR A := Min( [X,Y] );
    A := IF A<0 THEN 0 ELSE A;
    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 might 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 might 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 can 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.

Summarizing IF, IFALL, and IFONLY

In most cases, you can just use **IF** without worrying about **IFALL** or **IFONLY**. The only reason to use **IFALL** is when you want to avoid the possibility that the dimensions of results can 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 strongly recommended except when it is one of multiple statements, and not the last, in a compound expression, enclosed in parentheses or **`BEGIN ... END`**.

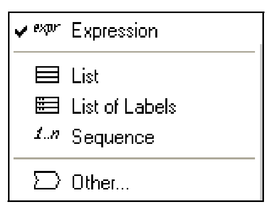
Creating an index

An ***index*** is a class of variable used to identify a dimension of an array. The same index can identify the same dimension shared by many arrays. Sometimes, variables of other classes, such as a decision, can also be used as an index to identify a dimension of an array. For clarity, use an index variable whenever possible.

You create an index much like any other variable:

Create an index node

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. Open the definition attribute for the new index:
   - Either double-click the index node to open its **Object** window
   - Or, select the index node, open the **Attribute panel** (page 24) and select **Definition** (page 108) from the **Attribute** menu.
5. Press the ***expr*** menu above the definition field, to see these options.

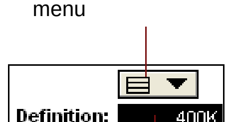


(If the variable already has a definition, Analytica confirms that you wish to replace it. Click **OK** to replace the definition with a one-element list.)

Define as a List

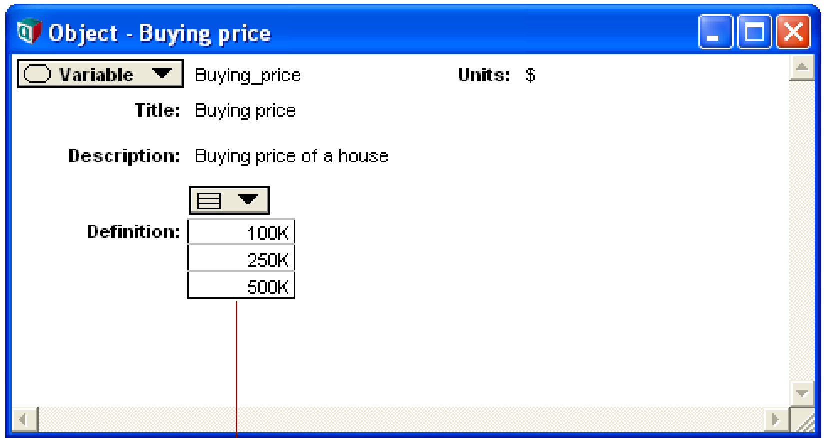
6. Select **List** (of numbers) or **List of Labels** according to whether you want to enter a list of numbers or text values. It will display a list with one item in the definition field.

List icon for the ***expr*** popup menu



New one-element list

7. Click the cell to select it, and Type in a number for **List** or text for **List of Labels**.
8. Press *Enter* or *down-arrow* to add a cell for the next item. Type in its value.
9. Repeat until you have entered all the values you want.

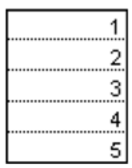


Values entered into a list

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.

Expression view You can display a list or list of labels as a ***list view***, the default view showing as a column of cells, or as an ***expression view***, showing it as a list of items between square brackets. Select from the toolbar to show the **expression view**. For example, here is a list of numbers in each view.

List view



Expression view

[1, 2, 3, 4, 5]

List of labels In a list of labels, every value is text. In the expression view, each label is enclosed in single quotation marks.

List view

Expression view

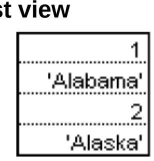
['Alabama', 'Alaska', 'Arizona','Arkansas']

To include a single quote (apostrophe) as part of the text in a label in expression view, insert two adjacent single quotes, or enclose in double quotes (see “Text values” on page 133):

['can''t','won''t','didn''t']

Mixing numbers and text A list can include a mix of text and numbers. In both views the text is contained in single quotation marks as shown below.

List view



Expression view

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

If you attempt to mix numbers and text in a list of labels, all the values are treated as text, as shown below.

List view

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. If you are defining an index object, whose sole purpose should be to serve as an index and not as an array result, then each element should evaluate to a scalar; otherwise, a warning will result. For general variables, the use of expressions that return array results is often very useful.

Editing a list

You can edit a list by changing, adding, or deleting ***cells*** (list items).

Insert a cell 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 add a cell at the end of the list, select the last cell and press *Enter* or the *down-arrow* key.

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. It duplicates the value of the last selected cell as the default for the new cells.

Delete a cell To delete one or more contiguous cells, select them and:

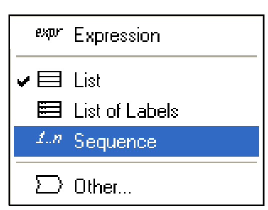
- Choose **Delete Rows** from the **Edit** menu.
- Or, just press *Control+k* or *Delete*.

Tip If you add or delete a cell in a list that is an index of one or more edit tables, it will warn you that it will change the corresponding slices of the tables (see “Editing a table” on page 171).

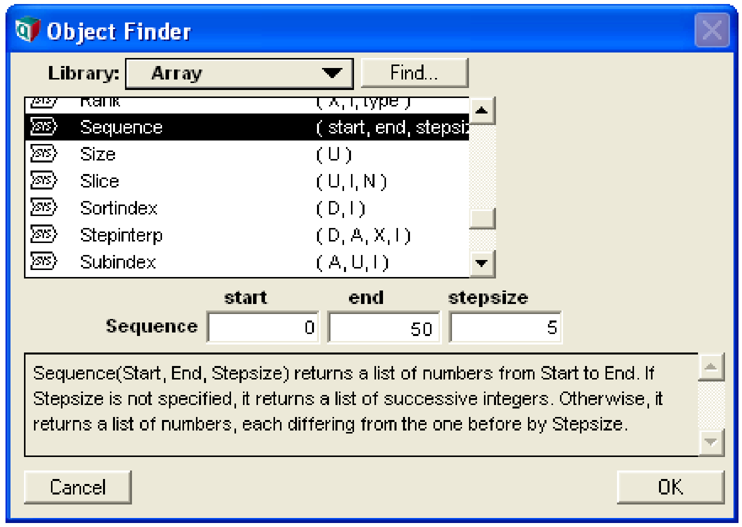
Navigating a list Use the *up* and *down-arrow* keys to move the cursor up and down the list, or simply click the cell you want.

Defining an index as a sequence

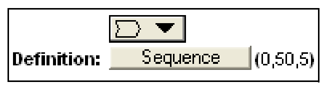
Create a list with the Sequence option To define an index as a list of equally spaced numbers, it is usually easier to select the **Sequence** function from the ***expr*** menu (instead of **List**).



Then it shows the **Sequence()** function in the **Object Finder** (page 167).



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



For more see “Sequence(start, end, stepSize)” on page 167.

Tip To change the start, end, or stepsize parameters of a sequence, click the **Sequence** button.

To define an index as a sequence of successive integers, you can use the “`..`” operator in the expression view, for example:

```
Index Year := 2000 .. 2012
```

See “m .. n” on page 167.

Functions that create indexes

It is usually easiest to define an index as a list, list of labels, or sequence, as described above (see “Creating an index” on page 163). Sometimes, you need to define an index using a more general expression, as a list of expressions, a list of variables, or a function such as **Subset()**, **Concat()**, and **SortIndex()**. This section describes these and other functions that you can use to create indexes.

$[\, u_1, u_2, u_3, \ldots u_m \,]$

A simple way to define an index is specify its definition as a list of values separated by commas and surrounded by square brackets. The values can be numbers, text values, or other expressions.

Examples

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

These lists are equivalent to using the **List** or **List of Labels** options in the ***expr*** menu, as described in “Creating an index” on page 163.

List of variables

A list of variables contains identifiers of variables in square brackets, separated by commas. Usually, the simplest way to create a list of variables is to define the variable initially as an empty list, for example:

```
Variable CompareVars := []
```

When you draw an arrow from a variable, **A**, into **CompareVars**, it will automatically add **A** as the next item in the list:

```
CompareVars := [A]
```

Suppose you draw arrows from **B** and **C**, the definition will become:

```
CompareVars := [A, B, C]
```

When you draw an arrow from a variable already in the list, it removes it from the list. Suppose we draw an arrow from **B** to **CompareVars**, it will become:

```
CompareVars := [A, C]
```

The result of **CompareVars** is an array of the values of the variables it contains, with a self index, also called **CompareVars**, that usually shows the titles of the variables.

If any or all the variables contain arrays, the result contains the union of the indexes of the contained variables. For example if **A** is an atom (not an array) and **C** is indexed by **c**, the result will be indexed by **I**. The slice of **CompareVars** for **A** will have the same value of **A** repeated for each value of **A**. See "Compare a list of variables" on page 155 for an example.

Self index The result will contain an extra index, a ***self index*** of **CompareVars**, comprising the list of the variables.

Clickable titles or identifiers in table Usually these display the titles of the variables in a table or graph result. (If you select **Show by Identifier** from the **Object** menu (or press *Control+y*) it toggles to show the identifiers instead of titles. If you double-click a title (or identifier) in a table, it will open the **Object** window for that variable. The values in the self index are actually ***handles*** to the variables. See "Handles to objects" on page 344 for more.

m .. n

Returns a sequence of successive integers from **m** to **n** — increasing if **n < m**, or decreasing if **n** > **m**. For example:

```
2003..2006 → [2003, 2004, 2005, 2006]
5 .. 1 → [5, 4, 3, 2, 1]
```

It is equivalent to **Sequence(m, n)**.

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" on page 337 on how to use this 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 you omit **stepSize**, it uses increments of 1. **start**, **end**, and **stepSize** must be deterministic scalar numbers, not arrays.

You can also select this function using the **Sequence** option from the ***expr*** menu, as described in "Create a list with the Sequence option" on page 165.

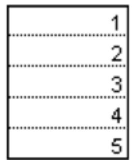
The expression **m .. n** using the operator **".."** is equivalent to **Sequence(m, n, 1)**.

Library Array

Examples If **end** is greater than **start**, the sequence is increasing:

```
Sequence(1,5) →
```

List view



Expression view

`[1,2,3,4,5]`

If **`start`** is greater than **end**, the sequence is decreasing:

```
Sequence(5, 1) → [5, 4, 3, 2, 1]
```

If **`start`** and **end** are not integers, and you omit **stepSize**, it rounds them:

```
Sequence(1.2, 4.8) → [1, 2, 3, 4, 5]
```

If you specify **`stepSize`**, it can create non-integer values:

```
Sequence(0.5, 2.5, 0.5) → [0.5, 1, 1.5, 2, 2.5]
```

Concat(i, j)

Returns a list containing the elements of index **i** concatenated to the elements of index **j**. Thus the number of items in the result is the sum of the number of items in **i** and the number of items in **j**. See “Concat(a1, a2, i, j, k)” on page 197 for how to concatenate two arrays.

```
Index Year1 := 2006 .. 2008
Index Years2 := 2009 .. 2010
Index YearsAll :=Concat(i, j) → [2006, 2007, 2008, 2009, 2010]
```

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(YearsAll < 2010) → [2006, 2007, 2008, 2009]`

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, which are each the same set of cities:

```
Index Origins
Definition:= ['London', 'New York', 'Tokyo', 'Paris', 'Delhi']
Index Destinations
Definition:= CopyIndex(Origins)

Variable Flight_times := Table(Origins, Destinations)
```

If you defined **`Destinations`** as equal to **`Origins`**, without using **Copyindex()**, **`Destinations`** would be indexed by **`Origins`**, and the resulting table would have only one dimension index. By defining **`Destinations`** with **CopyIndex()**, it becomes a separate index, so that the table has two dimensions.

Sortindex(d, *i*)

Assuming **d** is an array indexed by **i**, **SortIndex()** returns the elements of index **i**, reordered so that the corresponding values in **d** would go 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 this:

```
d[i=Sortindex(d, i)]
```

When **d** is a one-dimensional array, the index parameter **i** is optional. When omitted, the result is an unindexed list. Use the one-parameter form only when you want an unindexed result, for example to define an index variable. The one-parameter form does array abstract when 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) →
Car_type ▶
```

	VW	Honda	BMW
	Honda	VW	BMW

```
SortIndex(Maint_costs) →
SortIndex ▶
```

	Honda	VW	BMW

Define **`Sorted_cars`** as an index node:

```
INDEX Sorted_cars := Sortindex(Maint_costs)
Maint_costs[Car_type = Sorted_cars] →
```

	Honda	VW	BMW
	1800	1950	2210

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]
```

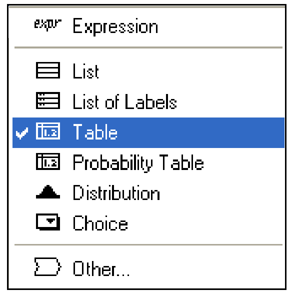
Defining a variable as an edit table

To define a variable as an edit table, you choose **Table** from the ***expr*** menu above its definition:

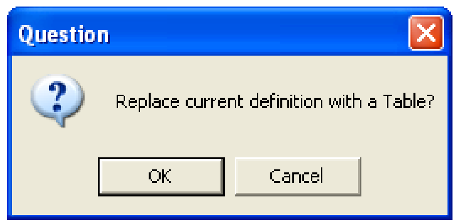
1. Select the variable and open its definition using one of these options:
   - Use the variable's **Object** window.

- From the **Attribute** panel of the **Diagram** window, select **Definition** from the **Attribute** popup menu.
- Press *Control+e*.

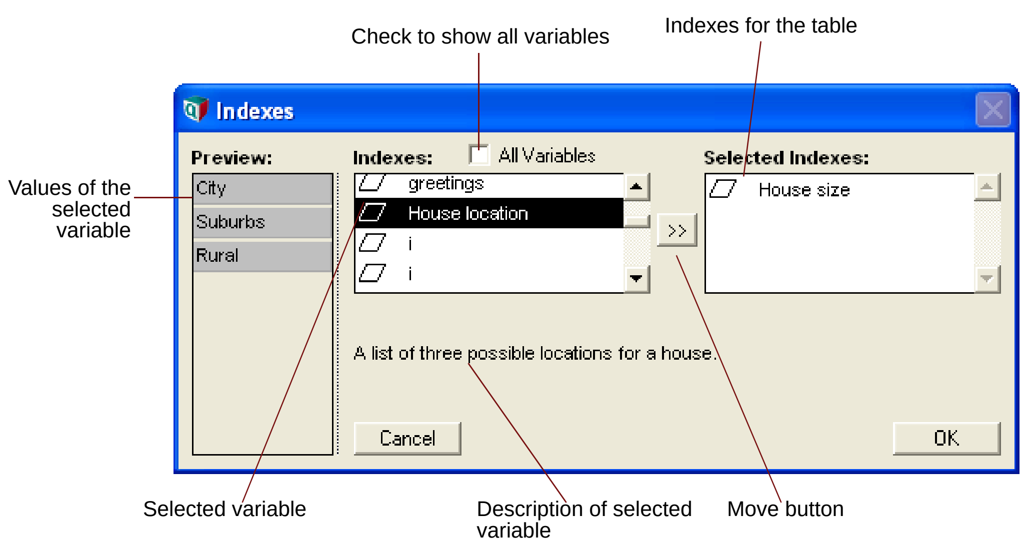
2. Press the ***expr*** menu above the definition field and select **Table**.



If it already has a definition, click **OK** to confirms that you wish to replace it.



3. It opens the **Indexes** dialog so you can select the table's indexes (dimensions). It already lists under **Selected indexes** any index variables from which you have drawn an arrow to this variable. You can 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.

5. Click **OK** to create the table and open the **Edit Table window** (page 171) for editing the table's values.

Indexes dialog

The **Indexes** dialog contains these features (see figure above):

Preview	A list of the values of the selected index variable. If the selected variable is not a list, it says "Can't use as index."

All Variables checkbox	If checked, the *Indexes* list includes all variables in the model. If not checked, it lists only variables of the class **`Index`** and **`Decision`**, plus the variable being defined (**`Self`**) and **`Time`**. If you select **`Self`** as an index, the variable itself holds the alternative index values.
Selected Indexes	A list of all indexes already selected for this variable.
New index	Select to create a new index.

To create an index

You can create an index variable in the course of creating a table, in the following way:

1. Select ***new index*** from the **Indexes** list in the **Indexes** dialog.
2. Enter a title for the index.

3. Click the **Create** button.
4. To make the new index an index of the table, click the >> button.
5. Enter the values of the Index in the **Edit Table** window (see the following section).

To remove an index from an array

1. Select the index from the **Selected Indexes** list.
2. 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 variables like any 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** (page 37) or the **Sample()** function (page 266).

`Time` is the index for **dynamic simulation** (page 281). It is the only index permitted for cyclically dependent modeling.

Editing a table

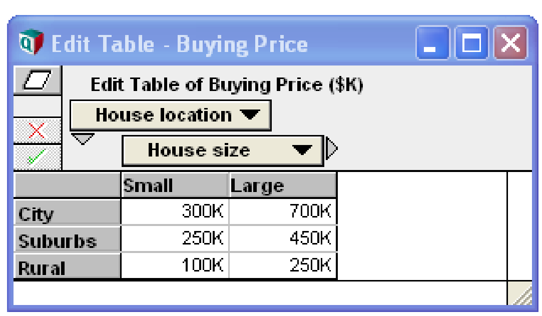
To open the **Edit Table** window, click the **Edit Table** button in either:

- The **Object window** (page 23)
- The **Attribute panel** (page 24) of the diagram

  In the **Attribute** panel, select **Definition** (page 108) from the **Attribute** popup menu.

The Edit Table window

The **Edit Table** window looks much like the **Result** window **table view** (page 32). The difference is that you can add indexes and edit the values in cells.



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 177 for more. Press *Enter* to accept the value and to select the next cell, or click in another cell.

Tip You can 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 the 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 also accepts 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 .. 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. See "Splice a table when computed indexes change" on page 173 for more information.

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" on page 121.

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 happens 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. You can change this default with the system variable **`Sys_tableCellDefault`**. You can also set table-specific default values, using the **`TableCellDefault`** attribute. See "Splice a table when computed indexes change" on page 173 for details.

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 add an item to an index or delete an item from an index that is also used by another 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. Adding an item will add a new slice containing zeros, just as it does for the one you are editing. Similarly, deleting an item will delete a slice from these other edit table.

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 to prevent them from 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** (page 170). 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 sub-array 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**.
- Or, click in the edit table to open the **Indexes dialog** (page 170). 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.

Splice a table when computed indexes change

A computed index is an index that depends on other variables (that is, not an explicit list of numbers or labels). Computed indexes use functions that return indexes, such as **Sequence()**, **Concat()**, or **Subset()**, for example:

```
Index Year := Start_year .. Horizon_year
Index K := Concat(i, j)
Index S := Subset(Year < 2002)
```

Splicing is what happens to an editable table (table, determtable, or prob table) when it uses a computed index that changes because of a change to one of its inputs. The change can cause slices to be added, deleted, or reordered. By default, if the changed index has an item with the same value (number or text) as the previous version, all editable tables retain the old data for the slice identified by that item, even if items are removed, reordered, or added. For example:

```
Variable Start_year := 2005
Index Year := Start_year .. (Start_year+2)
Variable Revenues := Table(Year)(100, 200, 300)
Revenues→
Year ▶
```

	2005	2006	2007
	100	200	300

Suppose, you change:

```
Start_year := 2006
```

Then by default, **Revenues** will change to:

```
Year ▶
```

	2006	2007	2008
	200	300	0

Thus, it loses the cell for **2005**. Cells for **2006** and **2007** retain their original values, and it adds a new cell with default **0** for the new year, **2008**. This is called ***associational correspondence***, because it retains the association between index label and value, even if the positions change.

Alternatively, if you change one or more index values to new text labels or numbers, it retains the same values of for the ***n***th slice, even though the index value changes. This is called ***positional correspondence***, because it retains correspondence where the nth position contains the same value.

The default splicing behavior is ***mixed correspondence***, preserving *associational* correspondence where labels are the same, and preserving *positional* correspondence where possible otherwise. It is possible to change this splicing behavior for each editable table to ***pure associational correspondence*** — retaining values *only* where index values are the same — or ***pure positional correspondence*** — going *only* by position in the index, irrespective of index values. See attribute CorrespondenceMethod in the Analytica wiki for details.

Subscript and slice of a subarray

These constructs and functions let you select a slice or subarray out of an array.

x[i=v]: Subscript construct

This is the most common method to extract a subarray:

```
x[i = v]
```

It 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 reason is that if an array **x** is not indexed by **i**, it means **x** is constant over all values of **i**. (The principle is described in “Constant value over an index not in array” on page 156.)

You can apply the subscript construct to an expression, simply by putting the square bracket immediately after the expression:

```
(Revenue - Cost)[Time = 2010]
```

Indexing by name not position

You can subscript over multiple dimensions, for example:

```
x[i=v, j=u]
```

The ordering of the indexes is arbitrary, so you get the same result from:

```
x[j=u, i=v]
```

Indexing by name means that you don't have to remember or use any intrinsic ordering of indexes in an array, such as rows or columns, inner or outer, common to most computer languages.

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**.

Subscript(x, i, v)

This function is identical to the subscript construct **`x[i=v]`**, 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 associating 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]
```

And also like the subscript construct, the ordering of the indexes is arbitrary.

Mixing subscript and slice constructs

You can mix slice and subscript operations in the same expression in any order:

```
x[@i=1, j=2, k=3]
```

Slice(x, i, v)

This function is identical to the slice construct **`x[@i=v]`**, using different syntax.

Slice(x, n)

If **Slice()** has only two parameters, and **x** has a single dimension, it returns the **n**th element of **x**. 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 **x** is a scalar, or if **x** is an array with two or more indexed dimensions and no unindexed dimensions, **Slice(x, n)** simply returns **x**.

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

Preceding time slice: x[Time-1]

x[Time-n] refers to the built-in index **`Time`** (see "The Time index" on page 282). It returns the value of variable **`x`** for the time period that is **n** periods prior to the current time period. This function is only valid inside the **Dynamic()** function (page 282).

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 121). **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 the popup (**inclAll** defaults to True).

Examples **`Choice(Years, 2)`** → **`1986`**

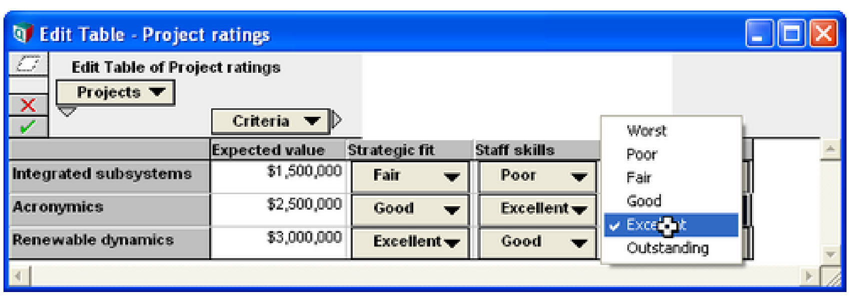
If **`n`**=0, and **inclAll** is true, it returns all values of **i**:

`Choice(Years, 0, 1)` →
`Years` ▶

	1985	1986	1987	1988

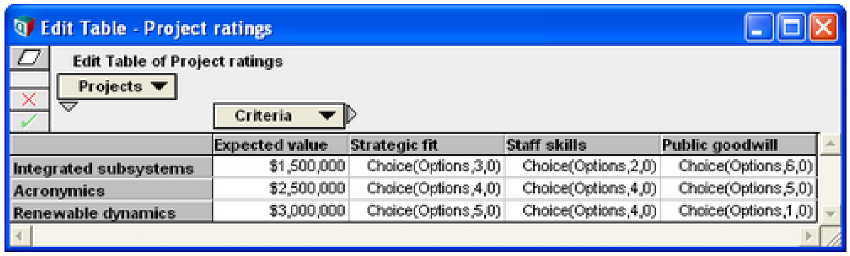
Choice menus in an edit table

You can include a drop-down (pull-down) 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 (page 176) in the edit table cells, similar to using **Choice** to specify a single menu for a variable:

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.
4. Copy and paste **`Choice(k, 1, 0)`** from the first cell to any others you want also to contain the menu. You can also use other indexes than **k** if you want to include menus with other options. Here is an example viewed in edit mode, with drop-down menus in some but not all cells.



5. 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+Page Up* and *Control+Page Down* 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

Mouse Click	Click in a cell to make it the current cell.
Mouse Shift+Click	Select the region from the previous anchor to this cell.
Mouse drag	Select the region from the cell in which you depress the left mouse button to the cell in which you release the button.
Mouse wheel	Scroll vertically without changing the selection.
Control+mouse wheel	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:

down-arrow	If you have selected the last row, add a row.
left-arrow	If you have selected the right column, add a column.
Control+i	If you have selected a row header, insert a row. If you have selected a column header, insert a column.
Control+k	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. Search from the current cell and select 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+Page Up*, *Control+Page Down* 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.

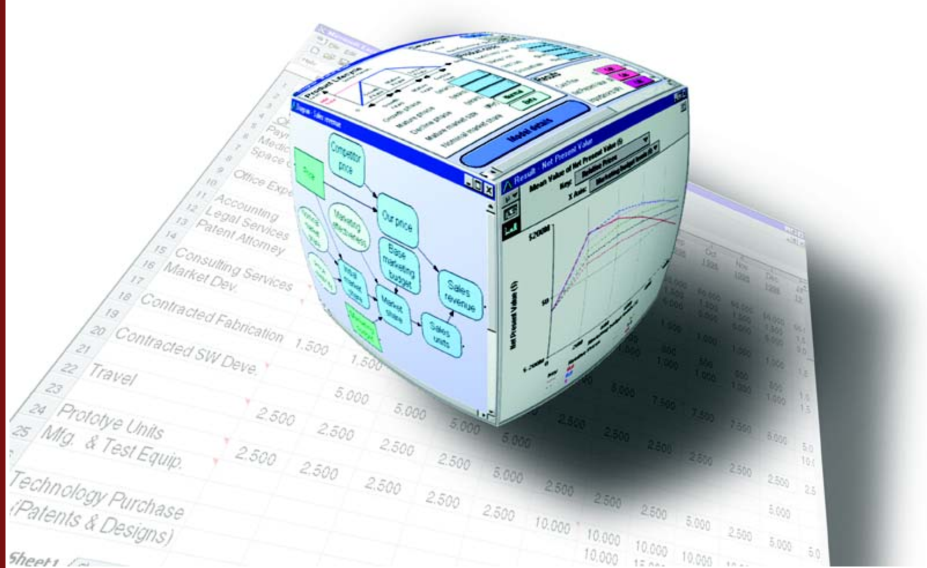
Other keys

Tab	Move one cell right. Same as right arrow.
Shift+Tab	Move one cell left. Same as left arrow.
Enter, Shift+Enter	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.
Return	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.
Shift+Return	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.

Control+a	Select all (body) cells. If a row/col header is selected, selects all rows/cols.

Chapter 12 *More Array Functions*

This chapter describes a variety of more advanced array functions, including functions that:



This chapter describes several classes of function and other constructs that work with arrays. If you have not already ready it, we recommend that you read “Introducing indexes and arrays” on page 144 in the previous chapter, before reading about the functions in this chapter.

Example variables Several examples in this chapter refer to these indexes and array variables:

```
Index Car_type := ['VW', 'Honda', 'BMW']
Index Years := 1985 .. 1988
Index Mpg := [26, 30, 35]
Index Time := 0 ..4
Variable Cost :=
Mpg ▼, Car_type ▶
```

	VW	Honda	BMW
26	2185	2810	3435
30	1705	2330	2955
35	1585	2210	2835

```
Variable Car_prices :=
Car_type ▼, Years ▶
```

	1985	1986	1987	1988
VW	8000	9000	9500	10K
Honda	12K	13K	14K	14.5K
BMW	18K	20K	21K	22K

```
Variable Cost_in_time :=
Mpg ▼, Time ▶, Car_type = VW
```

	0	1	2	3	4
26	2185	2294	2409	2529	2656
30	2810	2951	3098	3253	3416
35	3435	3607	3787	3976	4175

```
Mpg ▼, Time ▶, Car_type = Honda
```

	0	1	2	3	4
26	2385	2314	2529	2649	2856
30	2910	3041	3238	3343	3526
35	3535	3847	3897	4166	4365

```
Mpg ▼, Time ▶, Car_type = BMW
```

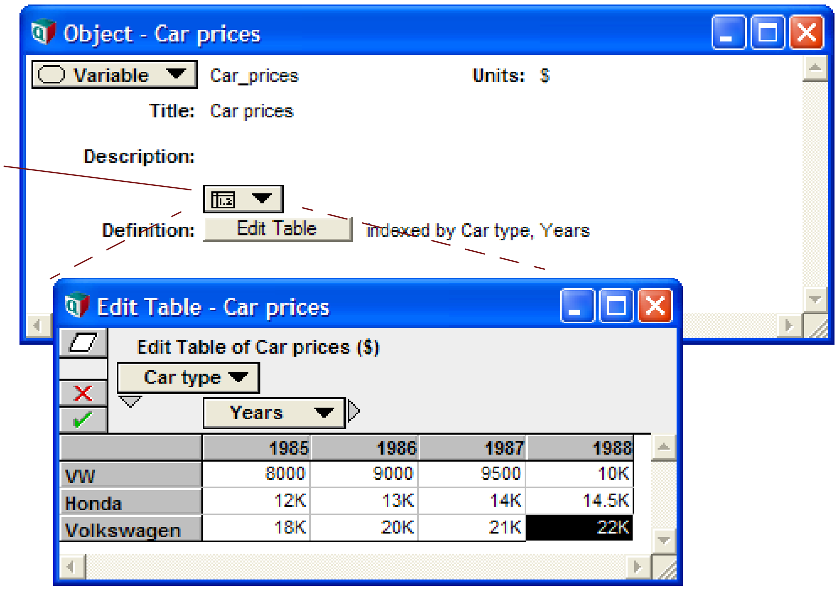
	0	1	2	3	4
26	3185	3294	3409	3529	3656
30	3810	3951	4098	4253	4416
35	4435	4607	4787	4976	5175

Functions that create arrays

Usually, the most convenient way to create an array of numbers or text values is as an ***edit table***. When viewing the definition of the variable, choose **Table** from the ***expr*** menu to create an edit table (see “Defining a variable as an edit table” on page 169). If you want to define a table by explicitly listing its indexes and providing expressions to generate its values or subarrays, you might find **Array()** more convenient.

An array viewed as a table

Table



If you select **expr** from the ***expr*** menu, it displays it as a table expression in the **Definition** field (rather than a separate edit table), listing the indexes and values.

An array viewed as an expression

expr menu

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** is an expression returning an array with at least **n** dimensions, each dimension with the number of elements matching the corresponding index. You can use array to change the index variable(s) from one to another with the same number(s) of elements. **Array()** is one of the few places where you actually need to worry about the order of the indexes in the array representation.

Use **Array()** to specify an array directly as an expression. **Array()** is similar to **Table()** (page 185); 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:

```
Index Car_type := ['VW', 'Honda', 'BMW']
Array(Car_type, [32, 34, 18])
```

Definition viewed as a table:

Car_type ▶

	VW	Honda	BMW
	32	34	18

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)" on page 199.

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.

```
Index Car_model := ['Jetta', 'Accord', '320']
Variable Table_a:= Table(Car_type) (32, 34, 18)
Variable Table_b:= Array(Car_model, Table_a) →
Car_model ▶
```

	Jetta	Accord	320
	32	34	18

Tip There are some significant disadvantages to using the **Array()** function to change the index of an array in the fashion demonstrated in Example 4. Specifically, if a second dimension were later added to **`Table_a`**, the index that the **Array()** function changes might not be the one you intended. The preferred method for changing the index, which does fully generalize when **`Table_a`** has many dimensions, is to use the slice operator (see Tip on re-indexing) as follows:

```
Table_a [ @Car_type = @car_model ]
```

Table(i1*, i2, … in*) (u1, u2, u3, … um)

This function is automatically created when you select **Table** from the ***expr*** menu to create an edit table. You can view it as an expression in this form in the definition of the variable by selecting **expression** from the ***expr*** menu. It. 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()** (page 183); **Table()** requires **m** numeric or text values.

A definition created as a table from the ***expr*** 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 of the definition without the second set of parameters:

```
Table(Car_type, Years)
```

It looks like this when viewed as an edit table:

Car_type ▼, Years ▶

	1985	1986	1987	1988
VW	0	0	0	0
Honda	0	0	0	0
BMW	0	0	0	0

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. Examples include, **Sum(x, i)**, **Product(x,i)**, **Max(x, i)**, **Min(x, i)**, and others described below. The subscript construct **x[i=v]** and related subscript and slice functions also reduce arrays by a dimension (see “Subscript and slice of a subarray” on page 174).

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
```

See “Example variables” on page 182 for example array variables used here and below.

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 might 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.

Reducing over an unused index If the index, **i**, is not a dimension of **x**, **Sum(x, i)** returns **x** unreduced (i.e., with the same number of indexes), 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
```

This is because **Car_type** has three elements (3 x 5 = 15). For **Product**:

```
Product(x, Car_type) → 125
```

That is, it multiplies **x** three times (5^3 = 125).

In this way, if we later decide to change the value for **x** for each value of **Car_type**, we can redefine **x** as an edit table indexed by **Car_type**. Any expression containing a **Sum()** or other reducing function on **x** works correctly whether it is indexed by **Car_type** or not.

Elements that are ignored The array-reducing functions described in this section ignore elements of an array that have the special value **Null**. For example, the **Average(x,i)** function sums all the non-null elements of **x** and divide by the number of elements that are not **null**.

When a **NaN** value (signifying an indeterminate number) appears as an element of an array, the result of the function that operates on the array will usually be **NaN** as well. **NaN** values result from indeterminate operations such as 0/0, and the fact that they propagate forward in this fashion helps ensure that you will not accidentally compute an indeterminate result without realizing it. However, in some cases you might wish to ignore **NaN** values in an array-reducing operation. The array-reducing functions **Sum**, **Product**, **Average**, **Min**, and **Max** all accept an optional parameter, **ignoreNaN** that can be set to **True**. **IgnoreNan** requires a named-parameter syntax, for example:

```
Max(x,i,ignoreNaN:True)
```

When you operate over an array containing some text and some numeric values, the **Sum**, **Min** and **Max** functions can be instructed to ignore all the non-numeric values using an optional **ignoreNonNumbers** parameter, for example:

```
Max(x,i,ignoreNonNumbers:True)
```

Reducing over multiple indexes

The array-reducing functions **Sum**, **Product, Average**, **Min**, **Max**, **ArgMin**, and **ArgMax** all allow you to specify more than one index as a convenient way to reduce over multiple indexes in a single call. For example:

```
Sum(x,i,j,k)
```

This is equivalent to:

```
Sum(Sum(Sum(x,i),j),k)
```

Sum(x, *i*)

Returns the sum of array **x** over the dimension indexed by **i**.

Library Array

Examples

`Sum(Car_prices, Years)` →
`Car_type` ▶

	VW	Honda	BMW
	36.5K	53.5K	81K

See "Example variables" on page 182 for example array variables used here and below.

Product(x, *i*)

Returns the product of all of the elements of **x**, along the dimension indexed by **i**.

Library Array

Examples

`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(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(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`

See "Example variables" on page 182 for example array variables used here and below.

Min(x, *i*)

Returns the lowest valued element of **x** along index **i**.

Library Array

Examples

```
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 none

Examples

```
CondMin(Cost_in_time, Time>=2, Time)→
Mpg ▼, Car_type ▶
```

	VW	Honda	BMW
26	2409	2529	3409
30	3098	3238	4098
35	3787	3897	4787

Subindex(a, u, i)

Returns the value of index **i** for which array **a** (indexed by **i**) is equal to **u**. If more than one value of **a** equals **u**, it returns the last value of **i** that matches **u**. If no value of **a** equals **u**, it returns

`Null`. If **a** has index(es) in addition to **i**, or if **u** is an array with other indexes, those indexes also appear in the result.

Library Special

Examples

```
Subindex(Car_prices, 12K, Car_type) →
Years ▶
```

	1985	1986	1987	1988
	Honda	«Null»	«Null»	«Null»

```
Subindex(Car_prices, 12K, Years) →
Car_type ▶
```

	VW	Honda	BMW
	«Null»	1985	«Null»

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	«Null»	«Null»	«Null»
21K	«Null»	«Null»	BMW	«Null»

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 is 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()** would return an ambiguous result.

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 “@: Index Position Operator” on page 190) returns the same result as `PositionInIndex(,x,i)`:

```
PositionInIndex(,'B',I) → 2
@[I = 'B'] → 2
PositionInIndex(,'D',I) → 0
@[I = 'D'] → 0
```

This is the result 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 $\mathbf{n}^{th}$ slice of the value of expression **e** over index **i**.

Examples

```
Index Car_type :=
```

VW	Honda	BMW

```
@Car_type →
Car_type ▶
```

	VW	Honda	BMW
	1	2	3

```
@[Car_type='Honda']→ 2
Car_type[@Car_type=2] → 'Honda'
```

More examples and tips

```
Index Time:
```

0	1	2	3	4

```
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 this operator to re-index an array by another index having the same length but different elements. For example, suppose **Revenue** is indexed by **Time**, this following gives the same array but indexed by **Years**:

```
Revenue[@Time=@Years]
```

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 **x1**) are used. **Area()** computes the total integral across **i**, returning a value with one less dimension than **r**. Compare **Area()** to **Integrate()** (page 193).

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 might 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 **x** is not indexed by **i**, **Cumulate(x, i)** operates as if **x** were indexed by **i**, but constant across **i**. Using this, a convenient trick for numbering the elements of an index is to use **Cumulate(1, i)**.

Library Array

Example

```
Cumulate(Cost_in_time, Time) →
Mpg ▼, Time ▶ , Car_type = VW
```

	0	1	2	3	4
26	2185	4479	6888	9417	12.07K
30	2810	5761	8859	12.11K	15.53K
35	3435	7042	10.83K	14.8K	18.98K

```
Cumulate(1,Car_type) →
Years ▶
```

	VW	Honda	BMW
	1	2	3

See “Example variables” on page 182 for example array variables used here and below.

Uncumulate(x, i, *firstElement*)

Uncumulate(x, i) returns an array whose first element (along **i**) is the first element of **x**, and each other element is the difference between the corresponding element of **x** and the previous element of **x**. **Uncumulate(x**, **i**, **firstElement)** returns an array with the first element along **i** equal to **firstElement**, and each other element equal to the difference between the corresponding element of **x** and the previous element of **x**.

Uncumulate(x, i) is the inverse of **Cumulate(x, i)**. **Uncumulate(x, i, 0)** is similar to a discrete differential operator.

Library Array

Example

```
Uncumulate(Cost_in_time, Time) →
Mpg ▼, Time ▶, Car_type = VW
```

	0	1	2	3	4
26	2185	109	115	120	127
30	2810	141	147	155	163
35	3435	172	180	189	199

```
Uncumulate(Cost_in_time, Time,0) →
Mpg ▼, Time ▶, Car_type = VW
```

	0	1	2	3	4
26	0	109	115	120	127
30	0	141	147	155	163
35	0	172	180	189	199

See “Example variables” on page 182 for example array variables used here and below.

Cumproduct(x, i)

Returns an array with each element being the product of all of the elements of **x** along dimension **i** up to, and including, the corresponding element of **x**.

Library Array

Example

```
Cumproduct(Cost_in_time, Time) →
Mpg ▼, Time ▶, Car_type = VW
```

	0	1	2	3	4
26	2185	5.012M	12.07G	30.54T	81.11Q
30	2810	8.292M	25.69G	83.57T	285.5Q
35	3435	12.39M	46.92G	186.6T	778.9Q

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 (or N) values are equal, they receive the same rank and the next higher value receives a rank 2 (or N) higher. You can use an optional parameter, **Type**, to control which rank is assigned to equal values. By default, the lowest rank is used, equivalent to **Rank(x,i,Type:-1)**. Alternatively, **Rank(x,i,Type:0)** uses the mid-rank and **Rank(x,i,Type:1)** uses the upper-rank.

Library Array

Examples Basic example:

```
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

See "Example variables" on page 182 for example array variables used here and below.

Optional **Type** parameter example:

```
Index I := 1..7
Index RankType := [-1,0,1]
Variable A :=
I ▶
```

	1	2	3	4	5	6	7
	10	4	9	4	4	1	4

Rank(A,I,Type:RankType) →
Rank_type ▼, I ▶

	1	2	3	4	5	6	7	
-1	7	2	6	2	2	1	2	Lowest rank for duplicates, 2
0	7	3.5	6	3.5	3.5	1	3.5	Mid rank for duplicates, 3.5
1	7	5	6	5	5	1	5	Upper rank for duplicates, 5

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()** (page 190).

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

See "Example variables" on page 182 for example array variables used here and below.

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 might be necessary to convert multi-dimensional data into two-dimensional tables before transferring.

MDArrayToTable(a, i, l)

Transforms a multi-dimensional array, **a**, into a two-dimensional array *(*i.e., a table) indexed by **i** and **l**. The result contains one row along **i** for each element of **a**. **l** must contain a list of names of the indexes of **a**, followed by one final element. All elements of **l** must be text values. The column corresponding to the final element of **l** contains the cell value. If **l** does not contain all the indexes of **a**, array abstraction creates a set of tables indexed by the dimensions not listed in **l**.

Before using **MDArrayToTable()**, you must define the index **i** with the appropriate number of elements. The number of elements in **i** can be either `size(a)`, or the number of non-zero elements of **a** (in which case the resulting table contains 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

See "Example variables" on page 182 for example array variables used here and below.

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=size(cols))`**. In this case, the elements of **cols** must be index identifiers or index names.

The first *n-1* columns of **t** specify the coordinates of a cell in the result. The final column of **t** specifies the value for the indicated cell.

Before using **MDTable()**, you must define all of the indexes for the result. Each index *must* include all values that occur in the corresponding column of **t** or an error results. The **Unique()** function is useful for defining the necessary indexes.

It is possible that two or more rows of **t** specify identical coordinates. In this case, a *conglomeration function* is used combine the values for the given cell. The **conglomFn** parameter is a text value specifying which conglomeration function is to be used. Possible values are **`"sum"`** (default), **`"min"`**, **`"max"`**, **`"average"`**, and **`"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')` →

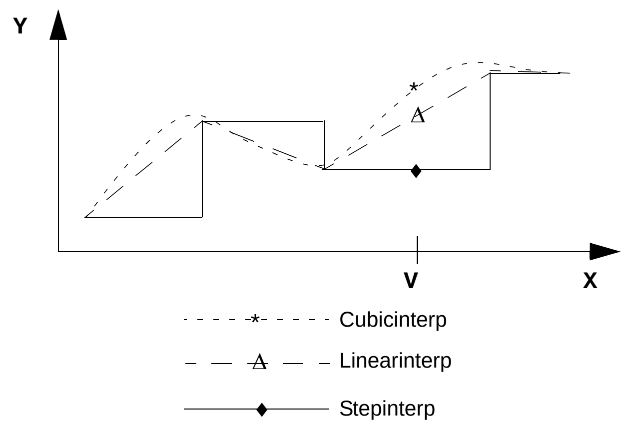
`Car_type ▼ , Mpg ▶`

	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`**. It uses **`average`** as the conglomeration function to combine these.

Interpolation functions

These three functions interpolate across arrays. Given arrays **y** and **x** with a common index **i**, these functions interpolate a value for **y** corresponding to value **v** along the **x** axis.



LinearInterp() and **CubicInterp()** use these 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

Stepinterp(x, y, v, *i*)

Returns the element or slice of array **y** for which **v** has the smallest value less than or equal to **x**. **x** and **y** must both be indexed by **i**, and **x** must be increasing along index **i**. If **v** is greater than all values of **x**, it returns the element of **y** for which **x** has the largest value.

When an optional parameter, **LeftLookup**, is specified as True, it returns the element or slice of **y** corresponding to the *largest* value in **x** that is less than or equal to **v**.

If **v** is an atom (scalar value), the result is an array indexed by all indexes of **a** except **x**'s index. If **v** is an array, the result is also indexed by the indexes of **v**.

If the first parameter **x** is an index of **y**, the fourth parameter is optional. **Stepinterp(x, y, v)** is similar to **y[x=v]** except that **y[x=v]** selects based on **v** being *equal* to **x**, while **Stepinterp(x, y, v)** selects based on **v** 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 **v** is a list of two values:

`Stepinterp(MPG, Cost, [28,33], MPG)` →

	VW	Honda	BMW
28	1705	2330	2955
33	1585	2210	2935

Linearinterp(x, y, v, *i*)

Returns linearly interpolated values of **v**, given **y** representing an arbitrary piecewise linear function. **x** and **y** must both be indexed by **i**, and **x** must be increasing along **i**. **y** is an array of the corresponding output values for the function (not necessarily increasing and might be more than one dimension). **v** might be probabilistic and/or an array.

For each value of **v**, **Linearinterp()** finds the nearest two values from **x** and interpolates linearly between the corresponding values from **y**. If **v** is less than the minimum value in **x**, it returns the first value in **y**. If **v** is greater than the maximum value in **x**, it returns the last value in **y**.

Library Special

Example `Linearinterp(Index_b, Array_a, 1.5, Index_b)` →
`Index_a` ▶

	a	b	c
	2	-2.5	2.5

Cubicinterp(x, y, v, *i*)

Returns the natural cubic spline interpolated values of **y** along **x**, interpolating for values of **v**. **x** and **y** must both be indexed by **i**, and **x** must be increasing along **i**.

For each value of **v**, **Cubicinterp()** finds the nearest values from **x**, and using a natural cubic spline between the corresponding values of **y**, computes the interpolated value. If **v** is less than the minimum value in **x**, it returns the first value in **y**; if **v** is greater than the maximum value in **x**, it returns the last value for **y**.

Library Special

Example `Cubicinterp(Index_b, Array_a, 1.5, Index_b)` →
`Index_a` ▶

	a	b	c
	0.6875	-2.875	2.219

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 These examples use these variables:

`Index Years :=`

1985	1986	1987	1988

Index More_years:

1989	1990	1991

Index All_years := Concat(Years, More_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(Car_prices, More_prices, Years, More_years, All_years) →
All_years ▼, Car_type ▶

	VW	Honda	BMW
1985	8000	12K	18K
1986	9000	13K	20K
1987	9500	14K	21K
1988	10K	14.5K	22K
1989	11K	15K	23.5K
1990	12K	15.5K	25K
1991	12.5K	16.5K	27K

See "Example variables" on page 182 for example array variables used here and below.

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 the **ResultIndex()** function, which must be an index with **size(i) * size(j)** elements.

Library Concatenation

To use this function, you must add the library to your model.

IndexNames(a)

Returns a list of the identifiers of the indexes of the array **a** as text values.

Library Array

Example **IndexNames(Car_prices) → ['Car_type','Years']**

IndexesOf(a: Array)

Returns a list of the indexes of the array **a** as handles (see "Handles to objects" on page 344).

It is similar to **IndexNames()**, except that it returns handles instead of identifiers as text values. It is possible for an array to have more than one local index having identical names. This is not recommended, but where it occurs, the index handles returned by **IndexesOf()** are unambiguous.

Library Array

Example **IndexesOf(Car_prices) → [Car_type, Years]**

IndexValue(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

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 atoms (elementary cells) in array **u**. The size of an atom is 1. The size of an empty list is 0.

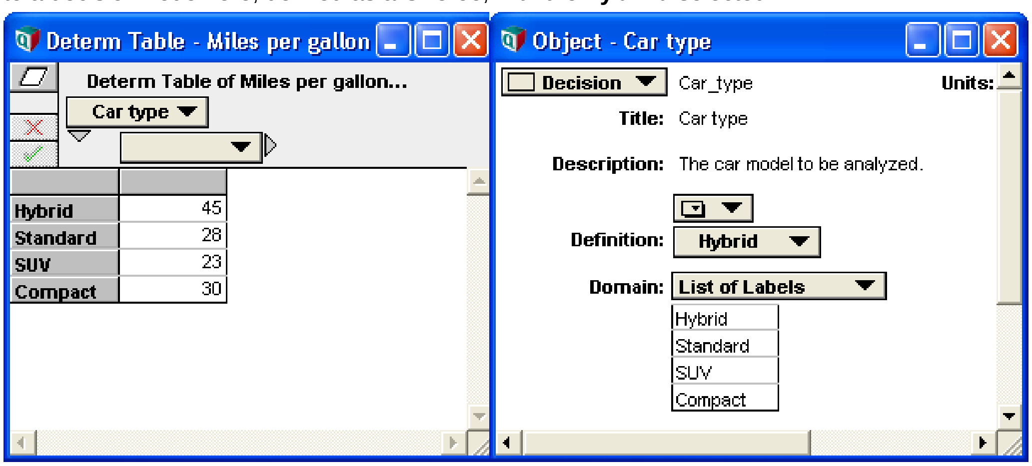
Library Array Functions

Examples
```
Size(Years) → 4
Size(Car_prices) → 12
Size(10) → 1
Size([]) → 0
```

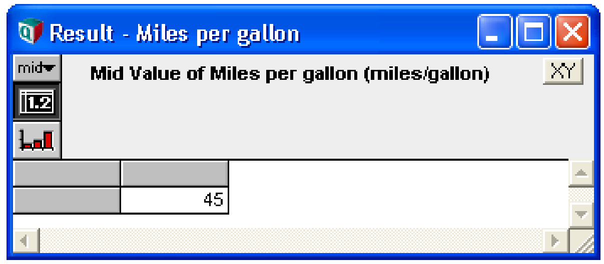
DetermTable: Deterministic tables

A **DetermTable** provides an input view like that of an edit table (see page 171), allowing you to specify values or expressions in each cell for all index combinations; however, unlike a table, the evaluation of a determtable conditionally returns only selected values from the table. It is called a determtable because it acts as a deterministic function of one or more discrete-valued variables. You can conceptualize a determtable as a multi-dimensional generalization of a **`select-case`** statement found in many programming languages, or as a value that varies with the path down a decision tree.

The following shows the edit view of a determ table, in which you can enter a different miles per gallon for each car type. **`Car_type`** has been changed from being an index in previous examples to a decision node here, defined as a **Choice**, with the **`Hybrid`** selected.



When `Miles_per_gallon` is evaluated, its result contains only the miles per gallon for the selected car type.



In comparison, the result of evaluating a straight table would include all values for all car types.

DetermTable inputs

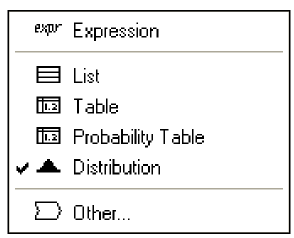
The dimensions of a determtable may be a combination of normal indexes and discrete variables. Each discrete variable used must have a domain that explicitly contains all possible values, and it is these values that are used for the dimension in the determtable edit view. The selection occurs over the discrete variables, so that **DetermTable()** behaves differently from **Table()** only when at least one of the dimensions is a discrete variable. The definition of each discrete variable specifies which value from its domain is selected.

When you define a discrete variable to serve as an input to **`DetermTable()`**, it is convenient to use a choice menu (see “Creating a choice menu” on page 121) with the index for the **Choice()** function set to **`Self`**. You must then set the domain attribute to either *List*, *List of Label*, or *Index*. The *List* and *List of Labels* options allow you to exist all possible values explicitly. An *Index* domain pulls the the list of possible values from a separate index object that already contains the list of possible values.

Creating a DetermTable

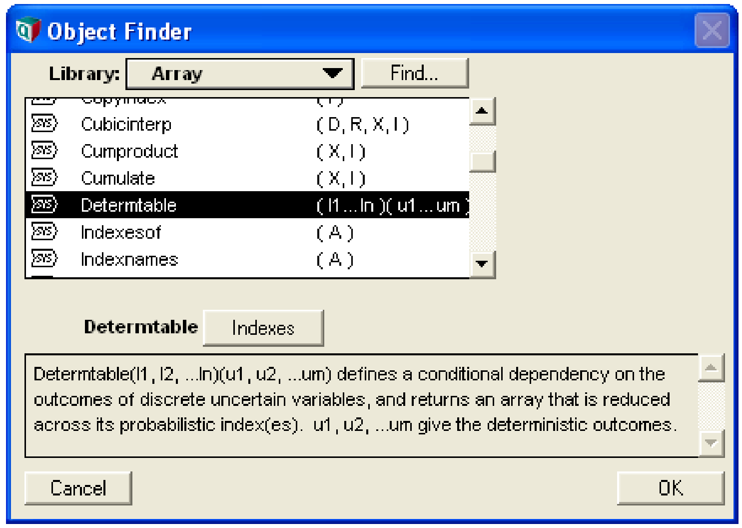
To define a variable as a determtable:

1. Decide on the inputs — the discrete conditioning variables.
2. Press the ***expr*** menu above the definition field and select **Other...**.

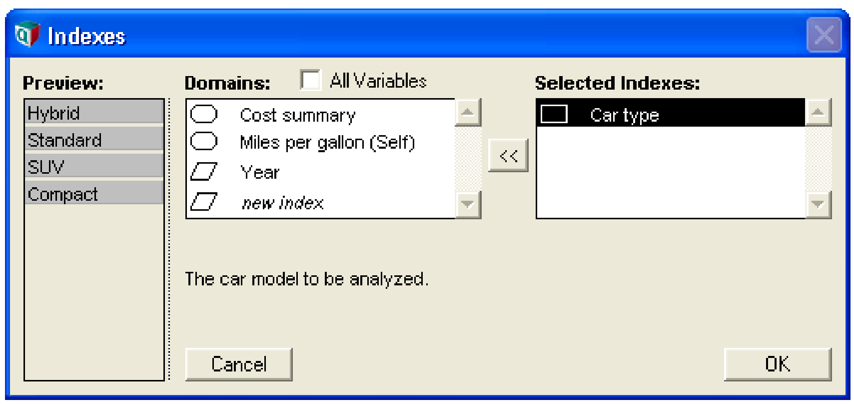


Analytica opens the **Object Finder dialog** (page 112).

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 variable(s).



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:

```
Determtable(i1, i2, … in) (r1, r2, r3, … rm)
```

This describes an *n*-dimensional conditional deterministic table, indexed by the indexes and discrete conditioning variables **i1, i2, … in**. The last index, **in**, is the innermost index, varying the most rapidly. **r1, r2, … rm** are the outcomes in the array.

Converting a Table to a DetermTable

To convert an existing table to a determTable, view either the Object Window or Attribute pane for the variable and use the pulldown to change the definition type to *Other...*. Answer *Yes* when asked to replace the current definition, and the **Object Finder dialog** (page 112) appears. From the *Array* library select **DetermTable** and press **OK**.

An alternative way to convert a table to a determTable is to view the table definition in expression mode and change the first word **Table** to **DetermTable**.

Use in Parametric Analysis

A parametric analysis varies one or more model inputs across several hypothetical values, computing results for each combination of inputs. Array abstraction makes it very easy to conduct parametric analyses in Analytica; however, the computational complexity and memory requirements scale multiplicatively as you vary more and more input variables simultaneously, resulting in practicality limits on the number of inputs that can be simultaneously varied.

Determtables provide a useful tool for coping with the complexity / dimensionality tradeoff. You can select a subset of input variables to vary parametrically, examine your model outputs as these vary, then re-run your model after selecting a different subset of inputs to vary. Using Choice menus for the inputs, and determTables for any tables based on those input dimensions, makes it possible to change your parametric inputs rapidly to quickly explore relationships elucidated by your model. Obtaining this agility is often a simple matter of converting existing tables to determ-Tables.

Subscript equivalence

You can achieve the equivalent functionality of **DetermTable()** without using the **DetermTable()** function, but **DetermTable** is a nice convenience that saves having an extra node in your model. As an alternative to a determtable, you can create a standard edit table in a variable, **A**, and then obtain the desired slice in a second variable, **B**, by defining it as `A[u=u,v=v]`, where **u** and **v** are the discrete conditioning variables. This works because **u** and **v** are both self-indexed (with the possible values being the self-index values) and also have their own value (the selected value).

SubTable

The purpose of **SubTable** is to provide the user an alternative editable view of part of an edit table. If a variable **a** is defined as an edit table, a variable **b** defined as **SubTable(a[i=x])** lets the user use u to view and edit a subarray of **a**, for which index **i** of **a** has value **x**. Any change you make to cells of **b** is reflected in **a**, and vice versa. The actual values are stored in edit table **a.**

SubTable(a[i = x])

A subtable can also show subarrays of **a** in a different order, if **x** is an array containing some or all values of **i** in a different sequence. **b** can also use different number formats.

A subtable also works if **a** is defined using any editable table functions, including edit table (**table**), probability table (**probtable**), deterministic table (**determtable**), or even another subtable.

SubTable() must be the main expression in the definition of a variable. It cannot be a subexpression or inside a function. Its parameter must be a slice or subscript operator. For example, in the simplest form:

```
SubTable(a[i=x])
```

where **x** is an element of index **i** and **x** is a value of **i**. Many other variations are also useful including:

```
SubTable(a[i=x])
SubTable(a[i=x, j=y])
SubTable(a[i=b])
SubTable(a[@i=c])
```

If the subarray returned by **Subtable()** is an atom (i.e., a single value with no indexes), you can edit it in a table view, or, if you define an input node for it, directly as an input field.

Matrix functions

A ***matrix*** is a square array, that is an array that has two dimensions of the same length. It can also have other dimensions, not necessarily of the same length. Matrix functions perform a variety of operations of matrices and other arrays.

Standard mathematical notation for matrix algebra, and most computer languages that work with matrices, distinguish rows and columns. In Analytica, rows and columns are not basic to an array: They are just ways you can choose to display an array in a table. Instead, it uses a named index to label each dimension. So, when using a matrix function, like any Analytica function that work with arrays, you specify the dimensions you want it to operate over by naming the index(es) as a parameter(s). For example:

```
Transpose(X, I, J)
```

This exchanges indexes **I** and **J** for matrix **X**. You don't need to remember or worry about which dimension is the rows and which the columns. **X** can also have other indexes, and the function generalizes appropriately.

Dot product of two matrices

The dot product (i.e., matrix multiplication) of **MatrixA** and **MatrixB** is equal to:

```
Sum(MatrixA * MatrixB, i)
```

where **i** is the common index.

Example

```
Variable MatrixA:
j ▼, i ▶
```

	1	2	3
a	4	1	2
b	2	5	3
c	3	2	7

```
Variable 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, aRow, aCol, b, bRow, bCol)

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 can 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                    x                    D
index1 ▼, index2 ▶  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** (page 203).

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** exchanging dimensions **i** and **j**, which must be indexes of the same size.

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 `v * C * v > 0`, for all vectors `v`.)

Cholesky decomposition computes a lower diagonal matrix **L** such that **L** * **L'** = **C**, where **L'** is the transpose of **L**.

Library Matrix

Example `Matrix`

L ▼, M ▶

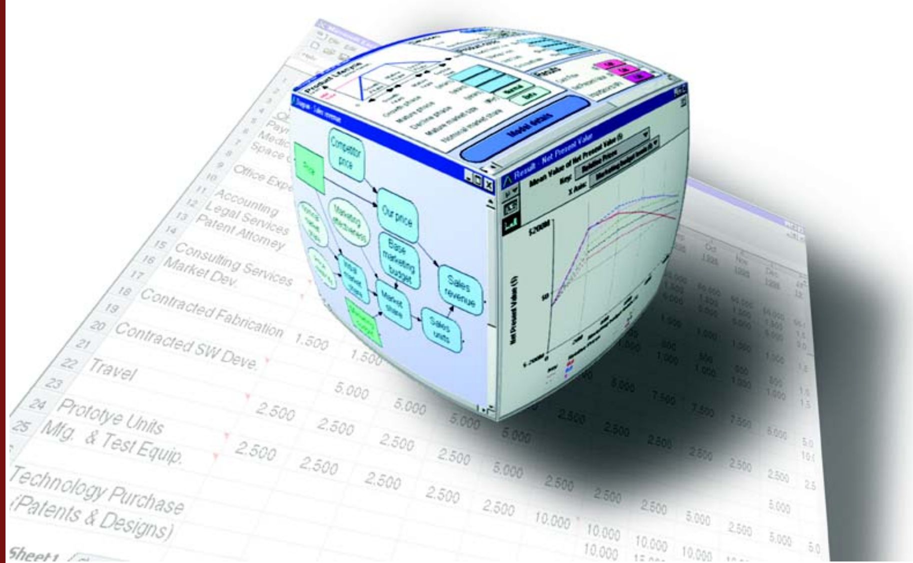
	1	2	3	4	5
1	6	[illegible]	6	3	1
2	2	4	3	1	3
3	6	3	[illegible]	3	4
4	3	1	3	8	4
5	1	3	4	4	7

SingularValueDecomp() requires its main parameter to be

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 page 206)
- Date functions for working with date numbers (see page 207)
- Advanced math functions (see page 209)
- Financial functions (see page 210)
- A library of extra financial functions, including functions for valuing options (see page 214)
- Advanced probability functions (see page 217)



Text functions

These functions work with **text values** (page 133) (sometimes known as *strings*), available in the built-in Text library.

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 multiline 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(substr, text, *start*,caseInsensitive) Returns the position of the first occurrence of the text **substr** within the text **text**, as the number of characters to the first character of **text**. If **substr** does not occur in **text**, it returns 0. **FindInText()** is case-sensitive unless the optional parameter **caseInsensitive** is true. For example:

```
Variable People := 'Amy, Betty, Carla'
FindInText('Amy', People) → 1
FindInText('amy', People) → 0
FindInText('amy', People, caseInsensitive:true) → 1
FindInText('Betty', People) → 6
FindInText('Fred', People) → 0
```

The optional third parameter, **start**, specifies the position to start searching at, for example, if you want to find a second occurrence of **substr** after you have found the first one.

```
FindInText('i','Supercalifragilisticexpialidocious') → 9
FindInText('i','Supercalifragilisticexpialidocious',10) → 14
```

TextTrim(t, *leftOnly, rightOnly, trimChars*) Removes leading and trailing spaces from the text. To remove characters other than spaces, specify the characters to remove in the optional **trimChars** parameter.

```
TextTrim(' Hello World ') → 'Hello World'
TextTrim(' Hello World ',leftOnly:True) → 'Hello World '
TextTrim(' Hello World ',rightOnly:True) → ' Hello World'
TextTrim(' [One,Two,Three] ',trimChars:' []') → 'One,Two,Three'
```

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 page 337 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*) 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 and time numbers*** — that is, the integer portion is number of days since the *date origin,* usually Jan 1, 1904, and the fractional portion is the fraction of a day elapsed since midnight. See “Date numbers and the date origin” on page 86. A date number displays as a date if you select a date format using the **Number format** dialog from the **Result** menu.

MakeDate() generates a date number from the year, month, and day. **DatePart** extracts the year, month, day, or other information from a date number. **DateAdd()** adds a number of days, weeks, months, or years to a date. **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

MakeTime(h, m, s)

Gives the fraction of a day elapsed since midnight for the given hour, minute and second. The hour, **h**, should be between 0 and 23 inclusive. Minutes and seconds should be between 0 and 59 inclusive.

Examples `MakeTime(12, 0, 0)` → `0.5`
`MakeTime(15, 30, 0)` → `0.6458 { 3:30:00 pm }`

Library Special Functions

DatePart(date, part)

Given a date-time value **date**, it returns the year, month, day, hour, minute, or seconds 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).
- **`H`** gives the hour on a 24-hour clock (0 to 23).
- **`h`** gives the hour on a 12-hour close (1 to 12).
- **`m`** gives the minutes (0 to 59).
- **`s`** gives the seconds (0 to 59.99).

Other date options for **part** are: YY→`'06'`, MM→`'01'`, **MMM** → `'Jan'`, **MMMM** → `'January'`, DD→`'09'`, **ddd** → `'1st'`, **dddd** → `'first'`, **Dddd** → `'First'`, **www** → `'Mon'`, **wwww** → `'Monday'`, and **q** → 1 to 4 for number of quarter of the year.

Other time options for **part** are: **HH**→`'15'`, **hh**→`'03'`, **mm**→`'05'`, and **ss**→`'00'`.

`DatePart` can also weeks or weekdays elapsed since the date origin or in the current year.

- **`wd`** (or **`wd+`**) gives the number of weekdays since the date origin including the indicated day.
- **`wd-`** gives the number of weekdays since the date origin not including the indicated day.
- **`#d`** gives the day number in the current year
- **`#w`** gives the week number in the current year (the week starting on Sunday)
- **`#wm`** gives the week number in the current year (the week starting on Monday)

The **`#w`** and **`#wm`** options consider the week containing Jan 1 to be week 1. Options **`e#w`** and **`e#wm`** return the European standard in which **`week1`** is the first week containing at least 3 days.

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)
```

This computes the number of weekdays between two dates, including both endpoints:

```
DatePart(date2,'wd+') - DatePart(date1,'wd-')
```

Library Special Functions

DateAdd(date, n, unit)

Given a date value **date**, it returns a date value offset by **n** years, months, days, weekdays, hours, minutes or seconds, according to whether **unit** is **Y**, **M**, **D**, **WD**, **h**, **m**, or **s**. 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(*withTime, utc*)

Returns the current date (or optionally date and time) as a date number — the number of days since the date origin, usually Jan 1, 1904. Unlike other functions, it gives a different value depending on what day (and time) it is evaluated. It is most often called with no parameters, **Today( )**, in which case the result is an integer representing the date in your local time zone. Including the optional parameter, **Today(withTime:True)** returns the current time of day in the fractional part. **Today(withTime:true,utc:True)** returns the coordinated universal date-time rather than the local date-time.

Since variables usually cache (retain) their value after computing it, the date could become out of date if the Analytica session extends over midnight. But, it will be correct again when you restart the model.

Library Special Functions

Advanced math functions

These functions can be accessed under the **Definition** menu **Advanced Math** command, or in the **Object Finder** dialog, Advanced Math library. Functions in this section are generally for more advanced mathematical users than those found in "Math functions" on page 136. There are additional advanced math functions covered in "Importance weighting" on page 257.

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
```

BesselJ(x,n), BesselY(x,n), BesselI(x,n), BesselK(x,n) Bessel functions of the first kind (J), second kind (Y), and modified Bessel functions of the first (I) and second (K) kinds. These are used in engineering applications involving harmonics in cylindrical coordinates. The second parameter, **n**, is the order of the Bessel function and can be integer or fractional. When **n** is non-integer, **x** must be non-negative. These functions are not exposed on the **Advanced Math** library menu.

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(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, 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 *per period*. For example, if periods are months, the rate should be adjusted to the monthly rate, not the annual rate (e.g., `8%/12`, or `1.08^(1/12)-1` with monthly compounding).
nPer	Number of periods in the lifetime of an annuity.
per	The period (between 1 and **nPer**) being computed.
pv	The present value of the annuity. For example, for a loan this is the loan amount (positive if you receive the loan, negative if you are the lender).
fv	The future value of the annuity. This is the remaining value of the annuity after the final payment. In the case of a loan, for example, this is the balloon payment at the end (positive if you are the lender, negative if you pay the balloon amount). This parameter is usually optional with a default value of zero.
pmt	The total payment per period (interest + principal). If you receive payments, this is positive. If you make payments, this is negative.
type	Indicates whether payments are due at the beginning or end of each period.
True	Payments are due at the beginning of each period, with the first payment due immediately.
False	(default) Payments are due at the end of each period.

Cumipmt(rate, nPer, pv*, startPeriod, endPeriod, type*)

Returns the cumulative interest paid on an annuity between, and including, **startPeriod** (shown as **sp** in equation below) and **endPeriod** (shown as **ep** in equation below). The annuity is assumed to have a constant interest rate and periodic payments. This is equal to:

$$\sum_{n=sp}^{ep} Ipmt(rate, n, nPer, Pv, 0, Type)$$

Example Interest payments during the first year on a $100,000 loan at 8% is:

CumIPmt(8%/12, 360, 100K, 1, 12) → -7,969.81

The result is negative since these are payments.

Cumprinc(rate, nPer, pv*, 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:

$$\sum_{n=sp}^{ep} PPmt(Rate, n, Nper, Pv, 0, Type)$$

Example The total principal paid during the first year on a $100,000 loan at 8% is:

CumPrinc(8%/12, 360, 100K, 1, 12) → -835.36

The result is negative since these are payments.

Fv(rate, nPer, pmt*, pv, type*)

Returns the future value of an annuity investment with constant periodic payments and fixed interest rate. The result is positive if you receive money at the end of the annuity's lifetime, and negative if you must make a payment at the end of the annuity's lifetime.

Examples You invest $1000 in an annuity that pays 6% annual interest, compounded monthly (0.5% per month), that pays out $50 at the end of each month for 12 months, and then refunds whatever is left after 12 months. The amount refunded is:

Fv(0.5%, 12, 50, -1000) → $444.90

You borrow $50,000 at a fixed annual rate of 12% (1% per month). You make monthly payments of $550 for 15 years, and then pay off the remaining balance in a single balloon payment. That final balloon payment is (the negative is because it is a payment for you):

-Fv(1%, 15*12, -550, 50000) → $25,020.99

You open a fixed-rate bank account that pays 0.5% per month in interest. At the beginning of each month (including when you open the account) you deposit $100. The amount in the account at the end of the each of the first three years is:

Fv(0.5%, [12, 24, 36], -100, 0, True) →
[$1239.72, $2555.91, $3953.28]

Ipmt(rate, per, nPer, pv*, fv, type*)

Returns the interest portion of a payment on an annuity, assuming constant period payments and fixed interest rate.

Example The interest you pay in the 24th month on a 30-year fixed $100K loan at 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()** has no solution and returns **`NaN`** (not a number). If a cash flow changes sign more than once, **Irr()** might have multiple solutions, and returns 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** is usually 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 lasts 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:

`Npv(5%, Earnings, Time)` → $865,947.76

Pmt(rate, nPer, pv, *fv, type*)

Returns the total payment per period (interest + principal) for an annuity with constant periodic payments and fixed interest rate.

Example You obtain a 30-year fixed mortgage at 8%/12 per month for $100K. Your monthly payment is (note that the result of **Pmt()** is negative since this is a payment for you):

`-Pmt(8%/12, 30*12, 100K)` → $733.76

Ppmt(rate, per, nPer, pv*, fv, type*)

Returns the principal portion of a payment on an annuity with constant period payments and fixed interest rate.

Example You have a 30-year fixed $100K loan at a rate of 8%/12 monthly. On your 24th payment, the amount of your payment that goes towards principal is (note that the result of **PPmt()** is negative since this is a payment for you):

`-PPmt(8%/12, 24, 12*30, 100K) → $78.18`

Pv(rate, nPer, pmt, fv, type)

Returns the present value of an annuity. The annuity is assumed to have constant periodic payments to you of **pmt** per period for **nPer** periods, with a return of **rate** per period.

Example To receive $100 per month from an annuity that returns 6%/12 per month for the next 10 years, you would need to invest (note that the result from **Pv()** is negative since you are paying to make the investment):

`-Pv(6%/12, 10*12, 100) → $9,007.35`

Rate(nPer, pmt, pv*, fv, type, guess*)

Returns the interest rate (per period) for an annuity. The value returned is the interest rate that results in equal payments of **pmt** per period over the **nPer** periods of the annuity.

In general, **Rate()** can have zero or multiple solutions. The implementation uses an interactive search algorithm. The optional **guess** can be provided as a starting point for the search, which usually results 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()** can have multiple solutions, but returns only the first solution found. The optional parameter, **guess**, can be provided as a starting point for the iterative search, and **Xirr()** generally finds 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:

$$BetaFn(a, b) = \int_0^1 x^{a-1}(1-x)^{b-1}dx$$

BetaI(x, a, b) The incomplete beta function, defined as:

$$BetaI(x, a, b) = \frac{1}{Beta(a, b)}\int_0^X x^{a-1}(1-x)^{b-1}dx$$

The incomplete beta function is equal to the cumulative probability of the beta distribution at **x**. It is useful in a number of mathematical and statistical applications.

The cumulative binomial distribution, defined as the probability that an event with probability *p* occurs *k* or more times in *n* trials, is given by:

$$Pr = BetaI(p, k, n-k+1)$$

The student's distribution with *n* degrees of freedom, used to test whether two observed distributions have the same mean, is readily available from the beta distribution as:

$$Student(x|n) = 1 - BetaI(n/(n+x^2), n/2, 1/2)$$

The F-distribution, used to test whether two observed samples with n_1 and n_2 degrees of freedom have the same variance, is readily obtained from `BetaI` as:

$$F(x, n_1, n_2) = BetaI(n_2/(n_1x+n_2))$$

BetaIInv(p, a, b) The inverse of the incomplete beta function. Returns the value **X** such that **BetaI(x, a, b)=p**.

Combinations(k, n) "**n** choose **k**." The number of unique ways that **k** items can be chosen from a set of **n** elements (without replacement and ignoring the order).

`Combinations(2, 4)` → `6`

They are: {1,2}, {1,3}, {1,4}, {2,3}, {2,4}, {3,4}

Permutations(k, n) The number of possible permutations of **k** items taken from a bucket of **n** items.

`Permutations(2, 4)` → `12`

They are: {1,2}, {1,3}, {1,4}, {2,1}, {2,3}, {2,4}, {3,1}, {3,2}, {3,4}, {4,1}, {4,2}, {4,3}

CumNormal(x, *mean*, *stddev*) Returns the cumulative probability:

$$p = Pr[x \le X]$$

for a normal distribution with a given mean and standard deviation. **mean** and **stddev** are optional and default to **mean** = 0, **stddev** = 1.

```
CumNormal(1) - CumNormal(-1) → .683
```

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:

$$p = Pr[x \le X]$$

mean and **stddev** are optional and default to **mean** = 0, **stddev** = 1.

Erf(x) The error function, defined as:

$$Erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

ErfInv(y) The inverse error function. Returns the value **X** such that **Erf(X)=y**.

```
ErfInv(Erf(2)) → 2
```

GammaFn(x) Returns the gamma function of **x**, defined as:

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$$

The gamma function grows very quickly. For example, when *n* is an integer, *GammaFn(n+1) = n!*. For this reason, it is often preferable to use the **LGamma()** function.

GammaI(x, a, *b*) Returns the incomplete gamma function, defined as:

$$GammaI(x, a, b) = \frac{1}{\Gamma(a)} \int_0^{x/b} e^{-t} t^{b-1} dt$$

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 $x \ge 0$.

The incomplete gamma function returns the cumulative area from zero to **x** under the gamma distribution.

The incomplete gamma function is useful in a number of mathematical and statistical contexts.

The cumulative Poisson distribution function, which encodes the probability that the number of Poisson random events (**x**) occurring will be less than *k* (where *k* is an integer) where the expected mean number is *a*, is given by (recall that parameter **b** is optional).

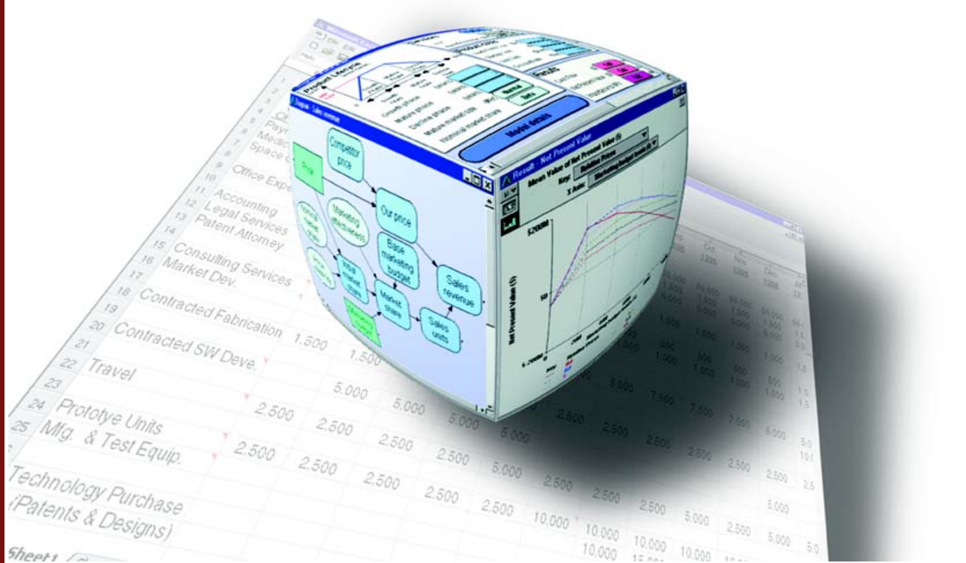
$$P(x < k) = GammaI(k, a)$$

GammaIInv(y, a, *b*) The inverse of the incomplete gamma function. Returns the value **x** such that **GammaI(x, a, b) = y**. **b** is optional and defaults to 1.

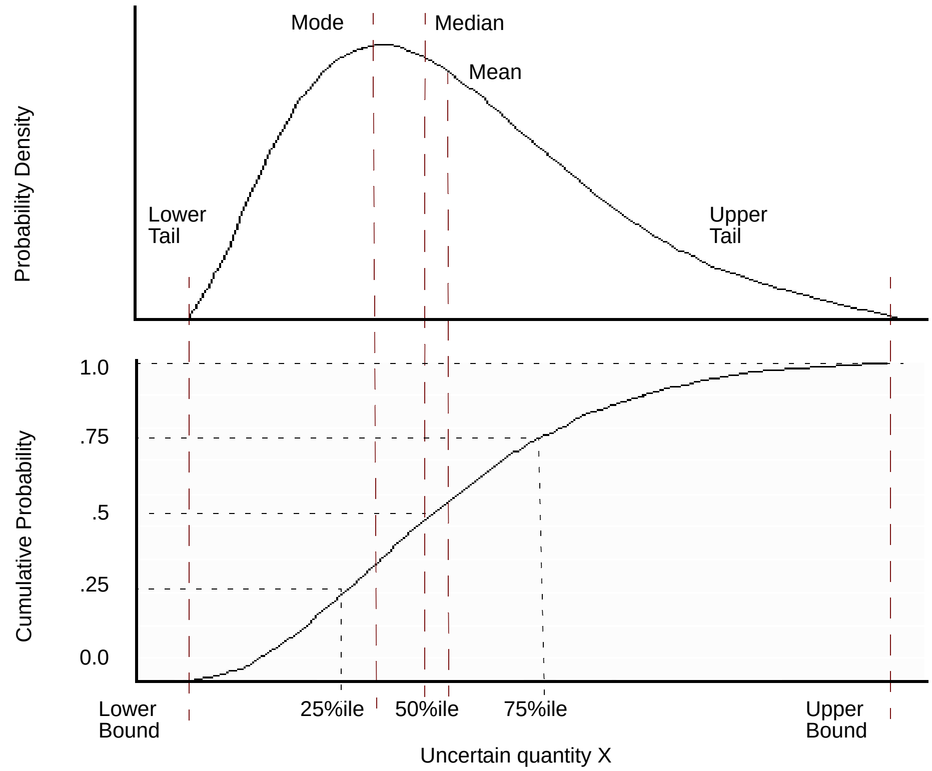
Chapter 14 *Expressing Uncertainty*

This chapter shows you how to:

- Choose a distribution
- Define a variable as a distribution
- Include a distribution in a definition
- 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 to 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 393 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 can base the distribution on available relevant data, on the judgment of a knowledgeable individual, or on some combination of data and judgment.

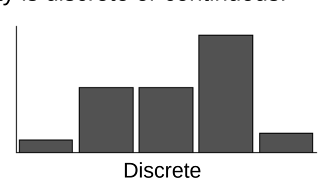
Answer the following questions about the uncertain quantity to select the most appropriate kind of distribution:

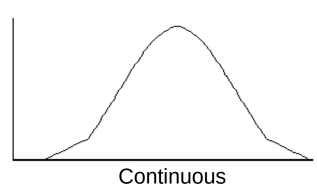
- Is it discrete or continuous?
- If continuous, is it bounded?
- Does it have one mode or more than one?
- Is it symmetric or skewed?
- Should you use a standard or a custom distribution?

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.

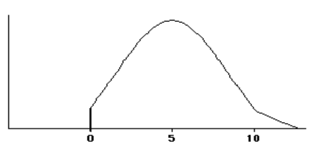
A ***continuous*** quantity can be represented by a real number, and has infinitely many possible values between any two values in its domain. Examples are the quantity of an air pollutant released during a given period of time, the distance in miles of a residence from a source of air pollution, and 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 might want to represent uncertainty about the number with a continuous probability distribution.

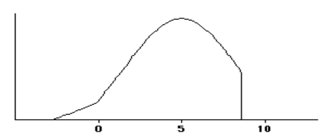
Conversely, it is often convenient to treat continuous quantities as discrete by partitioning the set of possible values into a small finite set of partitions. For example, instead of modeling human age by a continuous quantity between 0 and 120, it is often convenient to partition people into infants (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 maximum value?



Exact Lower Bounds



Exact Upper Bounds

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%.

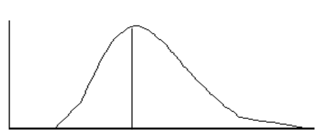
Most real world quantities have de facto bounds — that is, you can comfortably assert that there is zero probability that the quantity would be smaller than some lower bound, or larger than some upper bound, even though there is no precise way to determine the bound. For example, you can be sure that no human could weigh more than 5000 pounds; you might be less sure whether 500 pounds is an absolute upper bound.

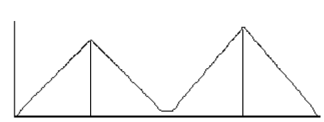
Many standard continuous probability distributions, such as the normal distribution, are unbounded. In other words, there is some probability that a normally distributed quantity is below any finite value, no matter how small, and above any finite value, no matter how large.

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 might 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.

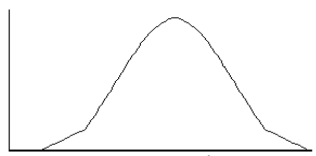

mode

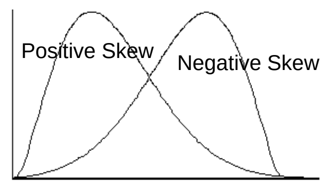

modes

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 might suggest that a particular standard distribution is appropriate. More often, however, there is no obvious standard distribution to apply.

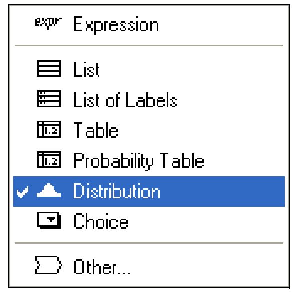
It is generally much faster to assess a standard distribution than a full custom distribution, because standard distributions have fewer parameters, typically from two to four. You should usually start by assigning a simple standard distribution to each uncertain quantity using a quick judgment based 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** (page 24) of the diagram with **Definition** (page 108) selected from the **Attribute** popup menu.

To define the distribution:

1. Click the ***expr*** menu above the definition field and select **Distribution**.

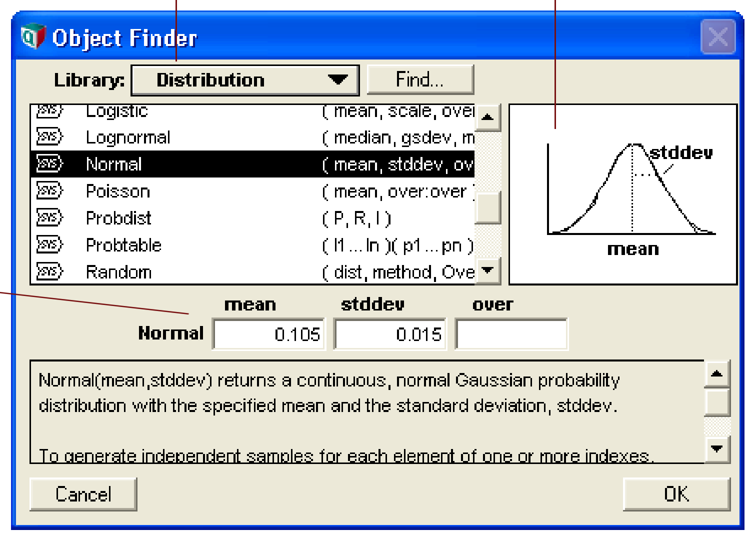


The **Object Finder** opens, showing the Distribution library.

Library popup menu: Distribution library is selected

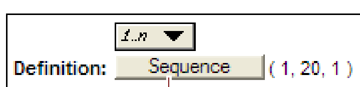
Example probability density, indicating parameters

Parameters to the distribution



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.



Button with the name of the distribution

Parameters of the distribution

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:

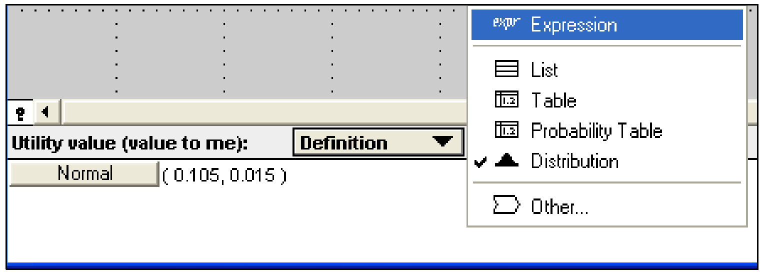
1. Set the cursor in the definition field and type in the distribution name and parameters, for example:

```
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 323).

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 ***expr*** 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** (page 33) — the mean, statistics, probability bands, probability density (or mass function), and cumulative distribution function. 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 might 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” on page 225).

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** (page 225).

Tip If you try to apply an **array-reducing function** (page 185) 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()** (page 266).

Example:

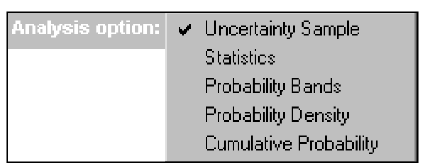
`Max(Sample(X), Run)` → **`0.8892`**

Uncertainty Setup dialog

Use the **Uncertainty Setup** dialog 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.

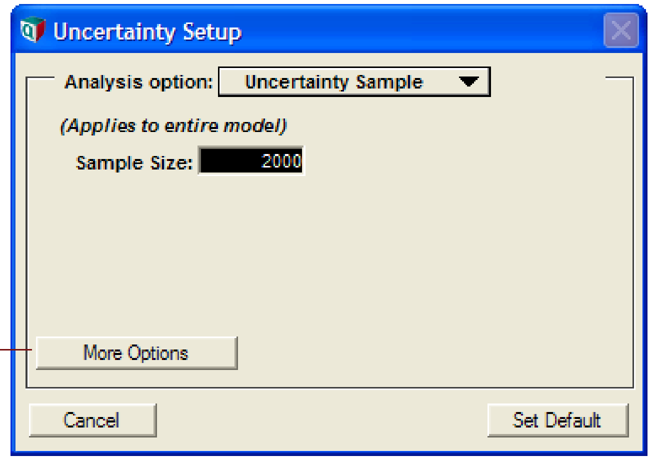
To open the **Uncertainty Setup** dialog, select **Uncertainty Options** from the **Result** menu or *Control+u*. To set values for a specific variable, select the variable before opening the dialog.

The five options for viewing and changing information in the **Uncertainty Setup** dialog can be accessed using the **Analysis option** popup menu.

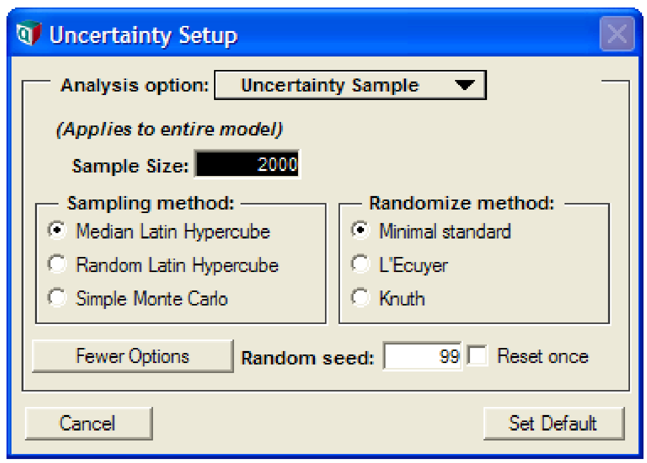


Uncertainty sample To change the sample size or sampling method for the model, select the **Uncertainty Sample** option from the **Analysis option** popup menu.

Press here to see additional uncertainty sample parameters.



The default dialog 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 “Appendix A: Selecting the Sample Size” on page 372 for guidelines on selecting a sample size. The sample size must be between 2 and 32,000. You can access this number in expressions in your models as the system variable **`Samplesize`**.

Sampling method The sampling method is used to determine how to generate a random sample of the specified sample size, **`m`**, for each uncertain quantity, **`X`**. Analytica provides three options:

- **Simple Monte Carlo**

  The simplest sampling method is known as Monte Carlo, named after the randomness prevalent in games of chance, such as at the famous casino in Monte Carlo. In this method, each of the **`m`** sample points for each uncertainty quantity, **`X`**, is generated at random from **`X`** with probability proportional to the probability density (or probability mass for discrete quantities) for **`X`**. Analytica uses the inverse cumulative method; it generates **`m`** uniform random values, $\mathbf{u}_i$ for **`i`**=1,2,...**`m`**, between 0 and 1, using the specified random number method (see below). It then uses the inverse of the cumulative probability distribution to generate the corresponding values of **`X`**,

  Xi where $P(x \le Xi) = u_i$ for i=1,2,...m.

  With the simple Monte Carlo method, each value of every random variable **`X`** in the model, including those computed from other random quantities, is a sample of **`m`** independent random values from the true probability distribution for **`X`**. You can therefore use standard statistical methods to estimate the accuracy of statistics, such as the estimated mean or fractiles of the distribution, as for example described in "Appendix A: Selecting the Sample Size" on page 372.

- **Median Latin hypercube (the default method)**

  With median Latin hypercube sampling, Analytica divides each uncertain quantity **X** into **m** equiprobable intervals, where **m** is the sample size. The sample points are the medians of the **m** intervals, that is, the fractiles

  Xi where $P(x \leq Xi) = (\mathbf{i}-0.5)/m$, for i=1,2,...m.

  These points are then randomly shuffled so that they are no longer in ascending order, to avoid nonrandom correlations among different quantities.

- **Random Latin hypercube**

  The random Latin hypercube method is similar to the median Latin hypercube method, except that instead of using the median of each of the **m** equiprobable intervals, Analytica samples at random from each interval. With random Latin hypercube sampling, each sample is a true random sample from the distribution. However, the samples are not totally independent.

Choosing a sampling method

The advantage of Latin hypercube methods is that they provide more even distributions of samples for each distribution than simple Monte Carlo sampling. Median Latin hypercube is still more evenly distributed than random Latin hypercube. If you display the PDF of a variable that is defined as a single continuous distribution, or is dependent on a single continuous uncertain variable, using median Latin hypercube sampling, the distribution usually looks fairly smooth even with a small sample size (such as 20), whereas the result using simple Monte Carlo looks 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 might 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:

```
X:= Uniform(1, Samplesize)
Y:= Sin(2*Pi*X)
```

This median Latin hypercube method gives 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

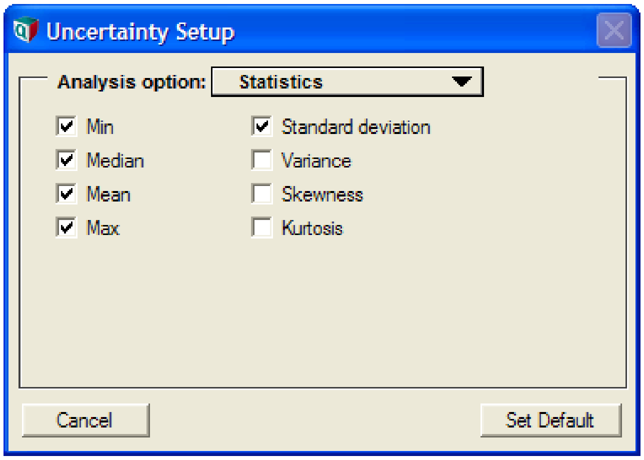
This value must be a number between 0 and 100,000,000 (10^8). The series of random numbers starts from this seed value when:

- A model is opened.

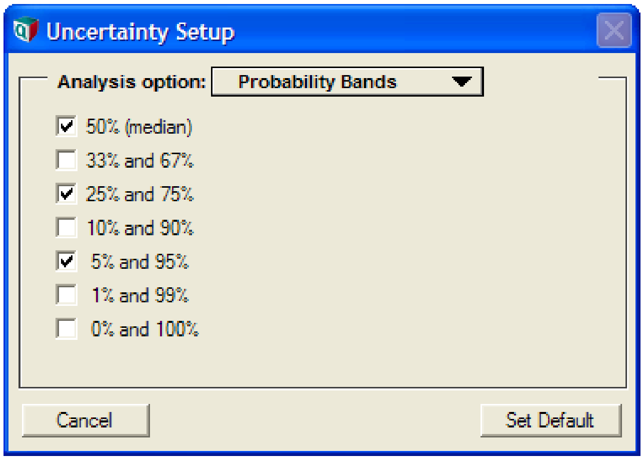
- The value in this field is changed.
- The *Reset once* box is checked, and the **Uncertainty Setup** dialog is closed by clicking the **Accept** or **Set Default** button.

Reset once Check the *Reset once* box to produce the exact same series of random numbers.

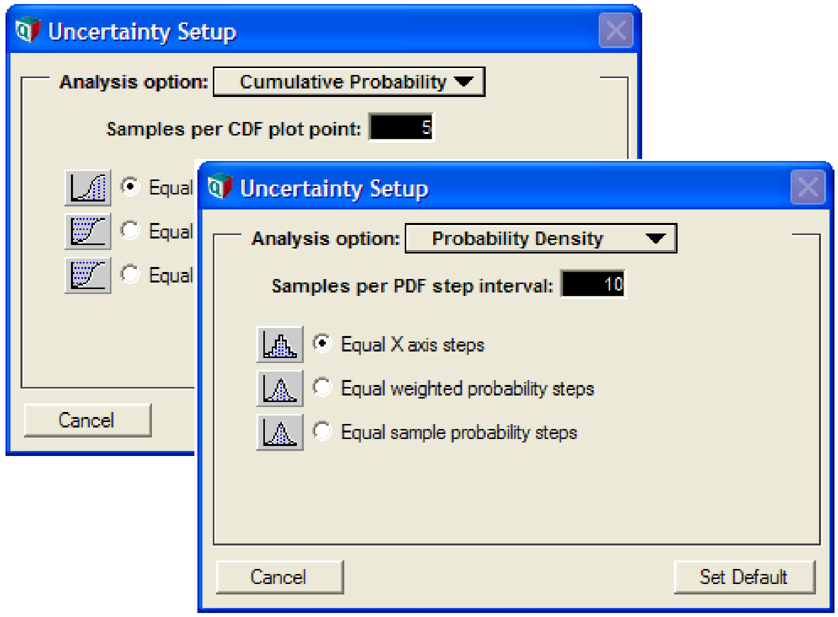
Statistics option To change the statistics reported when you select **Statistics** as the uncertainty view for a result, select the **Statistics** option from the **Analysis option** popup menu.



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 has 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 is slight and the curve looks smooth. In other cases, the noise can have a large effect, and using a larger number of samples per plot point produces a smoother curve. There is a trade-off; with larger numbers the smoothing can miss details of the shape of the curve. PDFs might be much more susceptible to random noise than CDFs, so you might wish to use larger numbers for PDFs than CDFs. Ultimately, to reduce the noise, use a larger sample size (for details on selecting the sample size, see “Appendix A: Selecting the Sample Size” on page 372).

Equal probability steps

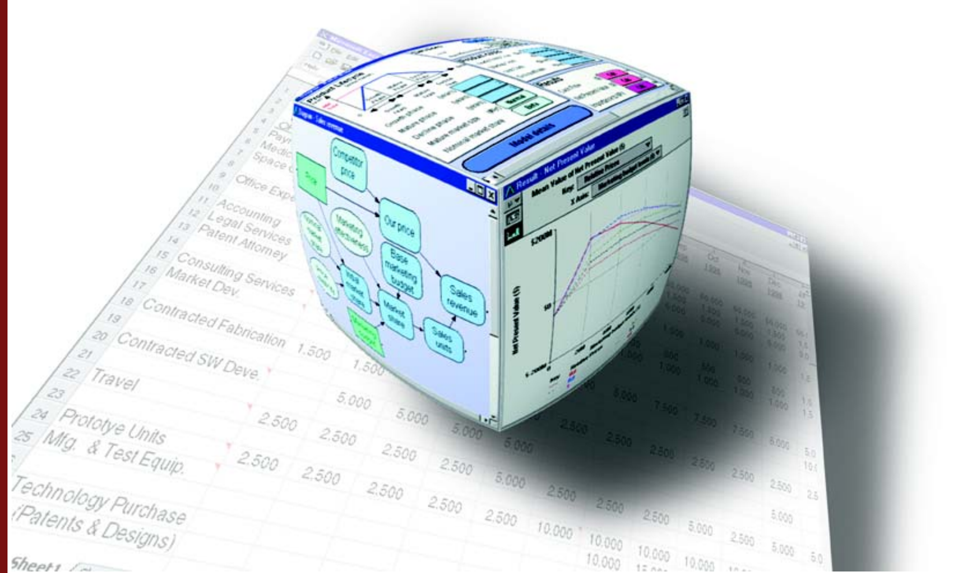
With this option, Analytica uses the sample to estimate a set of **m+!** fractiles (quantiles), **Xp**, at equal probability intervals, where **p**=0, **q**, 2**q**, ... 1, and **q** = 1/**m**. The cumulative probability is plotted at each of the points **Xp**, increasing in equal steps along the vertical axis. Points are plotted closer together along the horizontal axis in the regions where the density is the greatest. In the probability density graph view, the areas under the density function between successive fractiles are equal because they each represent the same probability, **q**. The density between two successive fractiles is plotted at the mid point (on the horizontal axis) of the two fractiles.

Equal X axis steps

With this option, Analytica estimates cumulative probability using equally spaced points along the **X** axis. In the probability density graph view, it shows a histogram where the height of each horizontal is estimated as the fraction of the sample values that fall within that **X** interval.

Chapter 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 Normal, Uniform, Bernoulli, 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?" on page 220 and "Glossary" on page 393 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 **Probtable** (page 239), 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** (page 253). You may set parameters to array values, specify an index to the optional **Over** parameter, or use functions from the **Multivariate** library.

Parametric discrete distributions

Bernoulli(p)

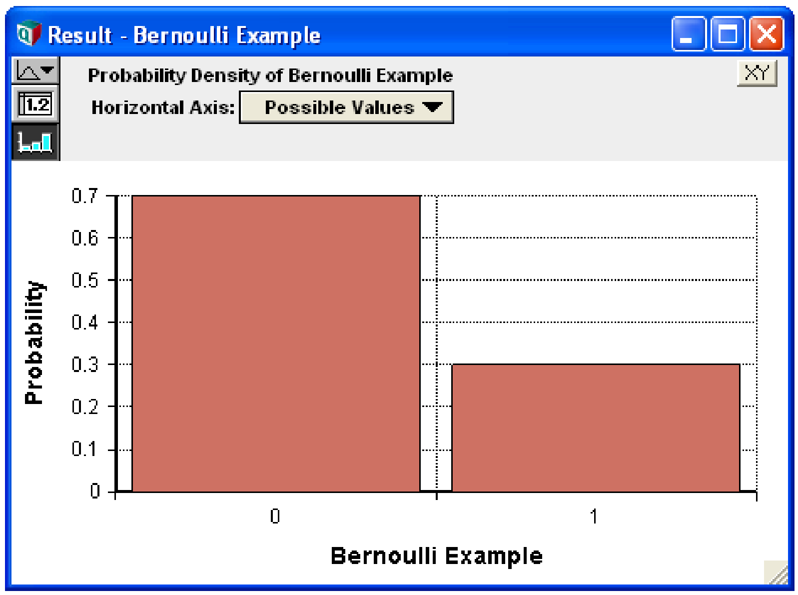
Defines a discrete probability distribution with probability **p** of result 1 and probability (1 - **p**) of result 0. It generates a sample containing 0s and 1s, with the proportion of 1s is approximately **p**. **p** is a probability between 0 and 1, inclusive, or an array of such probabilities. The 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 **`Binomial2`** 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**.

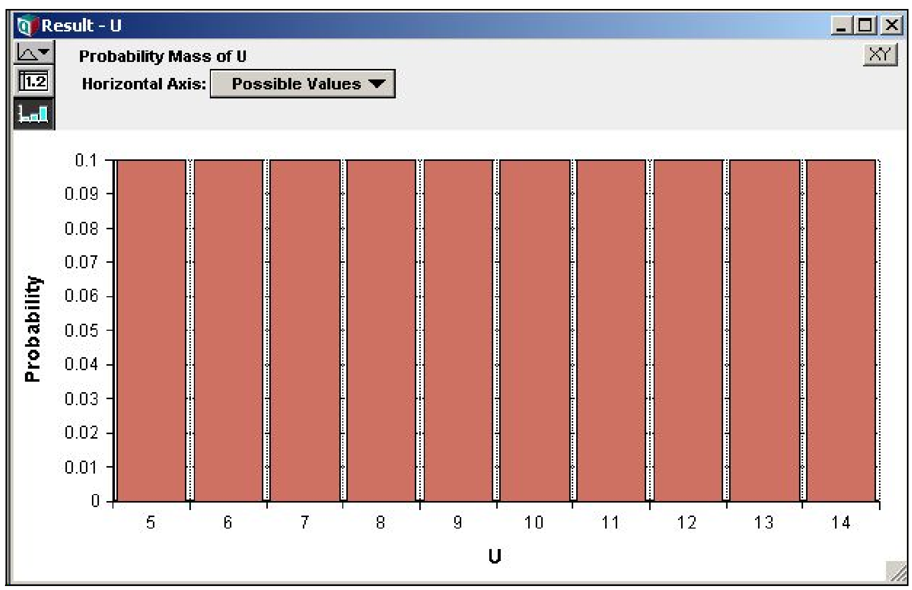
m The total number of successful events in the population, e.g, the number of red balls in the urn.

n The population size, e.g., the total number of balls in the urn, red and non-red.

Uniform(min, max, Integer: True)

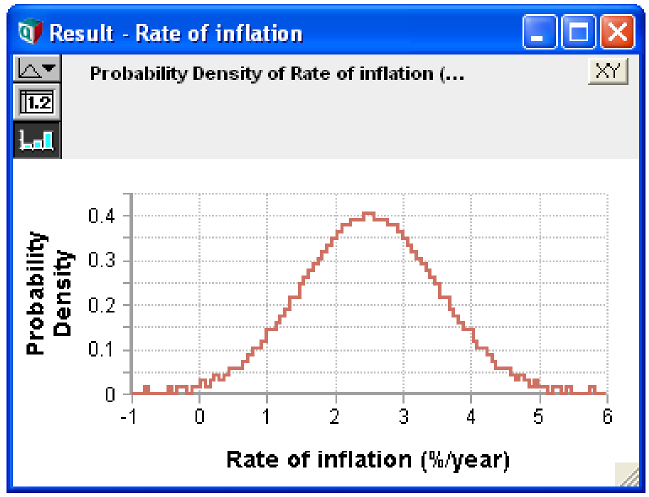
The **Uniform** distribution with the optional integer parameter set to True returns discrete distribution over the integers with all integers between and including **min** and **max** having equal probability.

```
Uniform(5, 14, Integer: True) →
```



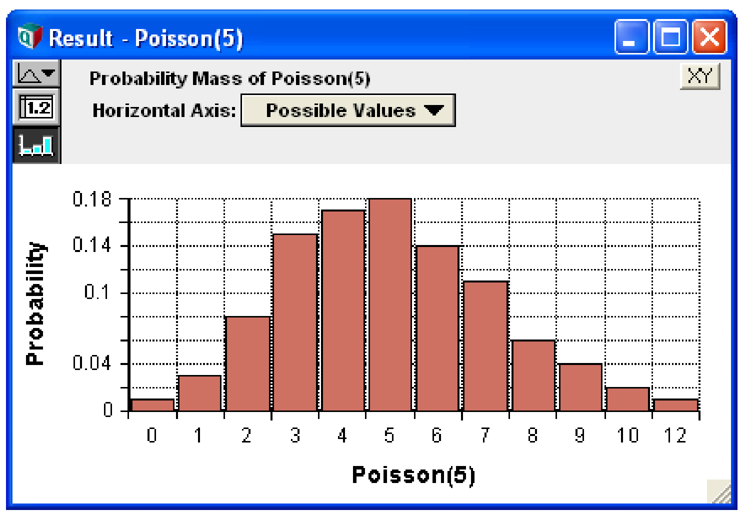
Probability density and mass graphs

When you select the **Probability density** as the **uncertainty view** (page 33) for a *continuous* variable, it graphs the distribution as a ***Probability Density function***. The height of the density shows the relative likelihood the variable has that value.

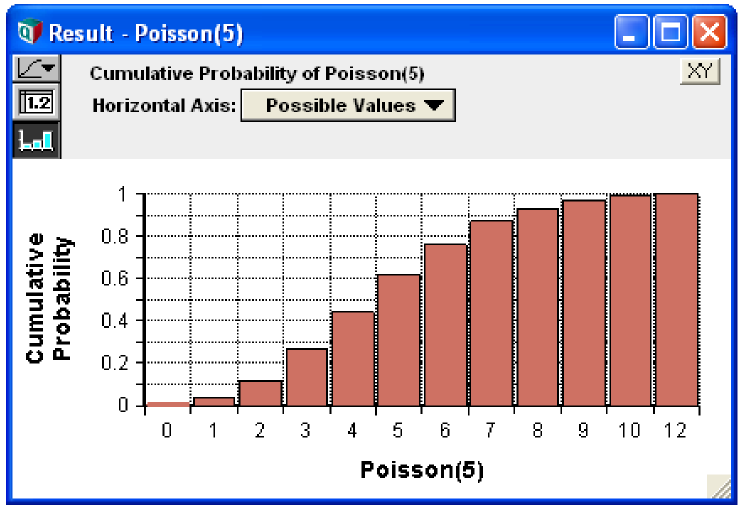


Technically, the probability density of variable **X**, means the probability per unit increment of **X**. The units of probability density are the reciprocal of the units of **X** — if the units of **X** are dollars, the units of probability density are probability per dollar increment

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** has 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 can contain numbers that are not integers, which it might 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

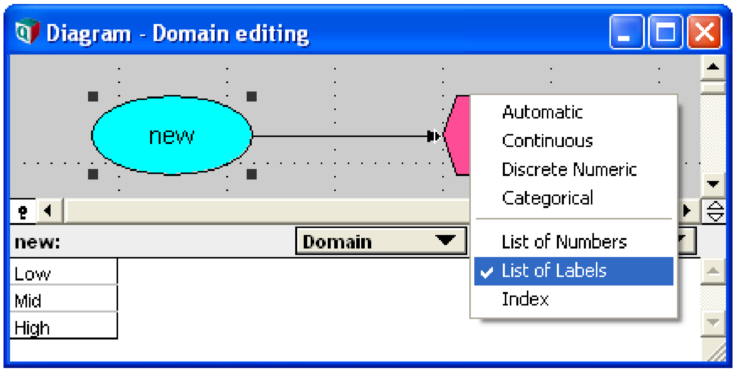
The ***domain*** attribute specifies the set of possible values for a variable. You rarely need to view or change a domain attribute explicitly. The most common reason to set the domain is for a variable defined as a custom discrete distribution, especially **ProbTable**. You can do this by editing it directly as an index in the **probtable view** (page 238), so you can usually ignore the information below. The rare case you 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 can 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.

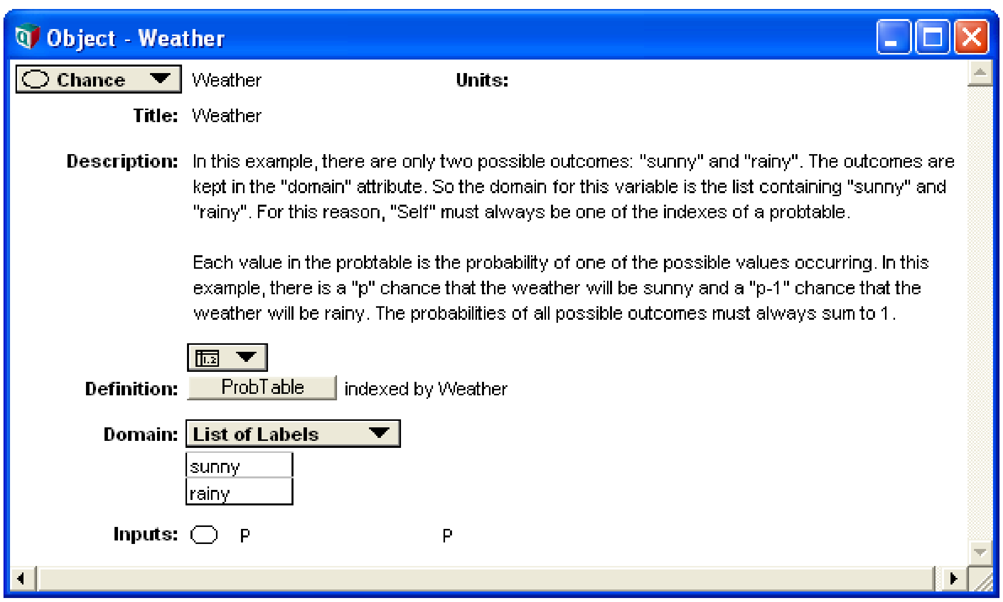
List of Labels: You specify a list of label (text) values, as illustrated.

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 “Creating an index” on page 163).

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” on page 306).



Tip

The domain of a discrete variable should include all its possible values. If not, its probability mass function might 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.

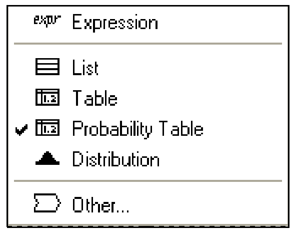
Probtable(): Probability Tables

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 143).

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** (page 24) of the **Diagram** window.
3. Press the ***expr*** menu above the definition field and select **Probability Table**.

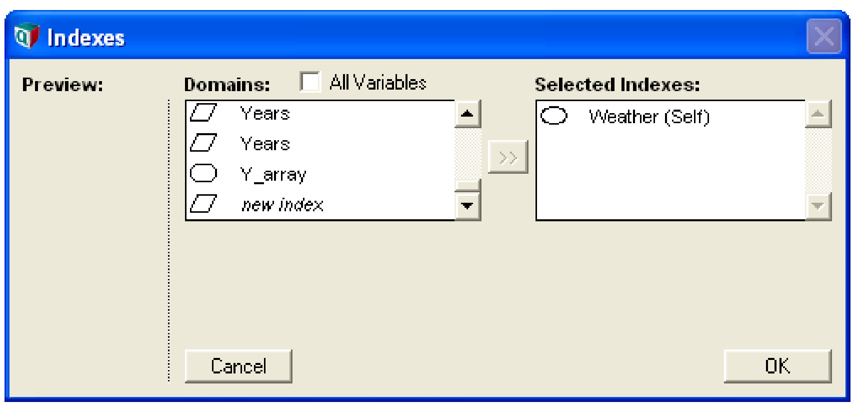


If the variable already has a definition, it confirms that you wish to replace it.

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 “Defining a variable as an edit table” on page 169).

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.

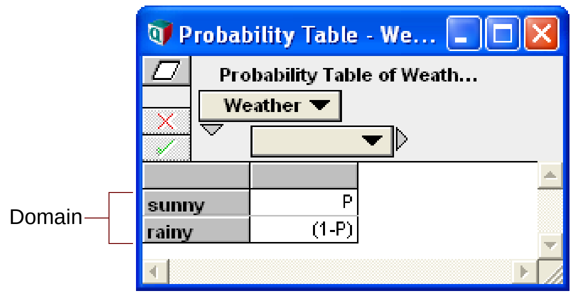


Tip

`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 **P** 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 like this.



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 might 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 238), the probability of rain might 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.
3. Click the *All Variables* checkbox above the left hand list.
4. Move the desired variable, e.g., **Season**, to add it as an index.
5. Click **OK** 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.

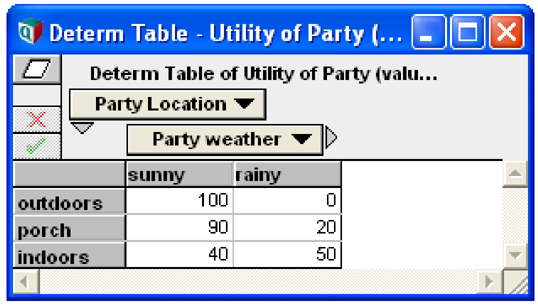
Determtable(): Deterministic conditional table

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. DetermTable() is described in Chapter 11, “Arrays and Indexes,” on page 199, but we also include it in this section on discrete probability distributions, even though it is not probabilistic, because you usually use it in conjunction with **Probtable** and other discrete distributions. It is an editable table, like **Probtable**, but with a single (deterministic) value, number, or text, in each cell.

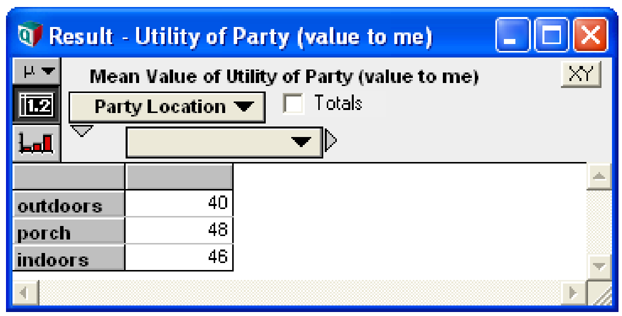
The **Determtable()** function looks like an edit table or a probability table, with an index (dimension) from each discrete variable on which it depends. Unlike **Probtable**, it does not need a self index. Its result is probabilistic if any of its conditioning variables are probabilistic.

For the steps to create a determTable, see “Creating a DetermTable” on page 200.

Example In “Create a probability table” on page 238, **`Weather`** is defined as a probability table. If **`P`**, the probability of “sunny” is 0.4, then the probability of “rainy” is 0.6. **`Party_location`** is a decision variable with values **`['outdoors', 'porch','indoors']`**. **`value_to_me`** is a determtable, containing utility values (or “payoffs”) for each combination of **`Party_location`** and **`Weather`**.



Evaluating **`value_to_me`** gives the value of each party location, considering the uncertain distribution of **`Weather`**. The mean value of **`value_to_me`** is the expected utility.



Chancedist(p, a, i)

Creates a discrete probability distribution, 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**.

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:`

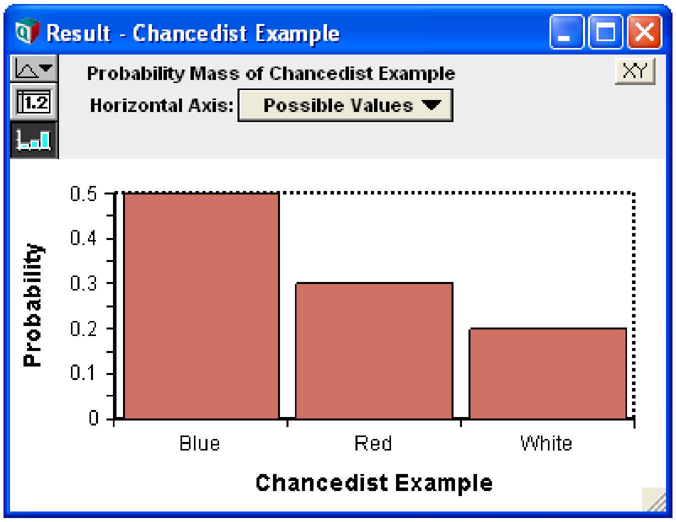
Red	White	Blue

`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 can look slightly different due to random variation in the Monte Carlo sampling.

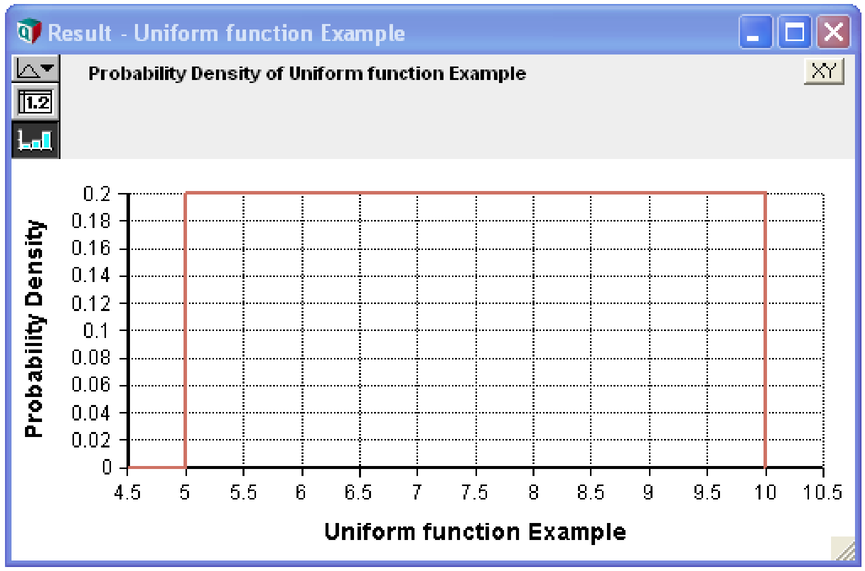
Uniform(*min, max*)

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)" on page 234.

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)` →



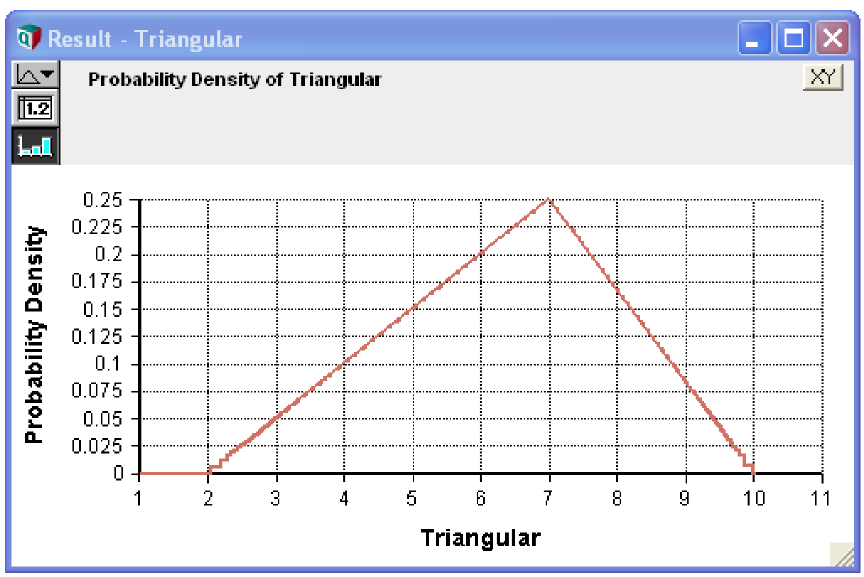
Triangular(min, mode, max)

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)` →



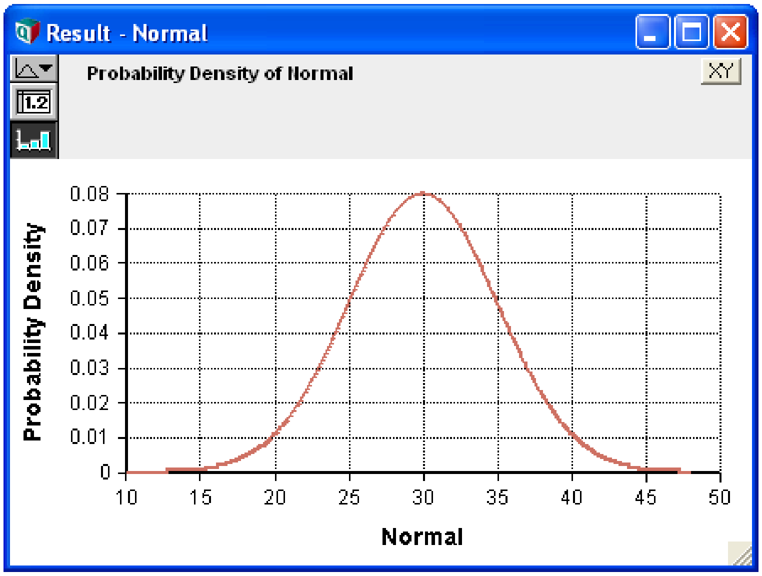
Normal(*mean, stddev*)

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.

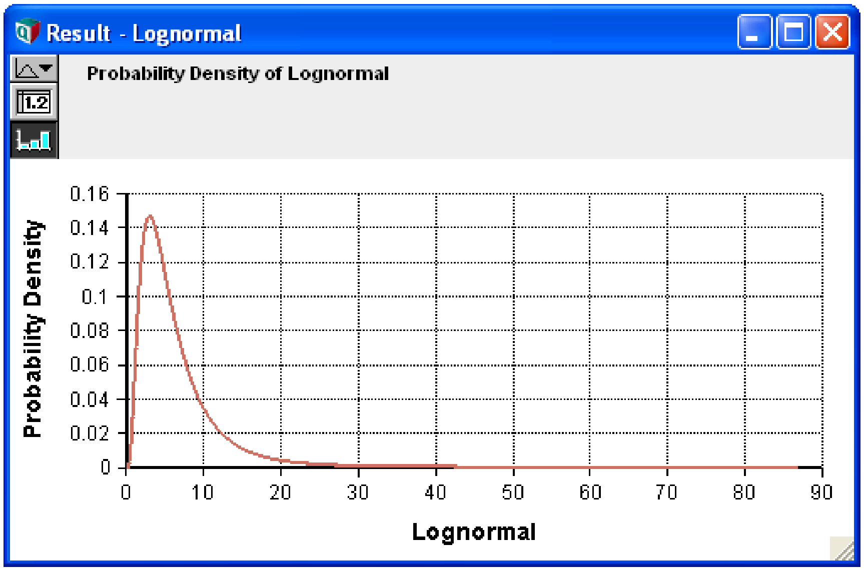
If **x** is lognormal **Ln(x)** has a normal distribution with mean **Ln(median)** and standard deviation **Ln(gsdev)**.

When to use Use the lognormal distribution if you have a sharp lower bound of zero but no sharp upper bound, a single mode, and a positive skew. The gamma distribution is also an option in this case. 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 to 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)
```

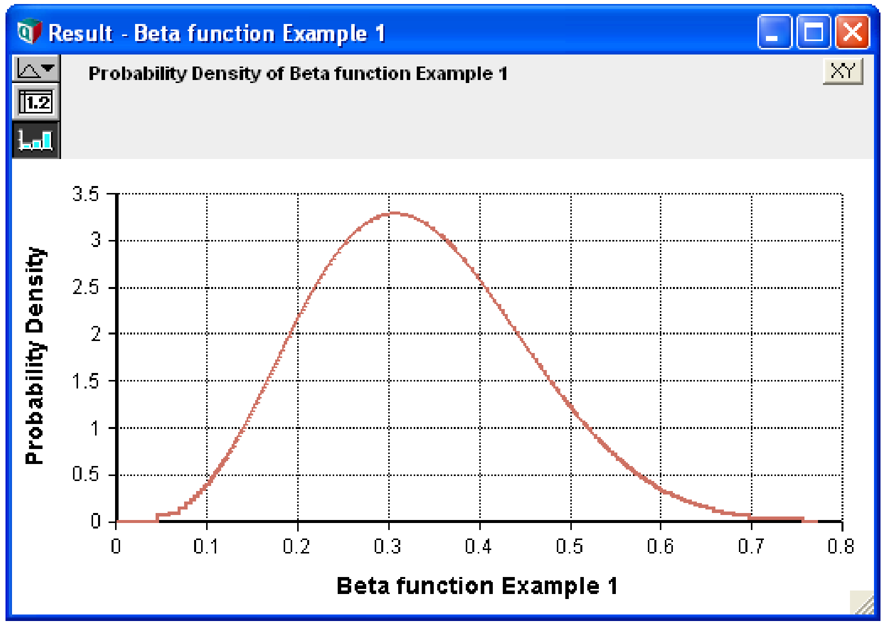
If you observe **r** heads in **n** tosses of the coin, your new (posterior) opinion about h, should be:

```
Beta(1 + r, 1 + n - r)
```

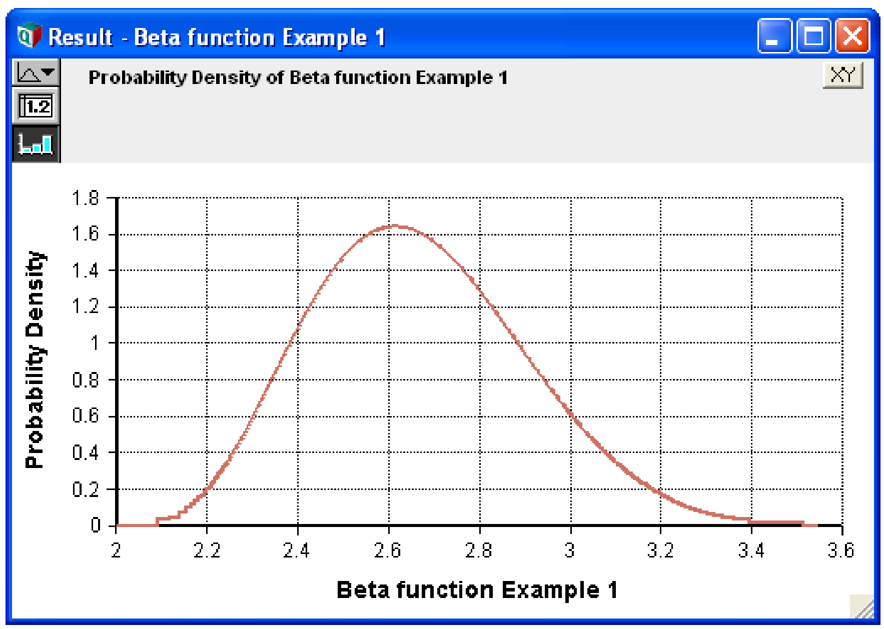
If the uncertain quantity has lower and upper bounds other than 0 and 1, include the lower and upper bounds parameters to obtain a ***transformed beta*** distribution. The transformed beta is a very flexible distribution for representing a wide variety of bounded quantities.

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 $a \to \infty$, the gamma distribution approaches a normal distribution in shape.

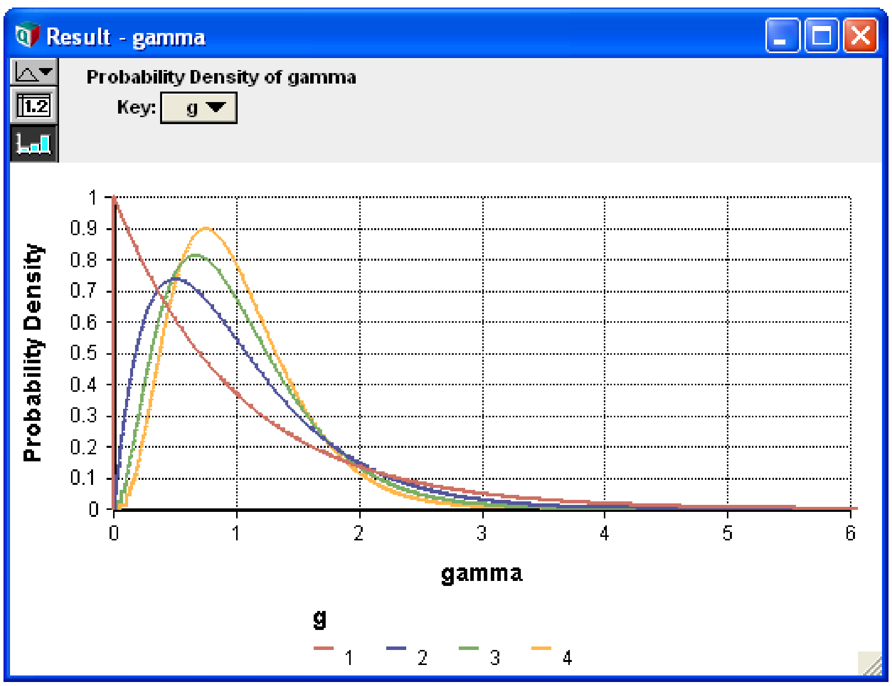
The gamma distribution encodes the time required for **a** events to occur in a Poisson process with mean arrival time of **b**.

Tip Some textbooks use Rate=1/**b**, instead of **b**, as the scale parameter.

When to use Use the gamma distribution with **a**>1 if you have a sharp lower bound of zero but no sharp upper bound, a single mode, and a positive skew. The Lognormal distribution is also an option in this case. **Gamma()** is especially appropriate when encoding arrival times for sets of events. A gamma distribution with a large value for **a** is also useful when you wish to use a bell-shaped curve for a positive-only quantity.

Library Distribution

Examples Gamma distributions with *mean=1*



Logistic(m, *s*)

The logistic distribution describes a distribution with a cumulative density given by:

$$F(x) = \frac{1}{1 + e^{\frac{-(x-m)}{s}}}$$

The distribution is symmetric and unimodal with tails that are heavier than the normal distribution. It has a mean and mode of **m**, variance of:

$$\frac{\pi^2 \times s^2}{3}$$

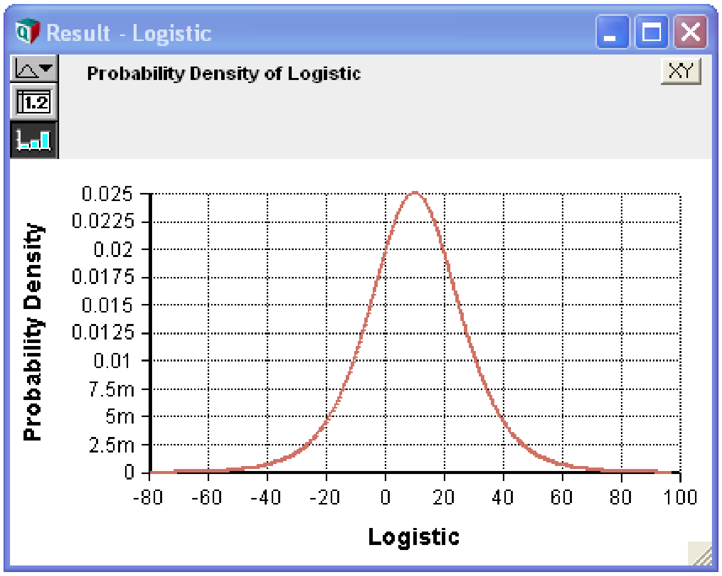
and kurtosis of 6/5 and no skew. The scale parameter, **s**, is optional and defaults to 1.

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* might be the dosage of a toxin, and *p(x)* the probability of death for an animal exposed to that dosage. Using *p(x)* = *F(x)*, the logit of *p*, given by:

Logit(p(x)) = Ln(p(x) / (1-p(x))) = x/s - m/s

This 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 - x)/(s Sqrt(n))
```

where ***x*** is the sample mean, ***m*** is the actual mean, ***s*** is the sample standard deviation, and ***n*** is the sample size. ***T*** is distributed according to StudentT with **d = *n-1*** degrees of freedom.

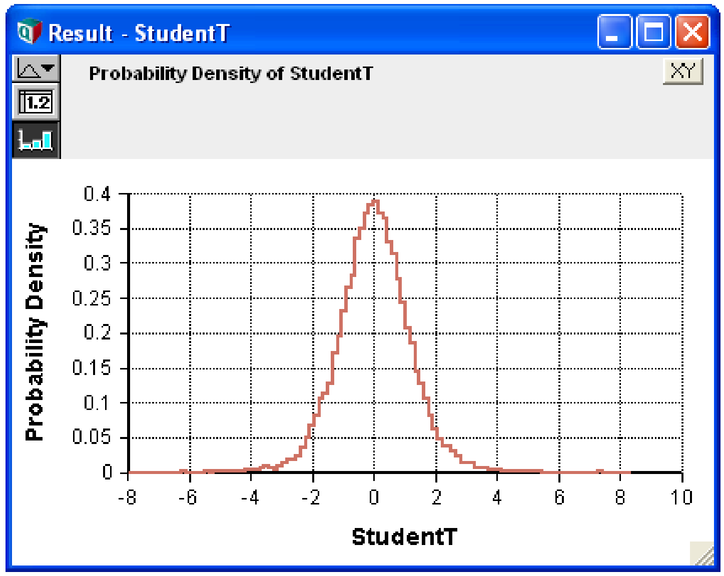
The StudentT distribution is often used to test the statistical hypothesis that a sample mean is significantly different from zero. If **`x1..xn`** measurements are taken to test the hypothesis ***m***>0:

```
GetFract(StudentT(n-1), 0.95)
```

This is the acceptance threshold for the ***T*** statistic. If ***T*** is greater than this fractile, we can reject the null hypothesis (that ***m***<=0) at 95% confidence. When using **GetFract** for hypothesis testing, be sure to use a large sample size, since the precision of this computation improves with sample size.

The 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.

Example `StudentT(8)`



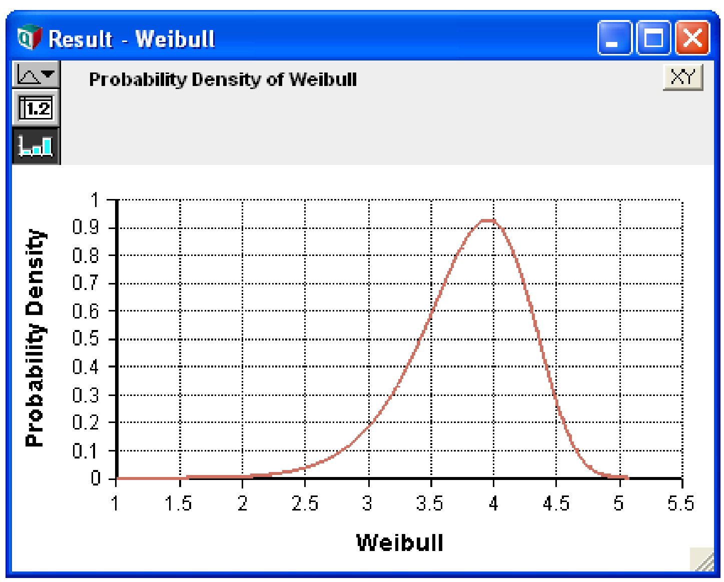
Weibull(n, *s*)

The Weibull distribution has a cumulative density given by:

$$f(x) = 1 - e^{-\left(\frac{t}{s}\right)^{n}}$$

for **t >= 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, $f(x)$ might 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 $\mathbf{y_i}$ 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**, for an array of corresponding outcome values, **r**. The values of **p** must be nondecreasing and should start with 0 and end with 1. The values of **r** must also be be nondecreasing over their common index. If **p** is an index of **r**, or **r** is an index of **p**, or if both have the same single index, the correspondence is clear and so you can omit **i**. Otherwise, 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

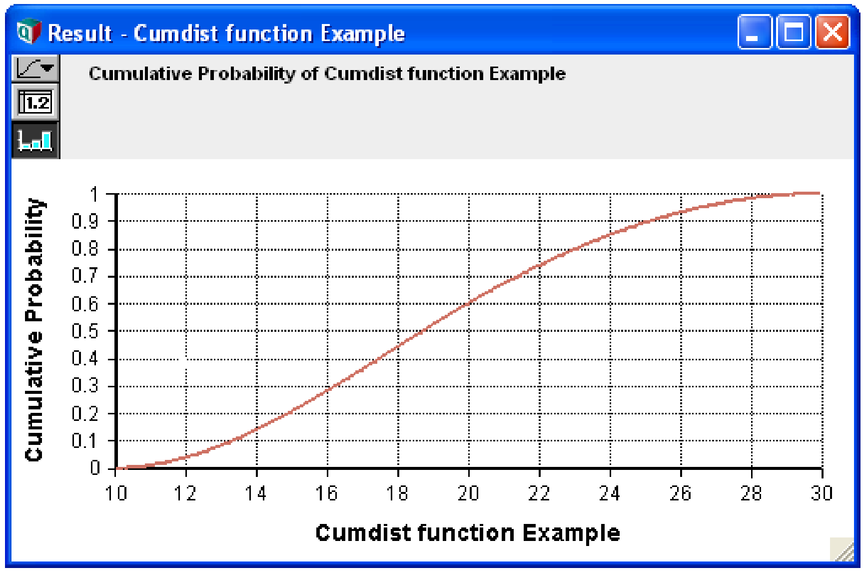
Array_b →
Index_a ▶

	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 $2\mathbf{r}_1 - \mathbf{R}_r$ (or $2\mathbf{r}_n - \mathbf{r}_{n-1}$).

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

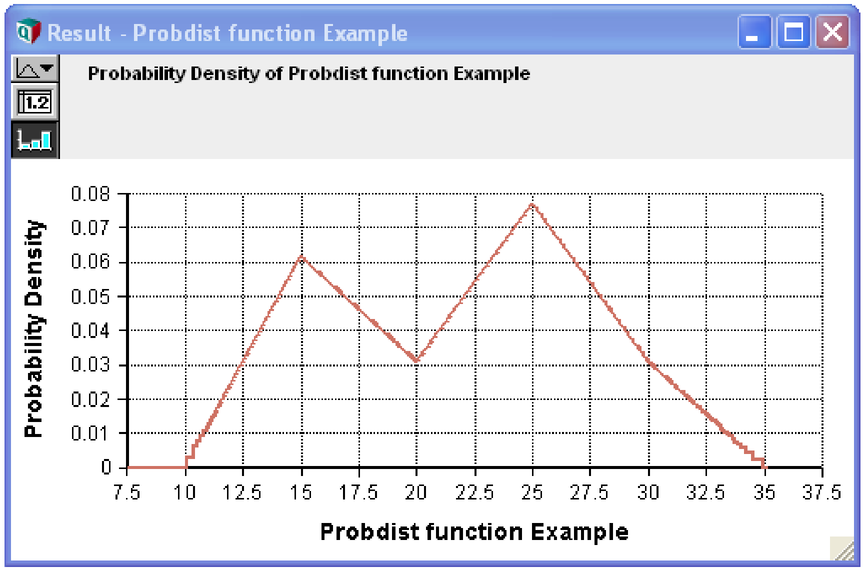
```
Array_p →
Index_a ▶
```

	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" on page 120) 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*)

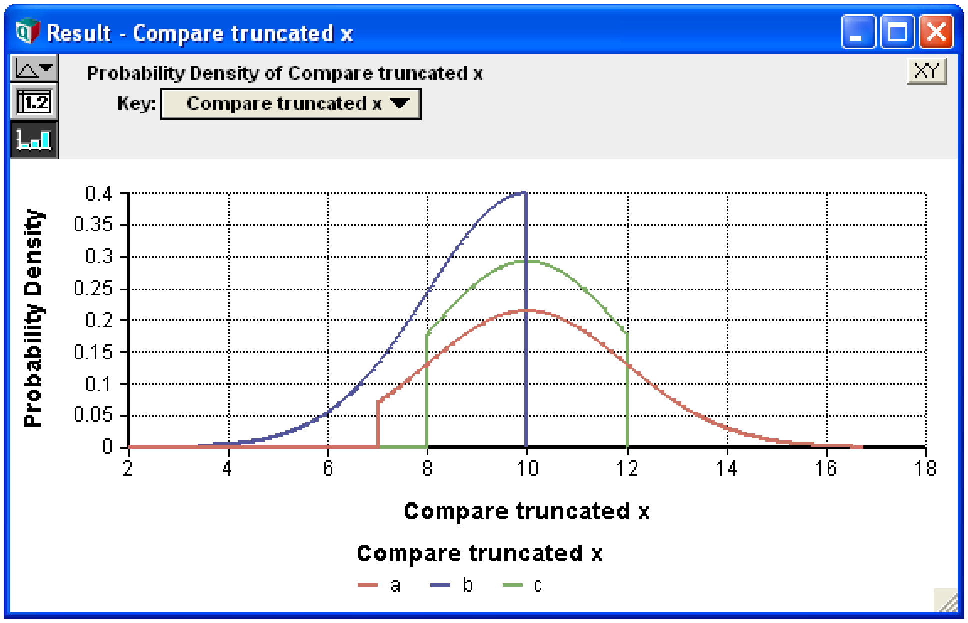
Truncates an uncertain quantity **u** so that it has no values below **min** or above **max.** You must specify one or both **min** and **max**.

It does not discard sample values below **min** or above **max**. Instead, 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))
```

This 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 prob 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 of 0=default, 1=Minimal standard, 2=L'Ecuyer, or 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, and 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 generates 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 generates 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 is an array of normal distributions over index **K**, 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 **k**:

```
Variable Ysd := Table(K)(2, 3, 4)
Variable Y := Normal(Xmean, Ysdeviation)
```

it generates an array of normal distributions over index **K,** 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 also appear in the result, following the usual rules of Intelligent Arrays.

Multivariate Distributions 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 310.

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 might 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 is also not normal, and the correlation is approximate. It generalizes appropriately if any of the parameters are arrays. The result array has 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 might 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** might 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 can 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** can 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 2D 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 * Product(x^(alpha-1), i)
```

where **k** is a normalization factor equal to:

```
GammaFn(Sum(alpha, i))/Sum(GammaFn(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

```
Sum(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., "A simple approach to the generation of uniformly distributed random variables with prescribed correlations," *Comm. in Stats - Simulation and Computation* 28: 785-791 (1999).

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 **x[t]**, but with samples reordered to have the specified rank correlation **r** between successive values. If **r** is indexed by **t**, **r[@t=k]** specifies the rank correlation between **y[@t=k]** and **y[@t=k-1]**. Then the first correlation, **r[@t=1]**, is ignored.

Normal_additive_gro(x, m, s, r, t)

Generates an array of values over index **t**, with the first equal to **x**, and successive values adding an uncertain growth, normally distributed with mean **m** and standard deviation **s**. If we denote the result by **g**, **r** specifies a serial correlation between **g[@t = k]** and **g[@t=k-1]**. **x**, **m**, **s**, and **r** each can be indexed by **t** if you want them to vary over **t**.

Dist_additive_growth(x, g, rc, t)

Generates an array of values over index **t**, with the first equal to **x**, and successive values adding an uncertain growth **g**, and serial correlation **rc** between **g[@t = k]** and **g[@t=k-1]**. **x**, **g**, and **rc** each can be indexed by **t** if you want them to vary over **t**.

Normal_compound_gro(x, m, s, r, t)

Generates an array of values over index **t**, with the first equal to **x**, and successive values multiplied by compound growth 1+**g**, where **g** is normally distributed with mean **m** and standard deviation **s**. It applies serial correlation **r** between **g[@t = k]** and **g[@t=k-1]**. **x**, **g**, and **rc** each can be indexed by **t** if you want them to vary over **t**.

Dist_compound_growth(x, g, rc, t)

Generates an array of values over index **t**, with the first equal to **x**, and successive values multiplying by an uncertain compound growth **g**, and serial rank correlation **rc** between **g[@t = k]** and **g[@t=k-1]**. **x**, **g**, and **rc** each can be indexed by **t** if you want them to vary over **t**.

Uncertainty over regression coefficients

For a description of **RegressionDist()**, **RegressionNoise()**, and **RegressionFitProb()**, see "Uncertainty in regression results" on page 279.

Importance weighting

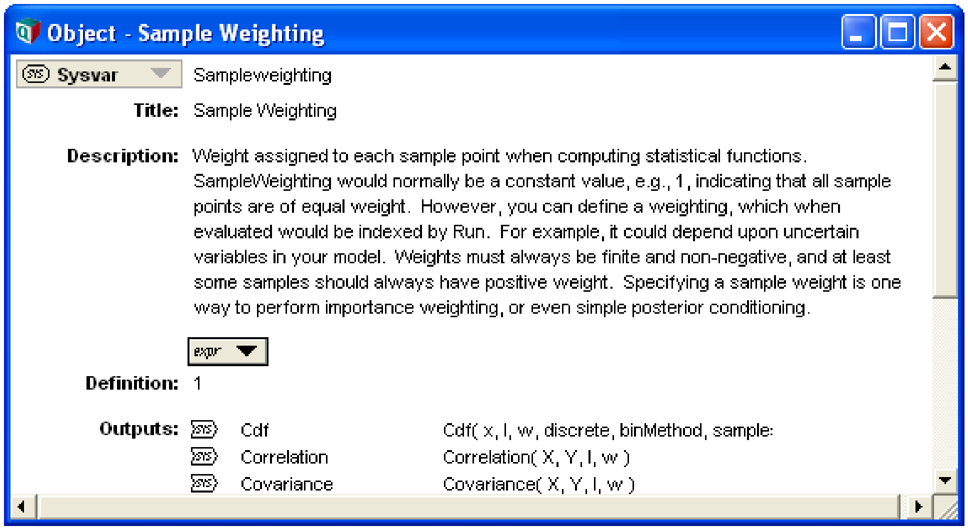
Importance weighting is a powerful enhancement to Monte Carlo and Latin hypercube simulation that lets you get more useful information from fewer samples; it is especially valuable for risky situations with a small probability of an extremely good or bad outcome. By default, all simulation samples are 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:

$$f(x) / ((p\ f(x) + (1 - p)\ cr(x)) \qquad \textbf{(3)}$$

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))
```

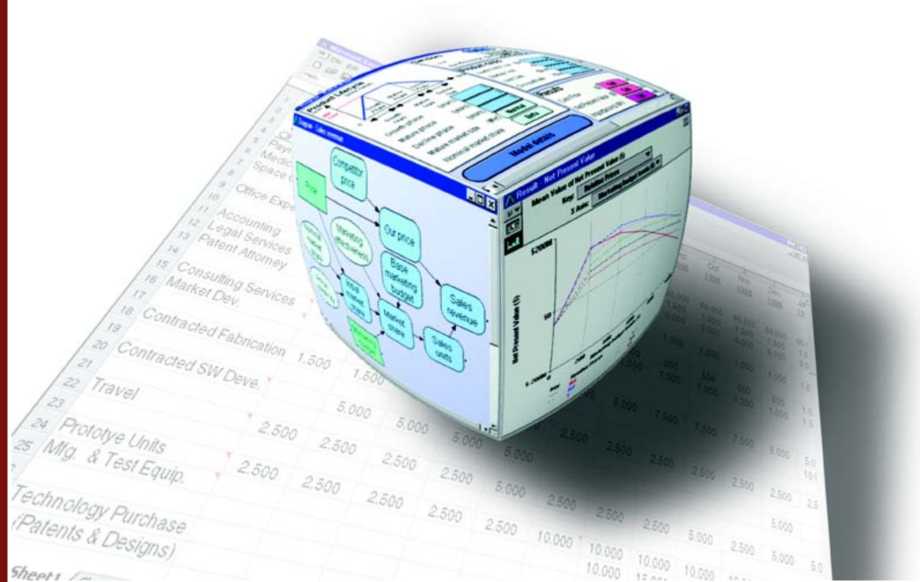
Thus, we compute a **Weighted_demand** as a mixture between the original distribution on **Demand** and the distribution in the critical region, **Excess_demand**. We assign weights to **SampleWeighting**, using the **Object** window for **SampleWeighting** opened as described above. See the Analytica Wiki at http://www.lumina.com/wiki for more.

For more on weighted statistics and conditional statistics, see "Weighted statistics and w parameter" on page 268.

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)
- Functions that show the sensitivity of a variable to one or more variables that affect it, including WhatIf and Tornado analysis
- 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 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** (page 33) available in the **Result** window use these functions.

Statistical functions force prob mode evaluation

Unlike other functions, statistical functions 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 vary from one evaluation to the next due to the variability inherent in random sampling. Hence, your results might not exactly match the results shown in the examples here. For greater precision, use a larger sample size (see “Appendix A: Selecting the Sample Size” on page 372 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(x)” on page 263)
s	Standard deviation (see “Sdeviation(x)” on page 263)
m	Sample size (see “Appendix A: Selecting the Sample Size” on page 372)

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 this 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 this 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 this 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 this 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 this formula.

$$\left(\frac{1}{m}\sum_{i=1}^{m}\left[\frac{x_i-\bar{x}}{\sigma}\right]^4\right)-3$$

Library Statistical

Example `Kurtosis(Skfuel_prices) → -0.48`

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

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** (page 225), *Probability Bands* option.

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** (page 268).

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" on page 225). 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

(Compare to the Bernoulli example on page 233.)

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** (page 225).

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** (page 225), Statistics option.

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)
```

These functions evaluate **x** in prob mode, and return 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 **Y** 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 “The domain attribute and discrete variables” on page 236). Alternatively, you can control the result by setting the optional parameter **discrete** as true or false. For example:

```
Variable X := Poisson(20)
PDF(X, Discrete: True)
```

This generates a discrete histogram over **X**. If **X** contains text values, i.e., categorical data, you might 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**.

In additional, **PDF** and **CDF** have these 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)
```

This 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 **Y** 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” on page 257, 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 might 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

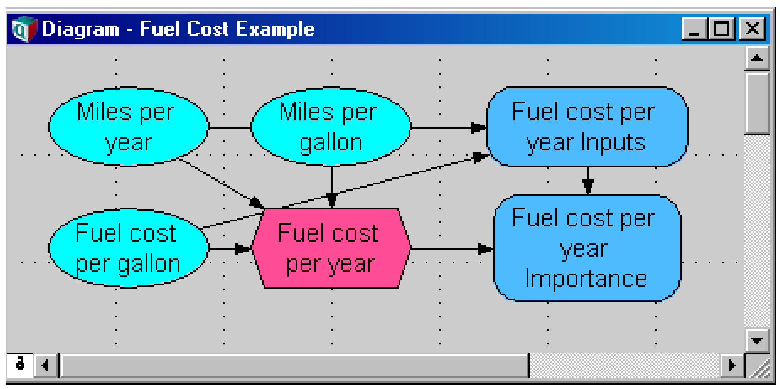
1. Be sure you are in edit mode, viewing a **Diagram** window. Select an output variable, **`U`**, that depends on two or more uncertain inputs — possibly, an objective.
2. Select **Make Importance** from the **Object** menu.

If the selected output is **`U`**, it creates an index **`U_Inputs`**, a list of the uncertain inputs, and a general variable, **`U Importance`**, containing the importance of those inputs to the output.

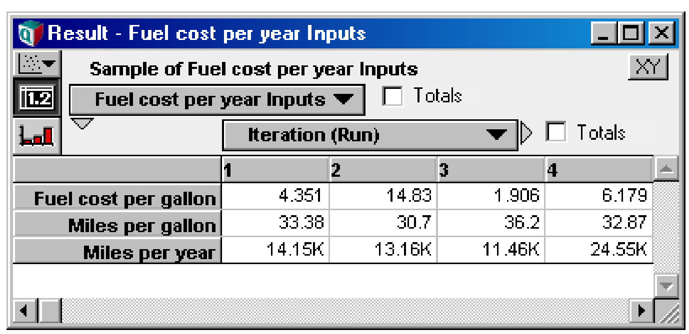
Example

```
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 defined as a list identifiers, containing all the chance variable ancestors of the output node. It evaluates to an array of probability distributions, one for each chance variable. This array is self-indexed, with the index values consisting of handles to each input variable.

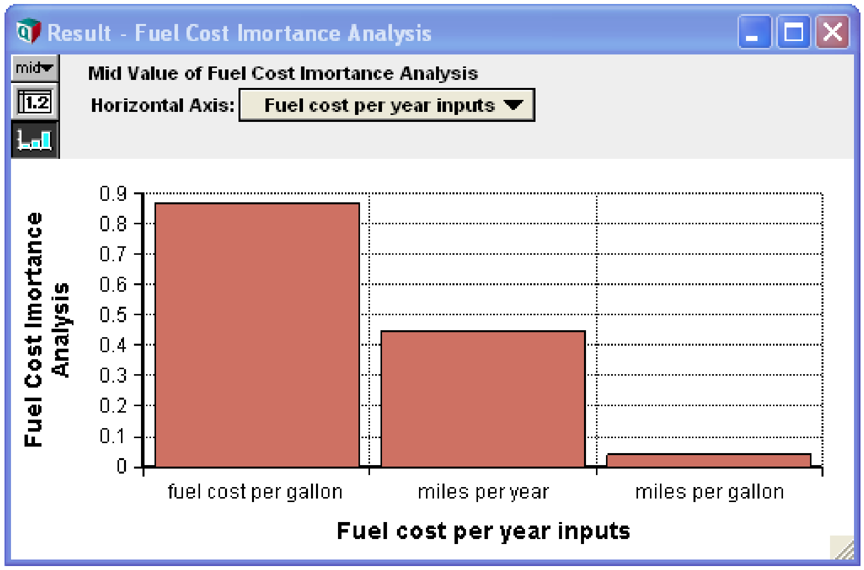


	1	2	3	4
Fuel cost per gallon	4.351	14.83	1.906	6.179
Miles per gallon	33.38	30.7	36.2	32.87
Miles per year	14.15K	13.16K	11.46K	24.55K

`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 can vary slightly from one computation to another due to random noise. For a sample size of 100, an importance of 0.1 might 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 **`U`**, and later add or remove uncertain variables that affect **`U`**, the uncertainty analysis is not automatically updated to reflect those changes. You can update the analysis either by:

- Select **`U`** and then select **Make Importance** from the **Object** menu. It automatically updates the importance analysis to reflect any new or removed uncertain inputs.
- Draw an arrow from any new uncertain input into index **`U inputs`**. It adds the new variable as an uncertain input. Similarly, you can remove a variable from **`U inputs`** by redrawing an arrow from that variable into **`U inputs.`**

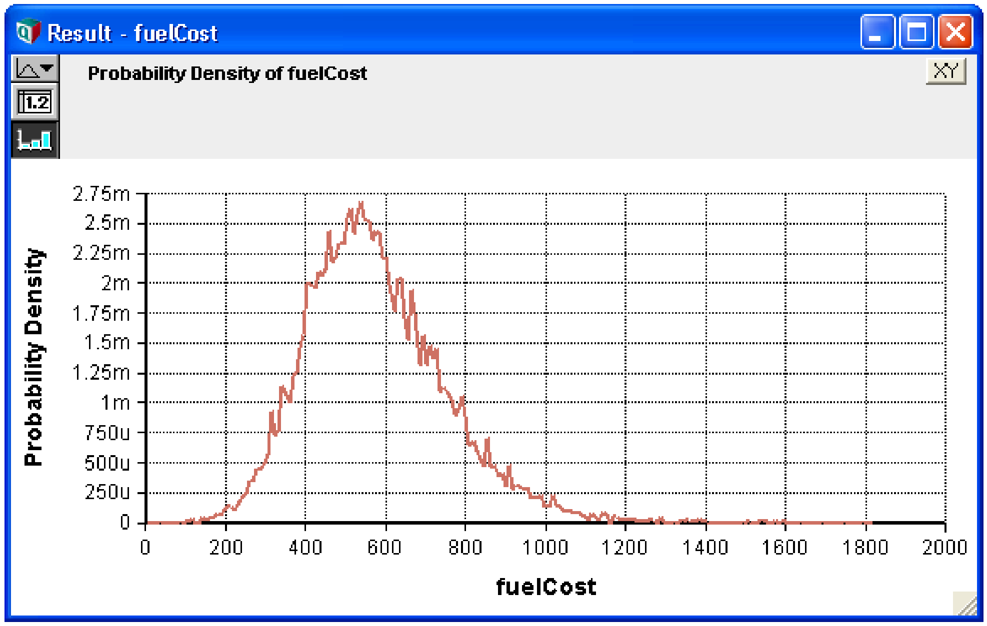
Sensitivity analysis functions

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:

`gasPrice`	**`Normal(1.3, .3)`**	Cost of gasoline per gallon within market fluctuations
`mpy:`	**`12K`**	The average number of miles driven per year
`mpg:`	**`Normal(28, 5)`**	Fuel consumption averaged over driving conditions
`fuelCost:`	**`gasPrice * mpy / mpg`**	Annual cost of fuel

The probability density of **`fuelCost`** is shown below.



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**. Mathematically, writing $y(x)$ to emphasize that **y** is a function of **x**, elasticity is defined as:

$$Elasticity(y, x) = \lim_{u \to 0} \frac{1}{u}\left(\frac{y(x(1+u)) - y(x)}{y(x)}\right) \quad (3)$$

When **x** is a positive scalar, but not when **x** is array-valued, **Elasticity()** is related to **Dydx()** in the following manner:

```
Elasticity(y, x) = Dydx(y, x)*(x/y)
```

Library Special

Examples

```
Elasticity(fuelCost, mpg) → -1
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).

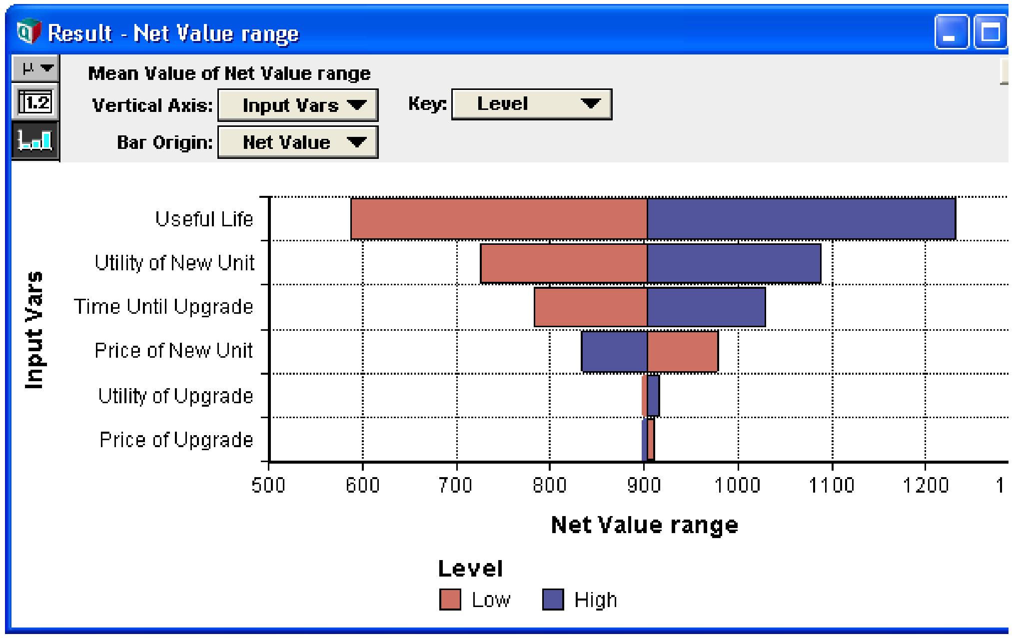
Suppose *Z* is a function of *A*, *B*, and *C*, and we wish to examine the effect on *Z* when each input is varied, one at a time, by *10%* from its nominal value. Define:

```
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 that might 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.*

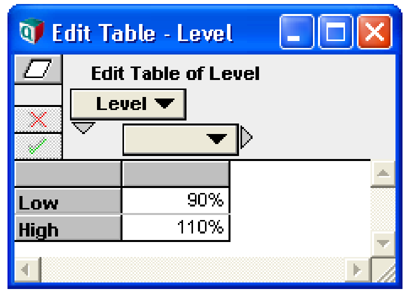
There are several options for selecting low and high values, including:

- 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, **`Level`**, and define it as a self-indexed table. (To do this, select **Table** from the ***expr*** menu, 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 **`Net_value`**. Define **`Tornado_Analysis`** as:

   ```
   WhatIfAll(X, Vars, Level * Vars)
   ```

4. Create a node, **`Input_Vars`**, defined as:

   ```
   sortIndex(-abs(Tornado_Analysis[Level='high'] -
   Tornado_Analysis[Level='low']))
   ```

5. Create a node, **`Net_value_range`**, to hold the final graph, defined as:

   ```
   Tornado_Analysis[Vars=Input_Vars]
   ```

Steps 4 and 5 are not necessary if you do not require your bars to be displayed from largest to smallest. If you do include steps 4 and 5, **`Net_value_range`** contains 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 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 is, in general, different for the different bars. If you have extra indexes in your tornado analysis, you need to either skip steps 4 and 5 above, and display non-sorted Tornados, or select a single sort order based on whatever criteria fits your needs, realizing that not all tornados display in sorted order.

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, **`Level`** would be set to the absolute low and high values, and the third parameter to **WhatIfAll()** would be simply **`Level`**.
- For the case where you select absolute low and high values separately for each input variable, you would index **`Level`** by **`Vars`**, fill in **`Level`**'s table appropriately, then set the third parameter to be just **`Level`**.
- And for the case where you vary all inputs between two given fractiles, you would set **`Level`** to the desired fractiles, and use the expression **`getFract(Net_value,Level)`** as the third parameter.

Graphing a tornado

It's customary to graph a tornado with the names of the input variables are listed down the vertical axis, and the bars displaying the effect on the output horizontally:

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 the **Chart Type** tab.
4. Set the *Line Style* to the filled bar setting and check the *Variable origin* checkbox. This will also set *Bar Overlap*=100% and *Swap horizontal and vertical* for you. Click **Apply**.
5. Next, we want to compare to the baseline value of **`Net_Value`**. Click the **XY** button to open the **XY Comparison Sources** dialog, check *Use another variable*, press **Add...**, and in the **Object Finder** select the variable **`Net_Value`**. Press OK twice.

6. In the Bar Origin pulldown, select **`Net_value`**.

X-Y plots

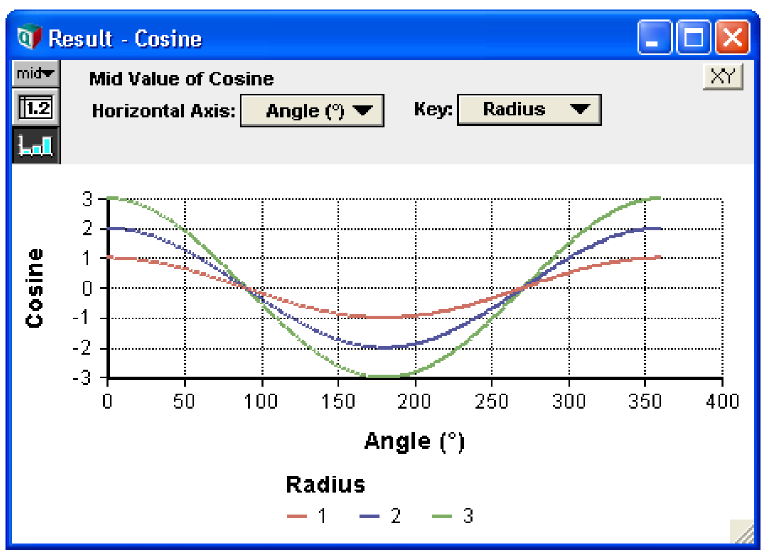
You can compare the result of a variable against another variable, or one column against another column of a result, using an XY-plot. XY plots can be graphed for **`Mid`**, **`Mean`**, **`Statistics`**, **`Probability Bands`**, and **`Sample`** view modes.

To graph one variable against another:

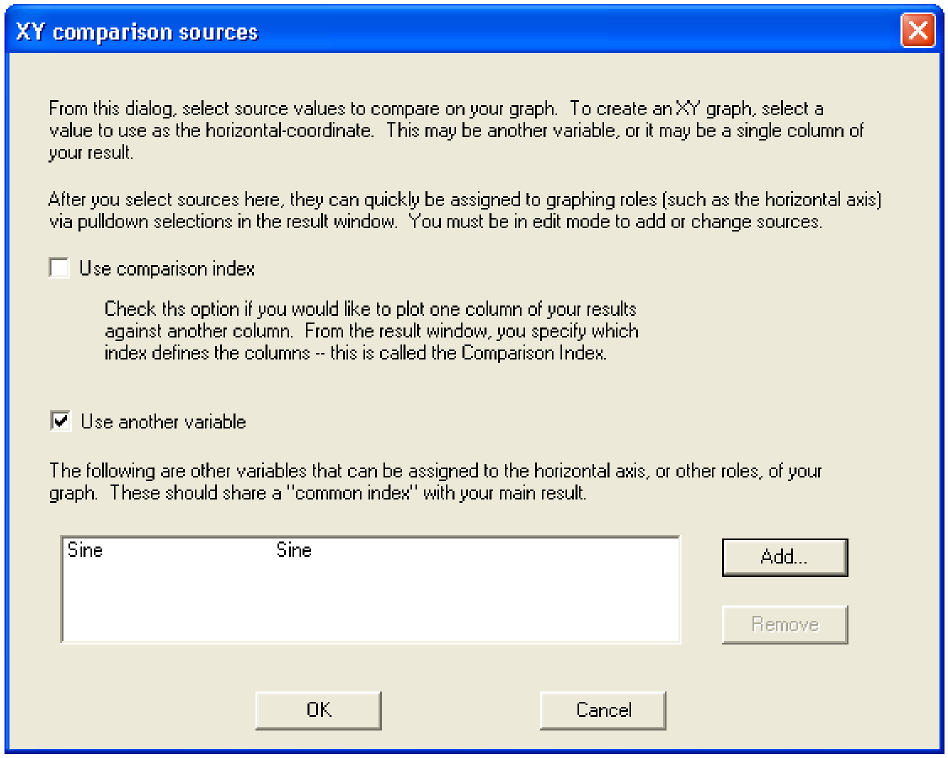
1. Open a **Result** window for the **`y-`** (vertical axis) variable.
2. Click the **XY** button XY located in the top-right corner of the window to open the **XY Comparison Sources** dialog.
3. Check the *Use another variable* checkbox, press **Add...**, and in the **Object Finder**, select the **`x-`** (horizontal axis) variable.

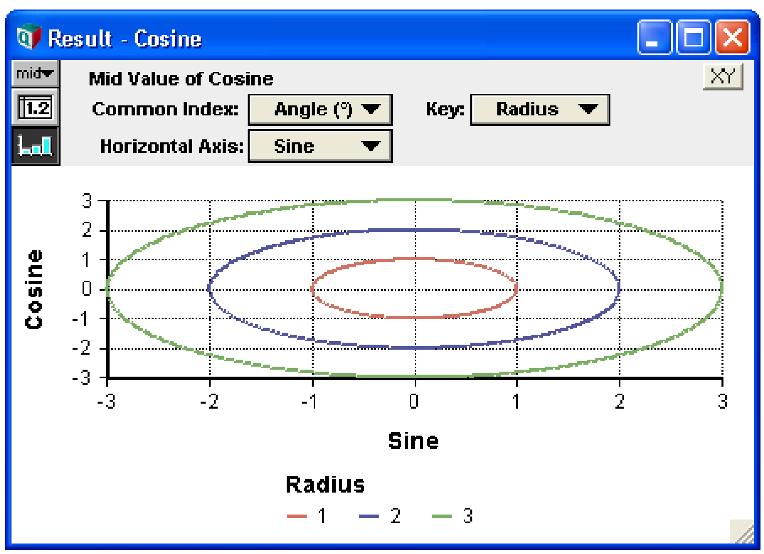
The two variables in an **XY** window must share at least one index, and all indexes of **x** must also be indexes of **y**. The popup menu in the index selection area becomes **Common Index** — only indexes of both **x** and **y** might be selected.

```
Variable Angle := Sequence(0, 360, 10)
Variable Radius := 1..3
Variable SinX := Radius * Sin(Angle)
Variable Cosine := Radius * Cos(Angle) →
```

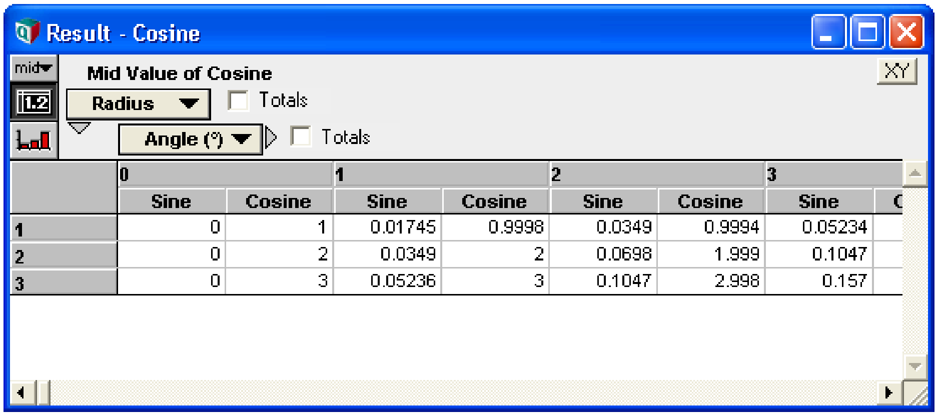


Click the **XY** button, check *Use another variable*, then **Add....**, and in the **Object Finder** dialog under **Current Module** select the variable **`Sine`** to display this result.





Click the **Table View** button to display this result.



	0		1		2		3	
	Sine	Cosine	Sine	Cosine	Sine	Cosine	Sine	C
1	0	1	0.01745	0.9998	0.0349	0.9994	0.05234	
2	0	2	0.0349	2	0.0698	1.999	0.1047	
3	0	3	0.05236	3	0.1047	2.998	0.157	

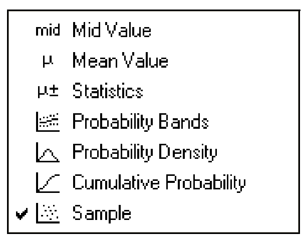
To return to the graph or table of `Cosine` vs. `Degrees`, click in the *XY* checkbox.

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.
3. In the **XY Comparison sources** dialog, check **Use Another variable**.
4. Press the Add... button, and in the **Object Finder**, select the **x** variable. Press **OK** twice.



5. In the **Uncertainty View** popup menu (at the top-left of the **Result** window), select the *Sample* view.

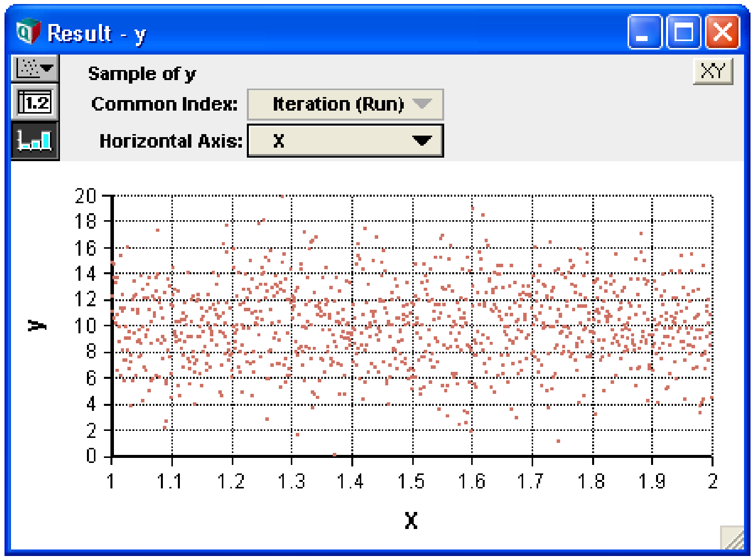
If the variables are independent, the scatter plot points fall randomly on the graph. If the variables are totally dependent, the scatter plot points fall along a single line. The strength of the relationship is indicated by the degree to which the points are close to a line. If the line is straight, the relationship is linear; if the line is curved, the relationship is nonlinear.

You can superimpose several scatter plots of **y** in an array of uncertain quantities depending on **x**. The different quantities are represented by differently colored dots or symbols.

Example

```
x: Uniform(1, 2)
y: Normal(10, 3)
```

The resulting scatter plot of two independent variables is shown below.



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** (page 268), 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 Logistic, probit, and poisson regression functions are in an add-in library, **`Generalized Regression.ana`**, and require Analytica Optimizer. These generalized regression functions are described in the *Analytica Optimizer* manual.

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:

$$y = \sum_k a_k b_k(\grave{x})$$

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**. Any datapoint having **`y=Null`** is ignored.

With the generalized form of linear regression, it is possible to have several independent variables, and your basis set might 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**.

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 might be additional sources of uncertainty because the future might be different from the past. These functions estimate uncertainty due to noise and imperfect fit to the historical data. You might 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 = Sum(c*b, k) + 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

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, s*)

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 = Sum(c*b, k) + 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

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 = Sum(c*b, i) + 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.

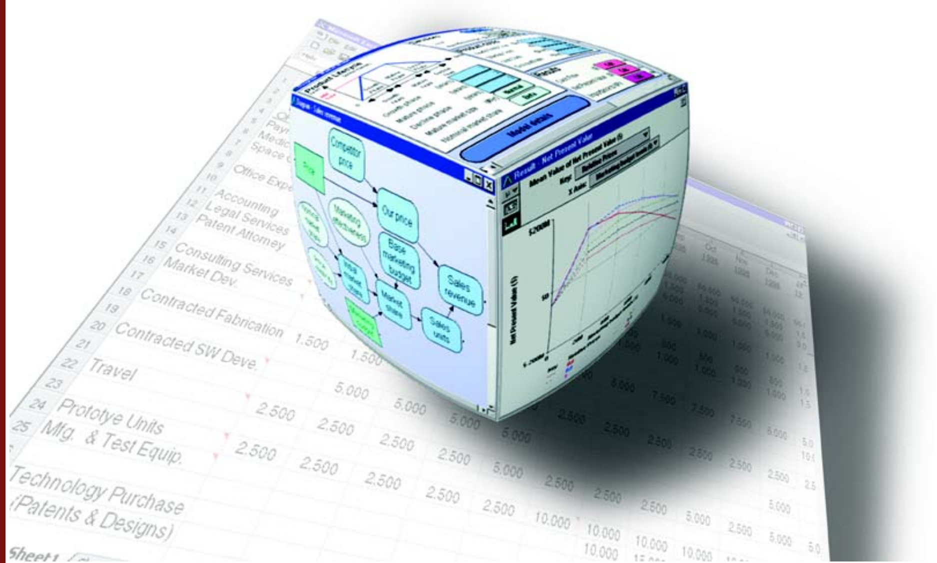
Library Multivariate Distributions

These functions express uncertainty in the coefficients of a linear regression. If you are using results form 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.

Chapter 17 *Dynamic Simulation*

This chapter shows you how to use the system function **Dynamic()** and the system variable **`Time`**.



A ***dynamic variable*** is a quantity that changes over time — for example, the effect of inflation on car prices over a ten-year period. The system function **Dynamic()** and system variable **`Time`** enable you to model changes over time.

Tip Read Chapter 11, "Arrays and Indexes," before using these features.

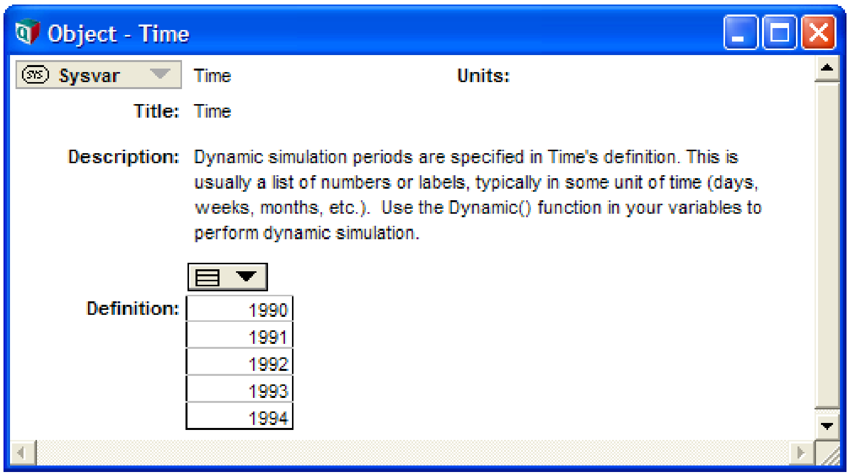
The term *dynamic* is used in this chapter to refer to the **Dynamic()** function.

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 might 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.

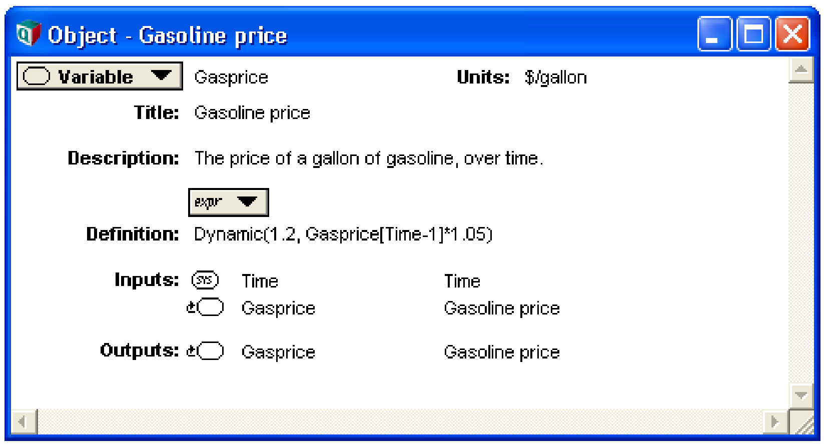
When a dynamic variable refers to itself, it appears in its own list of inputs and outputs, with a symbol for cyclic dependency: .

Library Special

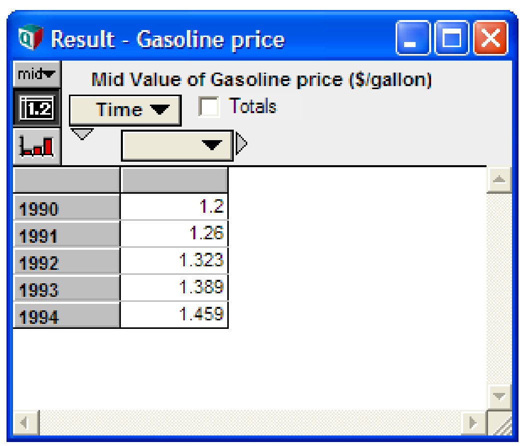
When to use Use **Dynamic()** for defining variables that are cyclically dependent. This is the only function in Analytica that permits reference to the same variable, or other dynamic variables, at earlier time periods.

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.



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**, can be an expression, and might 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

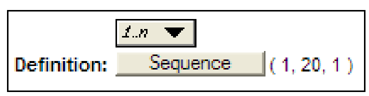
There are three ways to define the **Time** index, each of which has different advantages:

- 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 "Expression view" on page 164 and "Defining a variable as an edit table" on page 169). The numeric values for **Time** can be used in other expressions.

Example



Time as a list (numeric)

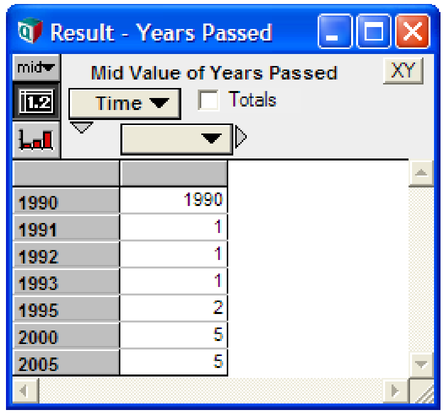
When **Time** is defined as a numeric list, it usually consists of increasing numbers. The intervals between entries can be unequal, and the values for **Time** can be used in other expressions.

Example **Time**

1990
1991
1992
1993
1995
2000
2005

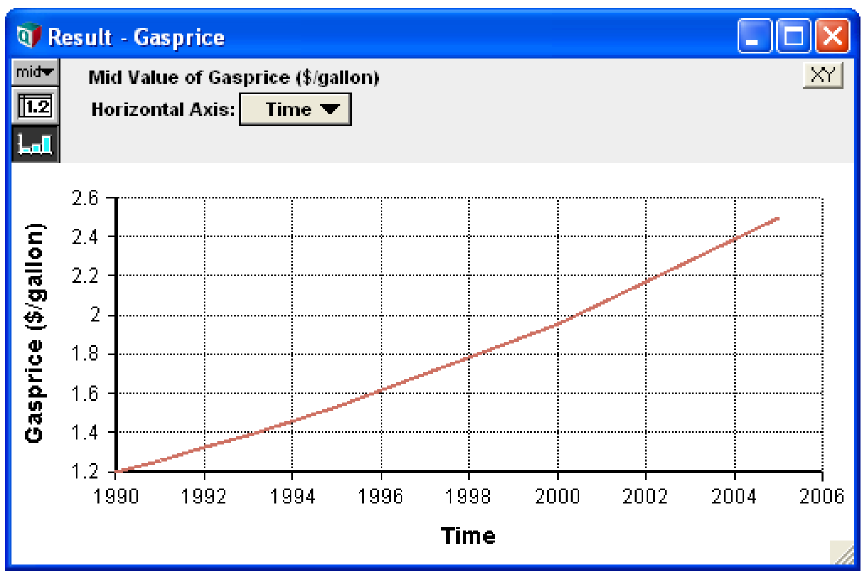
When you use time periods that differ by a value other than 1, typing **(Time-1)** won't provide the value of the previous time period. You can use the syntax **x[Time-1]** if you want to utilize a variable indexed by **Time**, but if you want to perform an operation that depends on the difference in 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 on page 283 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)

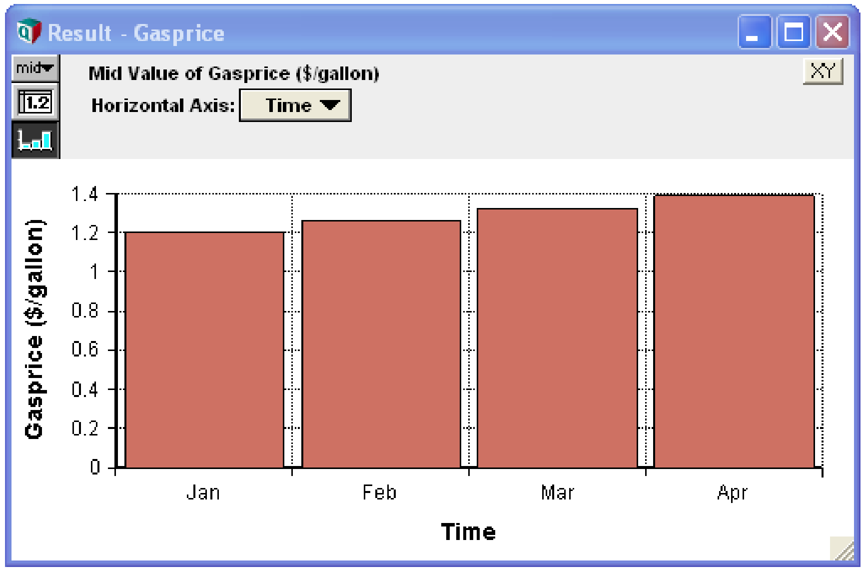
When **`Time`** is defined as a list of labels, **`Time`** values cannot be used in other expressions as numbers.

The resulting graph of any **Dynamic()** function, with the *x*-axis set to **`Time`**, shows 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” on page 55).

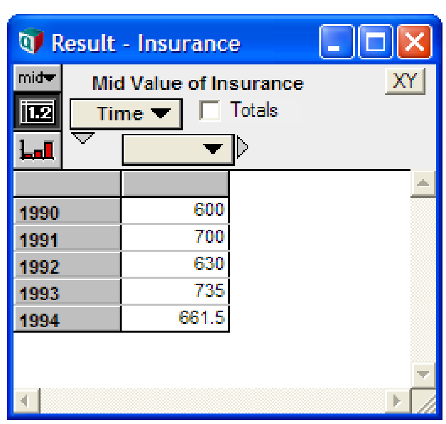
When the **`Time`** node displays on a diagram, arrows from **`Time`** to all dynamic variables display by default.

Initial values for Dynamic

A dynamic definition of **`var`** usually includes the expression **`Self[Time-k]`** or **`var[Time-k]`**, where *k* is the number of time periods to subtract from the current **`Time`** value. You must supply at least *1* initial value.

As an example, when *k* in **`[Time-k]`** is greater than 1, suppose your car insurance policy depends on the premium you paid two years ago. To calculate your payments in 1992, you must refer to the amount paid in 1990. A dynamic variable representing such a rate for insurance needs two initial values for **`Time`**, such as:

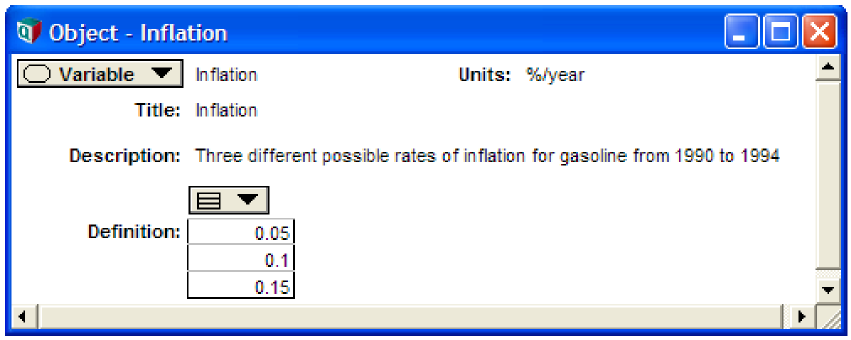
```
Insurance:
Dynamic(600, 700, Insurance[Time - 2] * 1.05) →
```



Using arrays in Dynamic()

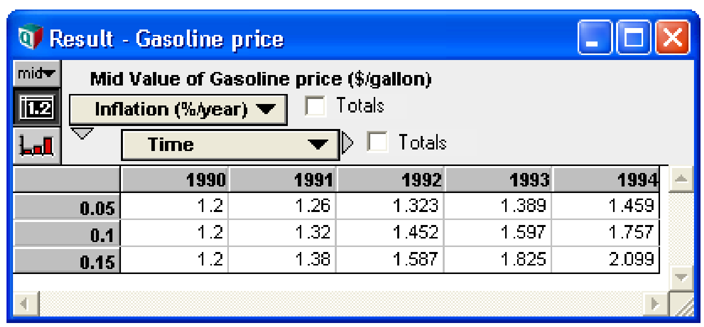
The initial value of a dynamic variable — that is, the first parameter to the **Dynamic()** function — can be a number, variable identifier, or other expression that evaluates to a single number, list, or array. Analytica evaluates a dynamic variable starting from each initial value, in each time period, so the result is a correctly dimensioned array.

Example Expanding the example (see "Using the Dynamic() function" on page 282), 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)) →
```

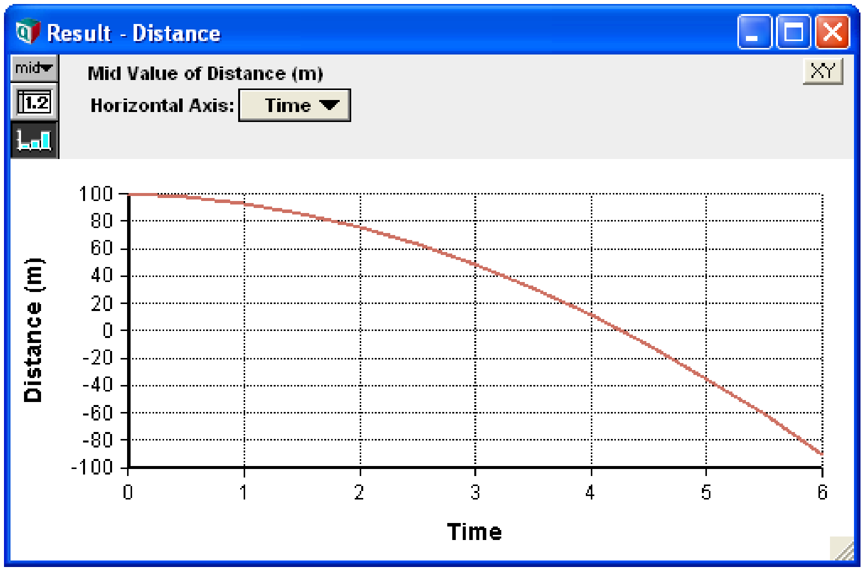


Dependencies with Dynamic

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** (page 78).
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.

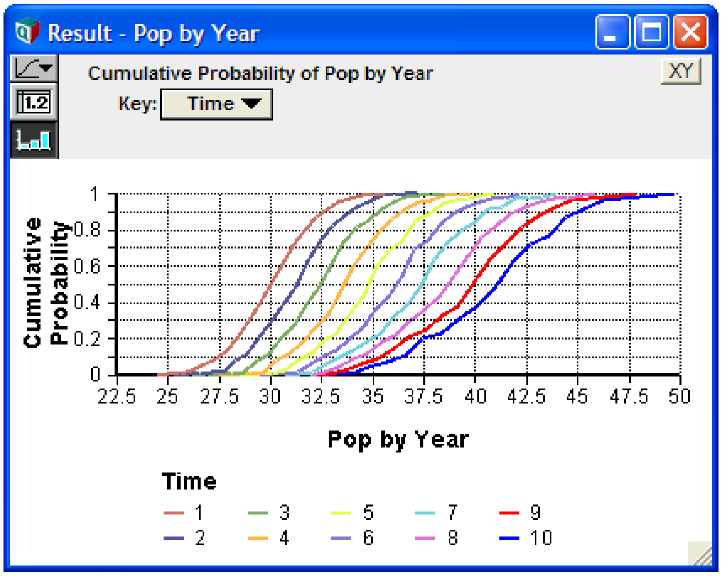
If you attempt to operate over the **`Time`** dimension from within a dynamic loop, Analytica issues the warning: *"Encountered application of an array function over the Time index from within a dynamic loop. The semantics of this operation might be different than you expect."*

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:

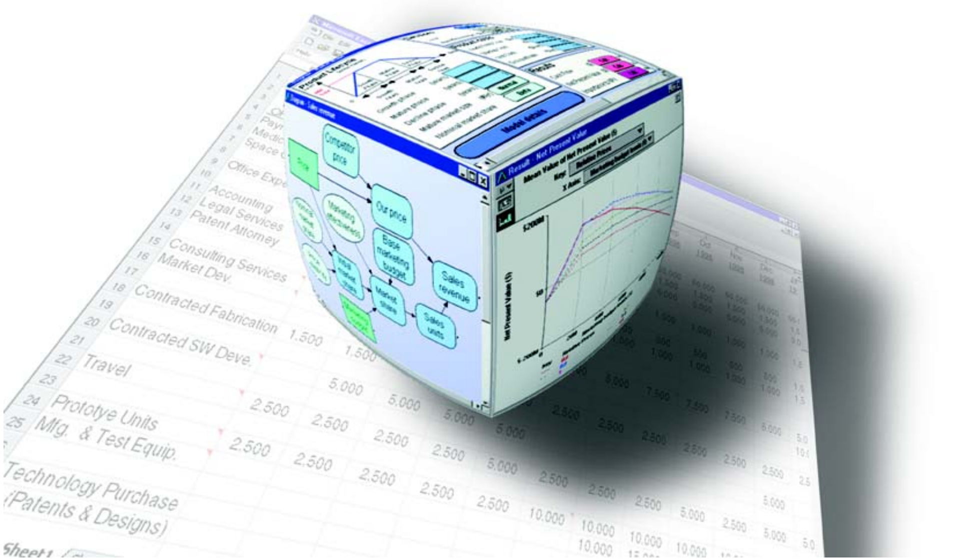
```
Birthrate:= Dynamic(Normal(1.2, .3), Normal(1.2, .3))
Pop_by_year:= Dynamic(Population, Self[Time-1] +
Birthrate)
```

Chapter 18 *Importing, Exporting, and OLE Linking Data*

OLE linking makes it possible to link data to and from external applications. With OLE linking, changes to inputs or results are automatically and instantaneously propagated between applications.

This chapter describes how to exchange data between Analytica and other applications. The primary methods are:

- Using the standard Copy and Paste commands
- Using OLE linking
- Using the Import and Export commands



Copying and pasting

You can use the standard **Copy** and **Paste** commands with any modifiable attribute of a variable, module, or function.

Pasting data from a spreadsheet

To paste tabular data from a spreadsheet into an Analytica table:

1. Select a group of cells in a spreadsheet.
2. Select **Copy** from that program's **Edit** menu (*Control+c*), 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** (page 132) 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.

Exporting to an image file

To export an influence diagram to an image file, with the diagram showing select **Export** from the **File** menu. From the **Save** dialog, select the desired format (e.g., EMF, PNG, JPEG). An image of the full diagram is stored (not just the selected nodes).

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" on page 300).

Copying a result graph

To copy a result graph:

1. Open the **Result** window containing the graph.
2. Select **Copy** from the **Edit** menu (*Control+c*) to copy an image representation of the graph to the clipboard.

Exporting a result graph to an image file

To export a result graph:

1. Open the **Result** window containing the graph.
2. Select **Export** from the **File** menu and select the desired image file format (e.g., EMF, PNG, JPEG).

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 either updates automatically, or if you prefer, when you explicitly decide to update the data.

You can link any of the result table views (i.e., Mid, Mean, Statistics, Probability Density, Cumulative Probability, and Sample table views). You can link any two-dimensional slice of a multi-dimensional table with the regular **Copy** command. For result tables with more than two dimensions, you might decide to link the entire table as a series of two-dimensional tables using the **Copy Table** option from the **Edit** menu. You can also link a rectangular region of cells that are a subset of a a two-dimensional table. However, you cannot link non-table data such as the information that is contained in the **Object** window or **Attribute** panel.

Linking procedure

Steps for linking result data from your Analytica model to an external OLE-compliant application are as follows. For concreteness, we'll assume here that the other application is Microsoft Excel.

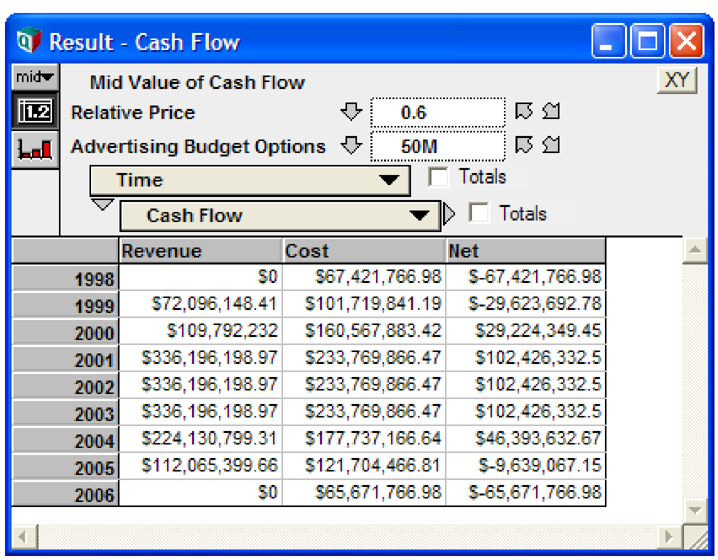
1. In the Analytica **Result** window, select the cells you want to link and choose **Copy** from the **Edit** menu (*Control+c*).
2. From Excel, select the cells where you would like the Analytica data linked.
3. From Excel, choose **Paste Special** from the **Edit** menu.
4. The **Paste Special** dialog appears.
5. In this box, choose the option **Paste Link**, select **Text** from the **As** list, and click **OK**.

You're done. Any changes to the source result table are propagated to the linked data in Excel. The procedure for linking Analytica model results to other OLE-compliant applications is 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 do not display "Text" as an option in Step 5 above, or disable the **Paste Special** menu item in Step 3.

Detailed example of linking Analytica results

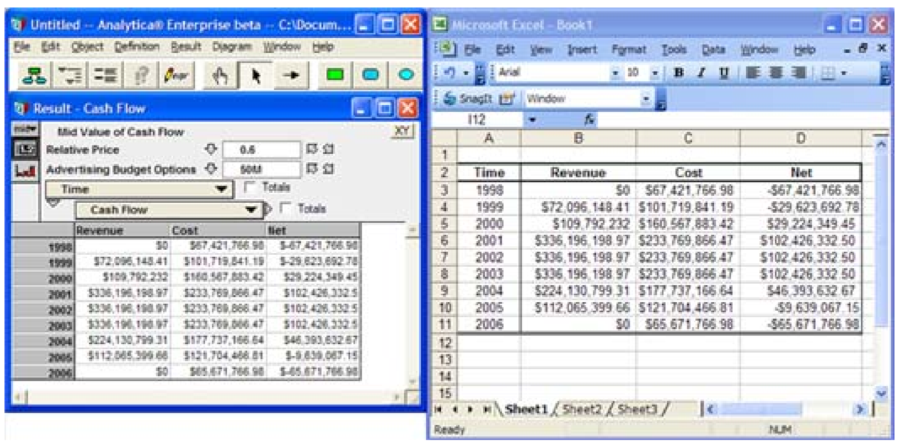


This example itemizes detailed steps for linking an Analytica result table into an Excel spreadsheet. Suppose you would like to link the model results displayed above into an Excel spreadsheet. You can start by linking the column and row headers. Go to the node titled *Cashflow Category* and evaluate its result. Notice the result of node *Cash Flow Category* is displayed as a column of cells, but you would like to have them linked into Excel as a row. Unfortunately you cannot link this data as a row with a single Copy/Paste Special operation since Excel does 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 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 is preserved in fixed point format, but you might want to use Excel formatting to get the dollar sign and thousand separator displayed. Excel might 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 are reflected in your linked table in Excel.



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 can 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 can 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, uncheck the checkbox 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 is 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 converts 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 is likely 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 occur:

1. A new instance of Analytica launches.
2. Analytica loads the model.
3. Analytica evaluates the variables upon which the links are dependent.
4. The links reactivate.
5. The linked data updates.

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 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, select the links you would like to refresh and click the **Update** button.

Linking data from other applications into Analytica

Using OLE linking, you can 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 can 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 cannot 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

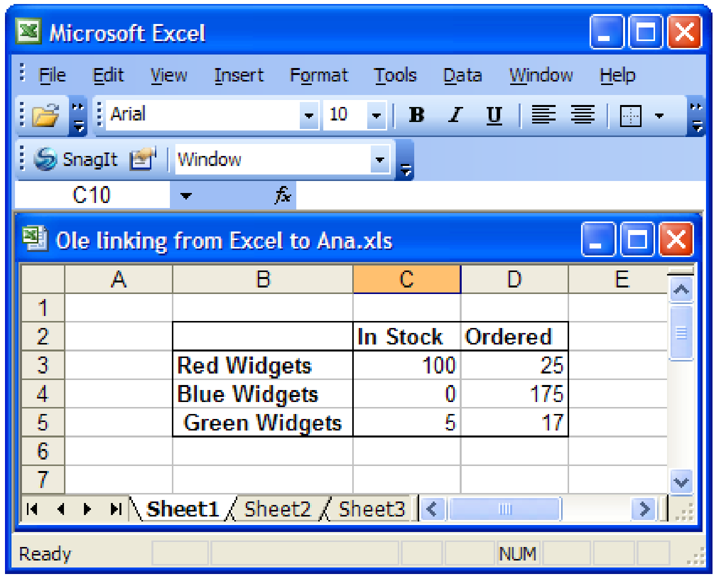
Steps for creating a linked edit table in Analytica with data from an Excel spreadsheet:

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**.
   The **Paste Special** dialog appears.
4. In this box, choose the option **Paste Link**, select **Text** from the **As** list, and click **OK**.

The process for linking data from Word or other OLE-compliant applications are analogous to the steps just outlined.

Example of linking a table into Analytica

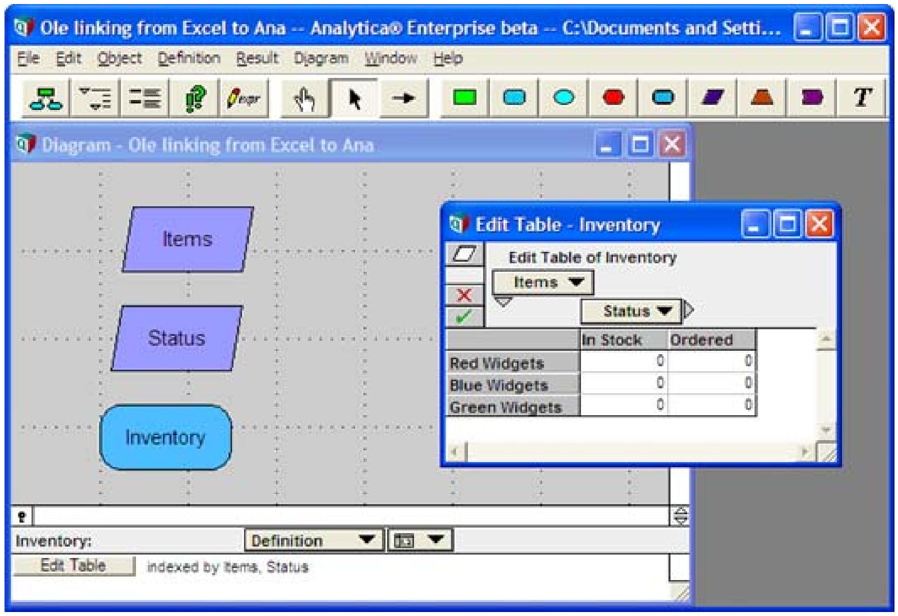
This section itemizes detailed steps for linking a table from Excel into Analytica by creating a node with a “Linked Table” definition. Specifically, suppose you desire to link the Excel table displayed in the following figure into Analytica.



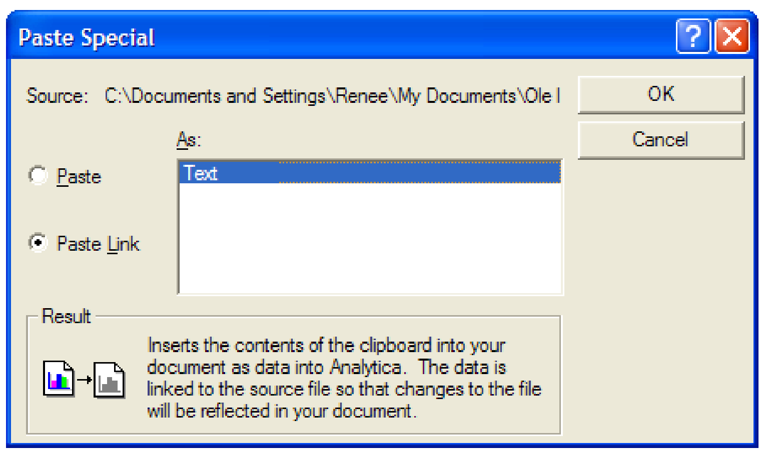
Start by creating two indexes in Analytica to store the row and column headers. Title the first index *Items* and the second *Status*. Select the node *Items* and then click the **Show definition** button on the toolbar (this is the button with the pencil icon) or right mouse menu. In the **Attribute** 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 gives 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 appears and you need only select the *Transpose* checkbox 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.

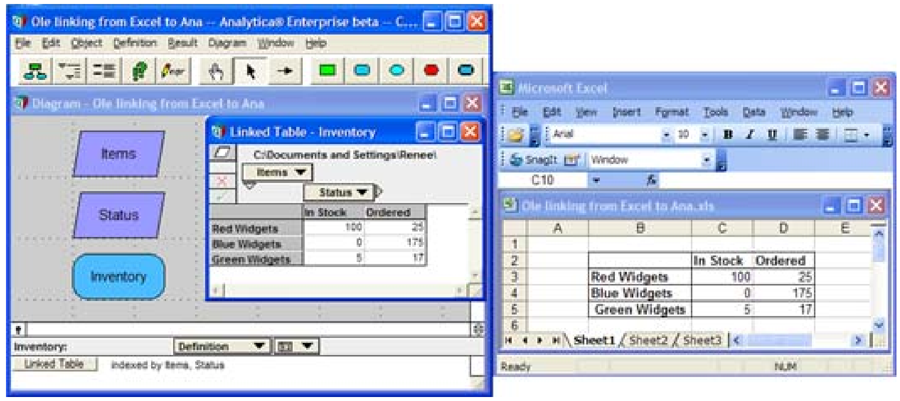
Since you’ve finished creating the indexes, you’re ready to start on the node that contains the linked table. Create a variable node in Analytica and title it *Inventory*. With this node selected, click the **Show definition** button on the toolbar. In the **Attribute** panel or **Object** window that appears, click the **expr** popup menu and choose **Table**. The **Indexes** dialog appears. In this dialog, select *Items* and click the ▼ button. This moves *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 **OK** and an edit table appears as follows.



Go to Excel and select the numerical values displayed in the table and choose **Copy** from the **Edit** menu (*Control*+*c*). 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 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 changes accordingly.



Tip The data within the table is linked and is updated automatically when altered, but the row and column headers are not linked and any changes to their values must 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 can change this setting to “manual.” This might 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 might 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.

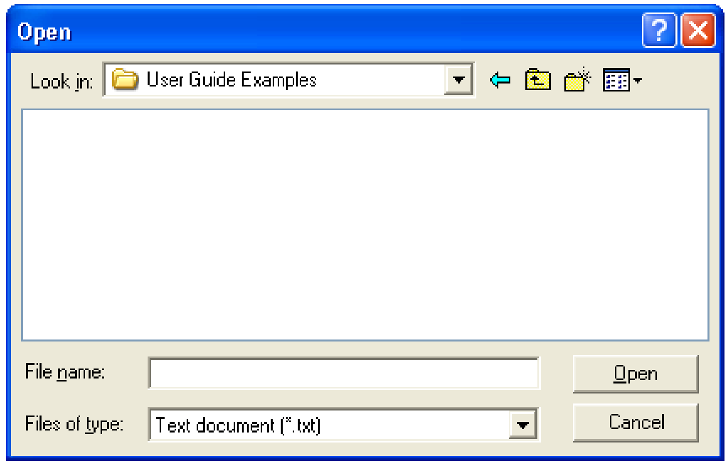
Activating the other application If you have linked data from an external application into Analytica, after loading Analytica you can make the other application visible using the **Open Source** button on the **OLE Links** dialog, accessed through the **Edit** menu. If you implement a portion of your 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 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 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 300). 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

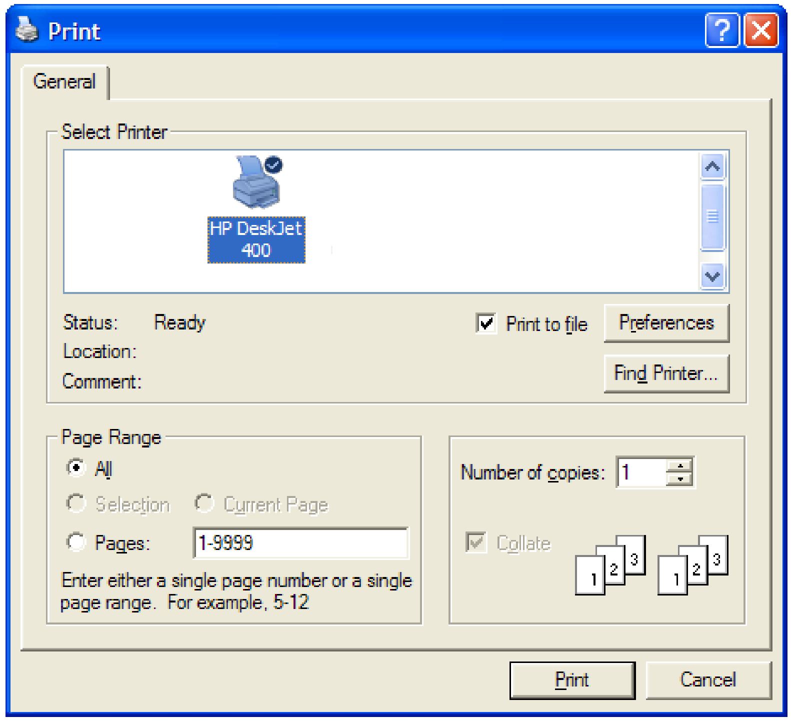
To export a variable's result table to a text file, first be certain that the text file is closed.

1. Select the variable to be exported from and open its **Result** window.
2. Select **Export** from the **File** menu. A dialog 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 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>
<Index identifier><line break>
<Index value><tab><Array value><line break>
```

Example

Keyword Attribute Variable identifier

```
TextTable Definition House_cost_inputs
   House_inputs
PropTax         3400
Tax rate        0.44
Maintenance     4000
Interest        0.105
Appreciation    0.08
```

Index identifier

Index values Array values

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

Keyword Attribute Variable identifier

```
TextTable Definition Mortgage
Down_payment      20000 45000  60000
Buying_price
  200000 180000 155000 140000
  400000 380000 355000 340000
  600000 580000 555000 540000
```

Index1

Index2

Index2 values Array values Index1 values

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 --- Array values
                     60000   10073  12157  13555
Index1         ---- Buying_price  400000  ---------- Index1 Value2
                    Years_owned  5 10 15
                    Down_payment
                    20000 10180. 14201. 16867.
                    45000 10160. 14199. 16882.
                    65000 10141. 14198. 16897.
Index1         ---- Buying_price  60000   ---------- Index1 Value3
                    Years owned  5 10 15
                    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” on page 82).

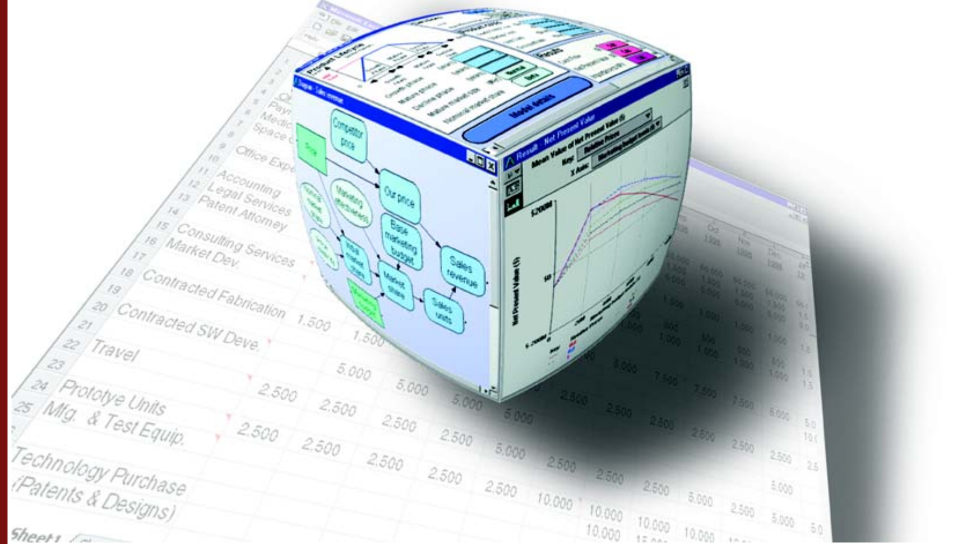
Numerical data is exported in the format set for the table, with these exceptions:

- Suffix format numbers are exported in scientific exponential format.
- Fixed decimal point numbers of more than 9 digits are exported in scientific exponential format.
- If a date format begins with the day of the week, e.g., “Saturday, January 1, 2000”, the weekday is suppressed: “January 1, 2000”.

Chapter 19 *Working with Large Models*

This chapter shows you how to:

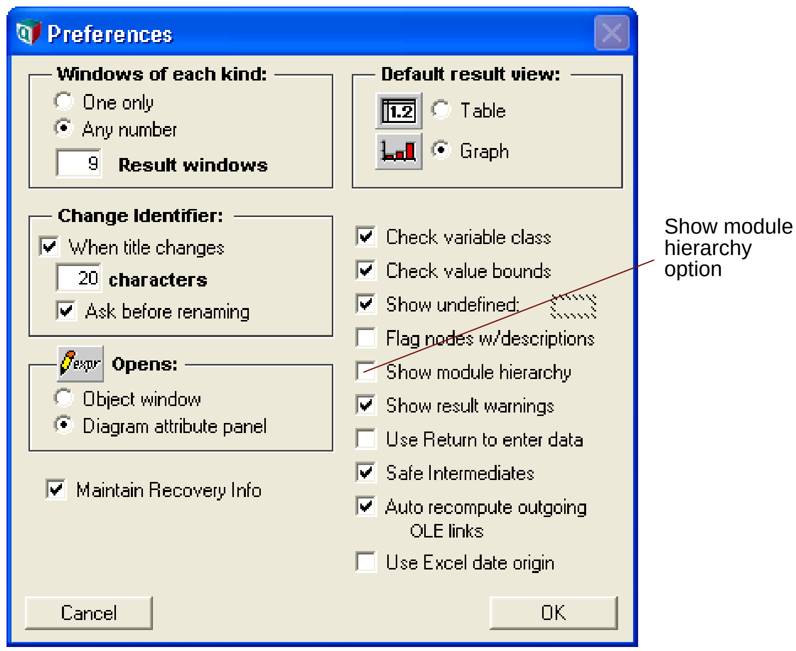
- Navigate large models
- Combine existing models into an integrated model



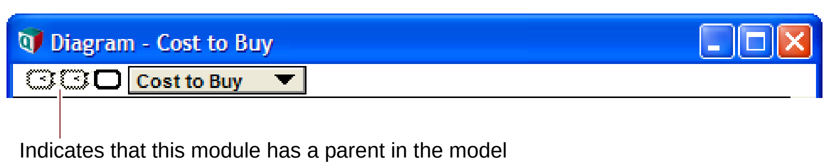
Large models, which include many variables organized into multiple modules at several levels of hierarchy, can be challenging to find your way around. The first part of this chapter describes how to navigate larger models, using the hierarchy preference, the **Outline** window, and variable input and 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.

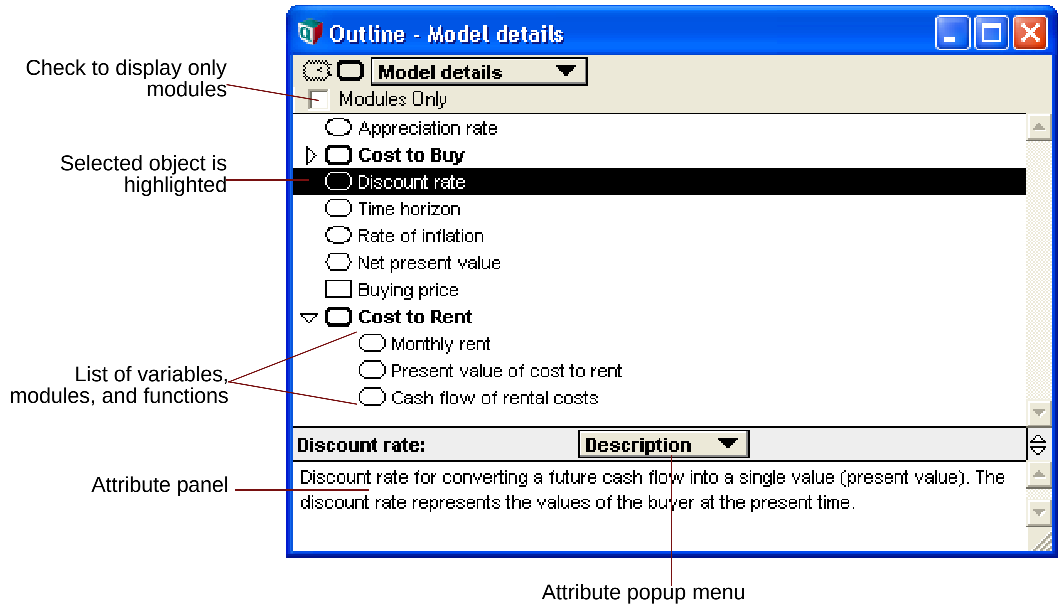


If you check the *Show module hierarchy* box, the top of the active **Diagram** window displays one or more module node shapes to indicate its hierarchy depth.



The Outline window

The **Outline** window displays a listing of the nodes inside a model. It can also show the module hierarchy as an indented list of modules. It provides an easy way to orient yourself in a large model and to navigate within it.



Opening the Outline window

To open the **Outline** window, click the **Outline** button in the toolbar .

The **Outline** window highlights the entry for the selected module or variable.

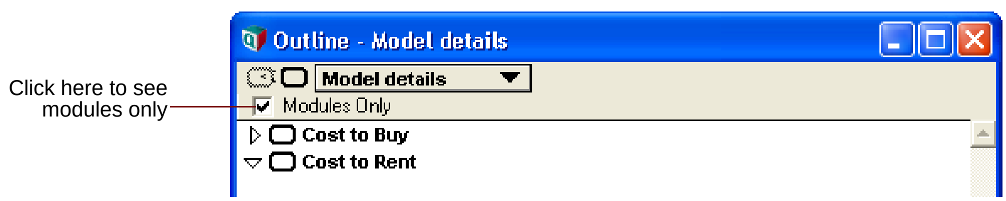
Opening details from an outline

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 ▷ or ▽ to let you display or hide the module's contents.

▷ Indicates that the module's contents *are not* shown in the **Outline** window. Click this icon to display the module's contents.

▽ 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** (page 19).

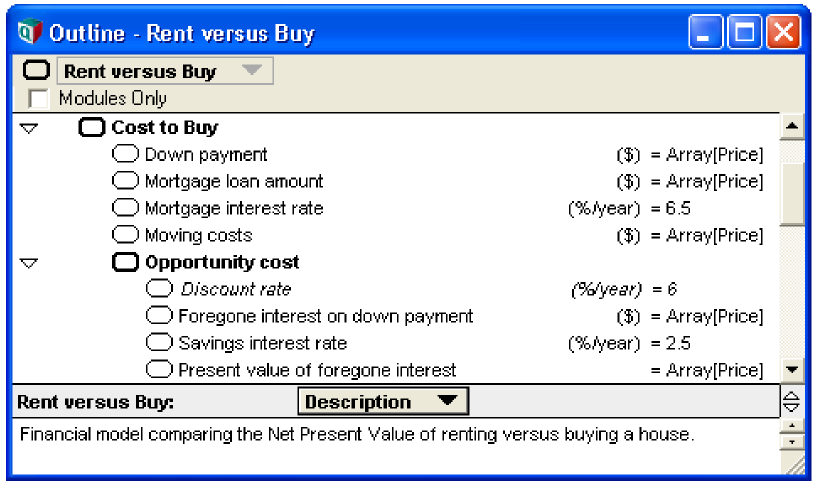
To view the attributes of a listed node:

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 108).

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** (page 26) from the **Object** menu. Each variable whose mid value has been evaluated and is an atom displays in the window.

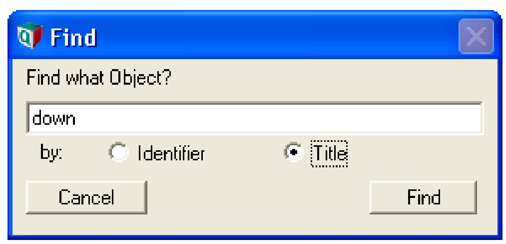


Finding variables

To locate a variable in its diagram, by identifier or by title, use the **Find** dialog.

Find dialog To display the **Find** dialog:

1. Select **Find** from the **Object** menu (*Control+f*).



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 is displayed.

To find the next object that is a partial match to the last identifier or title that you entered, select **Find Next** from the **Object** menu (*Control+g*).

To find an object whose identifier matches the selected text in an attribute field (such as a *definition* field), select **Find Selection** from the **Object** menu (*Control+h*).

Managing attributes

Every node in an Analytica model is described by a collection of ***attributes***. For some models, you might want to control the display of attributes or create new attributes. Some attributes apply to all classes (variable, module, and function). Others apply to specific classes, as listed in the following table.

Attribute	Function	Module	Variable
Author		*	
Check	+		+
Class	*	*	*
Created		*	
Definition	*		*
Description	*	*	*
Domain			+
File Info		*	
Help	+	+	+
Identifier	*	*	*
Inputs	+		+
Last Saved		*	
MetaOnly			+
Outputs	+		+
Parameters	*		
Probvalue			+
Recursive	+		
Title	*	*	*
Units	*		*
Value			+
User-created (up to 5)	+	+	+

Key:

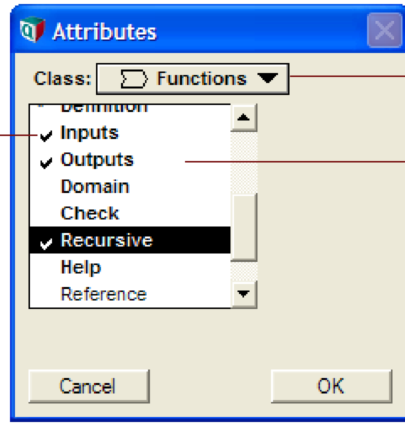
plain = modifiable by user	* = always displayed
italic = set by Analytica	+ = optionally displayed

For descriptions of the attributes, see “Glossary.”

Attributes dialog

Use the **Attributes** dialog to control the display of optional attributes in the **Object** window and **Attribute** panel and to define new attributes.

To open the **Attributes** dialog, 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 checkmark (√) 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 checkmark.

To hide an optional attribute, click it once to select it, then click it again to remove the checkmark.

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:

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, and turn the checkmark 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*** might 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
```

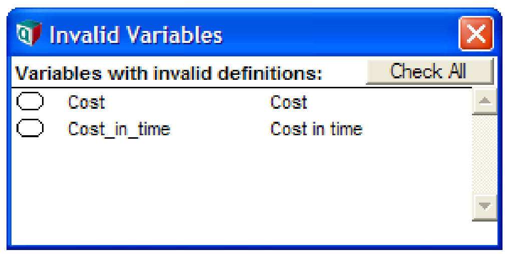
Library Special

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`** does 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

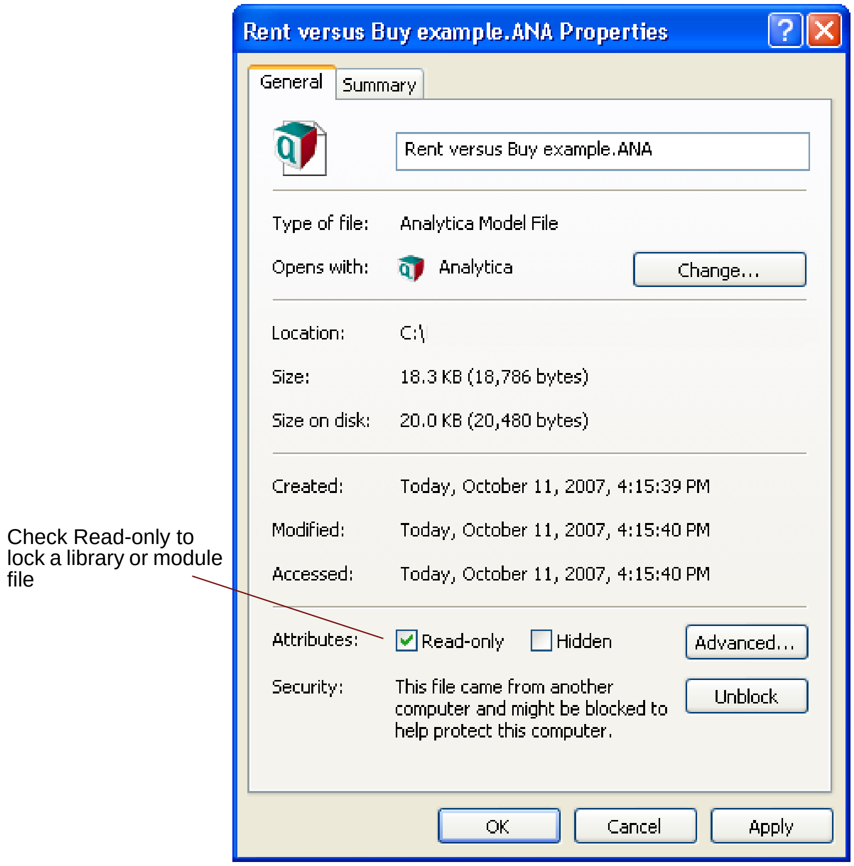
To create a filed module or library:

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 variables in the module, or create them elsewhere and move them into the module.
3. **Change the class** (page 57) of the module to **Module** or **Library**.
4. The **Save As** dialog 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* checkbox.
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** (page 310). 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** (page 310). 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 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 in **`/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 is deleted; a linked original still exists as a separate file.

Saving changes

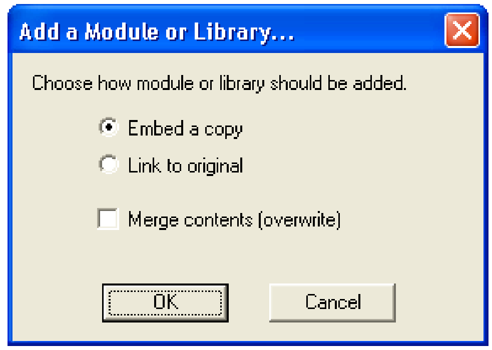
After you have linked to a filed module or library, the **Save** command saves every filed module and library that has changed, as well as the model containing them, in their corresponding files.

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

To add a module or library, select **Add Module** (*Control+I*) or **Add Library** from the **File** menu. The main difference is that Add module starts the file browser by default in the folder you opened the model (or last added module) from, where **Add Library** starts from the standard libraries folder installed when you installed Analytica. Either way, you must be in Edit mode or those options will be grayed out in the **File** menu.

The standard **Open Model** dialog appears. Select the desired module in that dialog. The following dialog 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 overwrite the preferences and uncertainty settings of the root model.

An added module or library can 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 do 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 are 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 checkbox 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 checkbox, and an object in the file being added has the same identifier as one in the active model, Analytica points that out and asks if you want to rename the variable. If you click **Yes**, it renames the variable in the *existing* model, and updates all definitions in the existing model to use the changed identifier. It leaves unchanged the identifier of the variable in the module it is adding (which might contain definitions referencing that identifier that it has yet to read.) Hence, all the definitions in the existing model and added module continue to reference the correct (original) variables.

Combining models into an integrated model

Large models introduce a unique set of modeling issues. Modelers might want to work on different parts of a model simultaneously, or at remote locations. During construction, a large model might 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 has 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 the **Add Module dialog** (page 310) 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" on page 309.

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 the **Add Module dialog** (page 310), add in the sub-models. Be sure to check the *Merge* option in the **Add Module** dialog. 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 are overwritten by the output nodes, thus linking the modules and avoiding duplication.

With identical identifiers, the individual modules cannot be evaluated alone because they are missing their input data. They can be evaluated only as part of the integrated model.

Redundant nodes Place the output node identifiers in the definition fields of their respective input nodes. Due to the node redundancy, this method requires more memory than using identical identifiers, and it is therefore less desirable when large tables of data are passed between modules. However, since no nodes are overwritten and lost upon integration, this method preserves the modules' structural integrity, with both input and output nodes visible in each module's diagram.

With redundant nodes, each module can be opened and evaluated alone, using stand alone shells.

Stand alone shells With redundant nodes, you can create a top-level model that contains one or more modules and the public variables module plus dummy inputs and outputs. Such a top-level model is called a ***stand alone shell*** because it allows you to open and evaluate a single module "standing alone" from the rest of the integrated model. Stand alone shells are useful when modelers want to examine or refine a particular module without the overhead of opening and running the entire model.

To create a stand alone shell for module **`Mod1`**, which is a filed module:

1. Open the integrated model and evaluate all nodes that feed inputs to **`Mod1`**.
2. Use the **Export** command (see “Importing and exporting” on page 298) to save the value of each feeding node in a separate file. Make a note of these items:
   - The identifier of each node and the indexes by which its results are dimensioned.
   - The identifiers of **`Mod1`**’s output nodes, if you want to include their dummies in the stand alone shell.
3. Close the integrated model.
4. Create a new model, to be the stand alone shell.
5. Use **Add Module** to add the public variables module.
6. For each input node, create a node containing an edit table, using the identifier and dimensions of the feeding nodes you noted from the integrated model.
7. Use the **Import** command (see “Importing and exporting” on page 298) 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 is usable within the integrated model.

Cautions in combining models

Identifiers Every object in a model must have a unique identifier. The identifiers of filed libraries and filed modules that you add to a model, as well as their variables and functions, cannot duplicate identifiers in the root model. See “Merge contents (overwrite)” on page 311.

Created attributes When you combine models with created attributes, the maximum number of defined attributes is five (see “Managing attributes” on page 306).

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:

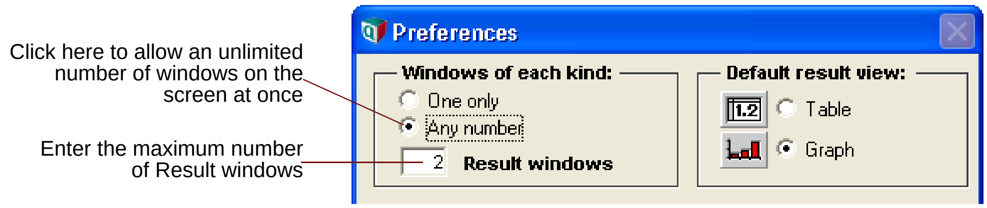
- The top-level **Diagram** window and not more than one **Diagram** window for each lower level in the hierarchy
- One **Object** window
- Two **Result** windows

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 while you click or double-click to open the new window.

- Use the **Preferences dialog** (page 58) to change the limits. Select **Preferences** from the **Edit** menu.



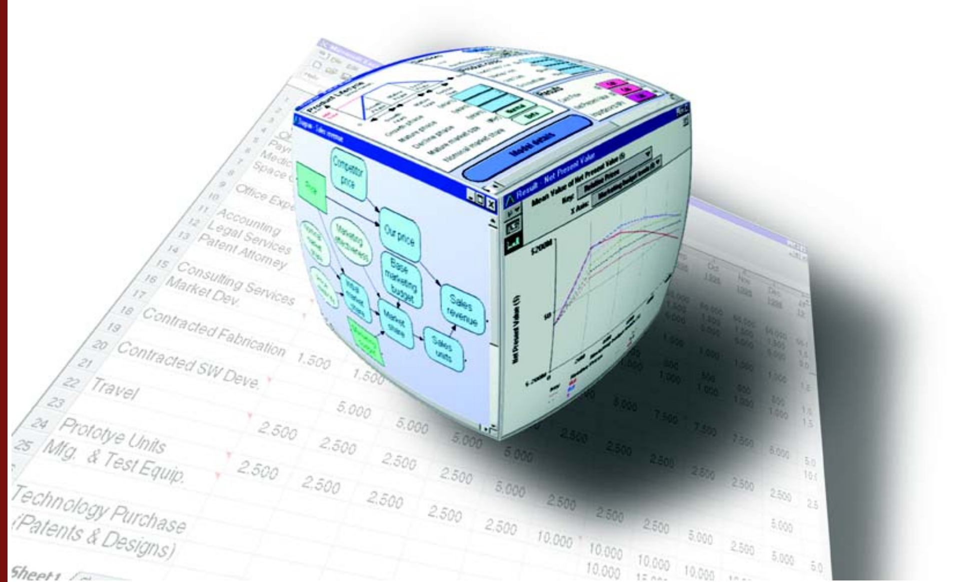
In the *Windows of each Kind* area, select *Any number* instead of *One only*.

To display more **Result** windows and keep the limit on **Diagram** and **Object** windows, enter the maximum number of **Result** windows.

Chapter 20 Building Functions and Libraries

This chapter shows you how to:

- Use functions
- Create your own functions
- Work with parameter qualifiers
- Create your own function 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***.

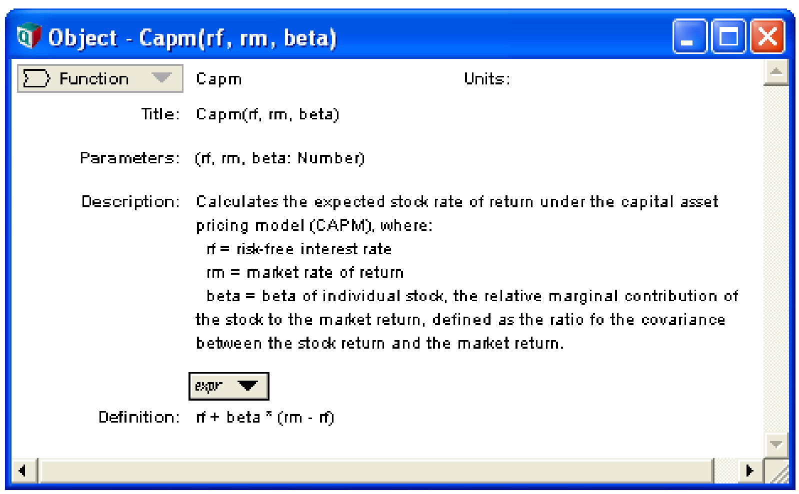
A ***library*** is a collection of user-defined functions grouped in a library file, for use in more than one model. Using libraries, you can effectively extend the available functions beyond those built in to Analytica. Analytica is distributed with an initial set of libraries, available in the **Libraries** folder inside the Analytica folder on your hard disk. If you add a library to a model, it appears with its functions in the **Definition** menu, and these functions appear almost the same as the built-in functions.

You might want to look at these libraries to see if they provide functions useful for your applications. You might also look at library functions as a starting point or inspiration for writing your own functions.

Analytica experts can 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 It has three parameters, **rf**, **rm**, and **beta**. The parameter qualifier **Number** says that it expects that the parameters are numbers.

Description The description says what the function returns and what its parameters mean.

Definition The definition is an expression that uses its parameters, **rf**, **rm**, and **beta**, and evaluates to the value to be returned.

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)
```

The function works equally well when **`StockBeta`** is an array of beta values — or if any parameter is an array — the result is an array of expected returns.

Using a function

Position-based calling 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)
```

This evaluates function **`Capm(Rf, Rm, Beta)`** with **`Rf`** set to 5%, **`Rm`** set to 8%, and **`Beta`** set to **`Stockbeta`**.

Name-based calling Analytica also supports a more flexible *name-based* calling syntax, identifying the parameters by name:

```
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. You can give the parameters in any order. They must include all required parameters. This method is much easier to read when the function has many parameters. It is especially useful when many parameters are **optional** (page 321).

You can mix positional and named parameters, provided the positional parameters come first:

```
Fu1(1, 2, D: 4, C:3)
```

You cannot give a positional parameter after a named parameter. For example, the following entry displays an error message:

```
Fu1(1, D: 4, 2, 3) Invalid
```

This *name-based calling syntax* is analogous to Analytica's *name-based subscripting* for arrays to obtain selected elements of an array, in which you specify indexes by name. You don't have to remember a particular sequence to write or understand an expression. See "x[i=v]: Subscript construct" on page 174.

Tip Name-based calling syntax works for all user-defined functions. It also works for most of the built-in functions, except for a few with only one or two parameters.

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. It shares many of the attributes of variables, including identifier, title, units, description, and definition, inputs, and outputs. It has a 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 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: Number)
```

The parameters can have type qualifiers, such as **`Number`** above (see the next section).

You can help make functions easier to understand and use by giving the parameters meaningful names, in a logical sequence. The parameters appear in the **Object Finder** dialog. When you select a function from the **Definition** menu, it copies its name and parameters into the current definition.

Description The description should describe what the function returns, and explain each of its parameters. If the definition is not immediately obvious, a 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.

Recursive Set to 1 (true) if the function is recursive — that is, it calls itself. This attribute is not initially displayed. Use the **Attributes** dialog from the **Object** menu to display it. See "For and While loops and recursion" on page 331.

Parameter qualifiers

Parameter qualifiers are keywords you can use in the list of parameters to specify how, or whether, each parameter should be evaluated when the function is used (called), and whether to require a particular type of value, such as number or text value. 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, or can be repeated. 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: Atom Text Optional ="NA")
```

It defines five parameters. **a** should be an array of numbers, indexed by parameters **i** and **j**, and optionally other indexes. **i** and **j** must be index variables. **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** is an **`Atom Text`**, meaning that it is reduced to a single text value each time the function is called, and is optional. If omitted it defaults to **`"NA"`**. 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:

```
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`** evaluates **`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 determinstically, 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**.

Atom **Atom** specifies that the parameter must be an atom — a single number, text, or other value not an array — *when the function is evaluated*; *but the actual parameter* can *be an array when you call the function.* If it is an array when you call the function, Analytica disassembles it into atoms, and evaluates the function separately on each atomic element of the array. After these evaluates, it reassembles the results into an array with the same indexes as the original parameter, and returns is returned as the overall result.

You need to use **Atom** only when the function uses one of Analytica's few constructs that require an atomic parameter or operand — i.e., that does not fully support array abstraction. See "Ensuring array abstraction" on page 336.

You might be tempted to use **Atom** to qualify parameters of every function, just in case it's needed. We strongly advise you not to do that: Functions with **Atom** parameters can take *much* longer to execute with array parameters, because they have to disassemble the array-valued parameters, execute the function for each atom value, and reassemble them into an array. So, avoid using it except when really necessary.

Scalar The parameter expects a single number, not an array. Means the same as **Number Atom**.

Array [i1, i2...] Specifies that the parameter should be an array with the designated index(es) when it the function is evaluated. Similar to **Atom** above, you can still call the function with the parameter as an array with indexes in addition to those listed. If you do, it disassembles the array into subarrays, each with only the listed indexes. It calls the function for each subarray, so that **a** is indexed only by the specified index(es). For example, if **`Fu1`** has the parameter declaration:

```
Function Fu1(a: Array[Time])
```

and if **a**, when evaluated, contains index(es) other than **`Time`**, it iterates over the other index(es) calling **`Fu1`**, for each one, and thus ensuring that each time it calls **`Fu1`**, parameter **a** has no index other than **`Time`**.

An array declaration can specified zero or more indexes between the square brackets. With zero indexes, it is equivalent to the qualifier **`Atom`**, specifying that the parameter must be a single value or atom each time the function is called.

The square brackets are sufficient and the qualifier word **`Array`** is optional, so you could write simply:

```
Function F(a: Number [I])
```

instead of

```
Function F(a: Number Array [I])
```

Each index identifier listed inside the brackets can 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 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:

```
x: All [i, j]
```

Here 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`** converts it into an array with indexes, **i** and **j**, repeating the value of **`x`** in each element. Without **`All`** Analytica would simply pass the atomic 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, including **`INF`**.
Nonnegative	Zero, or a number greater than zero including **`INF`**.
Text	A text value.
Reference	A reference to a value, created with the **`\`** operator.
Handle	A handle to an Analytica object, obtained from the **Handle** or **HandleFromIdentifier** functions. It also accepts an array of handles.
OrNull	Used in conjunction with one of the above type qualifiers, allows Null values in addition to the given type. For example: `x: Number OrNull` Some array functions ignore Null values, but require this qualifier for the null values to be accepted without flagging an error.

Coerce If you accompany a Type checking qualifier by the **`Coerce`** qualifier, it tries to convert, or *coerce*, the value of the parameter to the specified type. For example:

`a: Coerce Text [I]`

tries to convert the value of **`a`** to an array of text values. It gives an error message if any of the coercions are unsuccessful.

`Coerce` supports these conversions:

From	**To**	**Result**
`Null`	Text	"Null"
Number	Text	Number as text, using the number format of the variable or function calling the function.
Text	Number or Positive	If possible, interprets it as a date or number, using the number format.
`Null`	Reference	**`Null`**
Number	Reference	\X
Text	Reference	\Text

Other combinations, including **`Null`** to **`Number`**, give an error message that the coercion is not possible.

Ordering qualifiers: Ascending and Descending

The ordering qualifiers, **`Ascending`** or **`Descending`**, check that the parameter value is an array of numbers or text values in the specified order. 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, it gives an evaluation error.

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, for example:

```
A: Ascending [I]
```

Optional parameters

You can specify a parameter as optional using the qualifier **Optional**, for example:

```
Function F(a: Number; b: Optional Number)
```

In this case, you can call the function without mentioning **b**, as:

```
F(100)
```

Or you can specify **b**:

```
F(100, 200)
```

You can specify a default value for an optional parameter after an **=** sign, for example:

```
Function F(a: Number; b: Number Optional = 0)
```

It uses the default value if the actual parameter is omitted. Given an equal sign and default value, the **`Optional`** qualifier is itself optional (!):

```
Function F(a: Number; b: Number = 0)
```

Optional parameters can appear anywhere within the declaration — they are not limited to the final parameters. For example, if you declare the parameters for **G** as:

```
Function G(A: Optional; B; C: Optional; D; E: Optional)
```

You can call **G** in any of these ways:

```
G(1, 2, 3, 4, 5)
G(1, 2, , 4)
G( , 2, , 4)
G( , 2, 3, 4, 5)
```

Generally, you must include the commas to indicate an omitted optional parameter, before any specified parameter, but not after the last specified parameter.

Or you can use named-based calling syntax, which is usually clearer and simpler:

```
G(B: 2, D: 4)
```

IsNotSpecified(v) If you omit a parameter that is not given a default value, you can test this inside the function definition using function **IsNotSpecified(v)**. For example, the first line of the body of the function might read:

```
If IsNotSpecified(a) then a := 0;
```

But it is usually simpler to specify the default value in the parameter list as:

```
Function H(x;, a : = 0)
```

Repeated parameters (...)

Three dots, "**...**" qualifies a parameter as repeatable, meaning that the function accepts one or more actual parameters for the formal parameter. For example:

```
Function ValMax(x: ... Number) := Max(x)
ValMax(3, 6, -2, 4) → 6
```

ValMax() returns the maximum value of the actual parameters given for its repeated parameter, **x**. Unlike the built-in **Max()** function, it doesn't need square brackets around its parameters.

During evaluation of **ValMax()**, the value of the repeated parameter, **x**, is a list of the values of the actual parameters, with implicit (**Null**) index:

```
[3, 6, -2, 4]
```

ValMax() can also take array parameters, for example:

```
Variable Z := [0.2, 0.5, 1, 2, 4]
ValMax(Sqrt(Z), Z^2, 0 )
```

By itself, the qualifier "**...**" means that the qualified parameter expects one or more parameters. If you combine "..." with **Optional**, it accepts *zero* or more parameters.

Calling a function that has only its last parameter repeated is easy. You just add as many parameters as you want in the call. The extra ones are treated as repeated:

```
Function F2(a; b: ...)
F2(1, 2, 3, 4)
```

Within the function, **F2**, the value of **a** is **1**, and the value of **b** is a list **[2, 3, 4]**.

If the repeated parameter is not the last parameter, or if a function has more than one repeated parameter, for example:

```
Function Fxy(X: ... scalar; Y: ... Optional Scalar)
```

You have several options for syntax to call the function. Use name-based calling:

```
Fxy(x: 10, 20, 40, y: 2, 3, 4)
```

Or use position for the first repeated parameter group and name only the second parameter **y**:

```
Fxy(10, 20, 40, y: 2, 3, 4)
```

Or enclose each set of repeated parameters in square brackets:

```
Fxy([10, 20, 40], [2, 3, 4])
```

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/index.php/Function_Parameter_Qualifiers).

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 57):

- 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** (page 57) of the module to library or filed library.
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 are not accessible to models using the library.

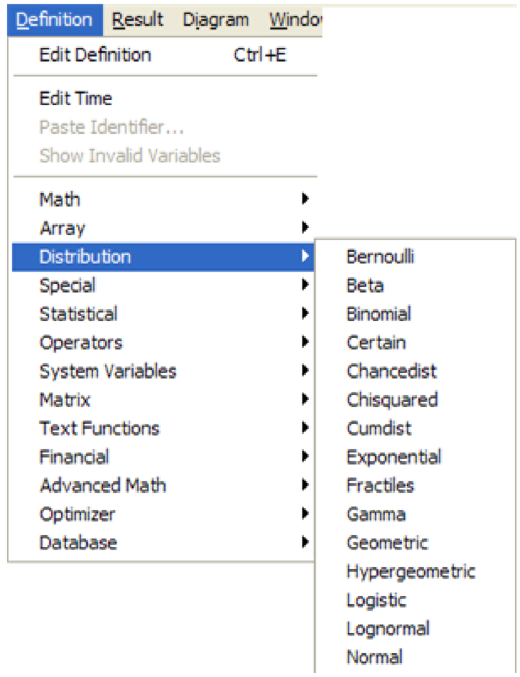
Adding a filed library to a model

Add a filed library to a model using the **Add Module dialog** (page 310).

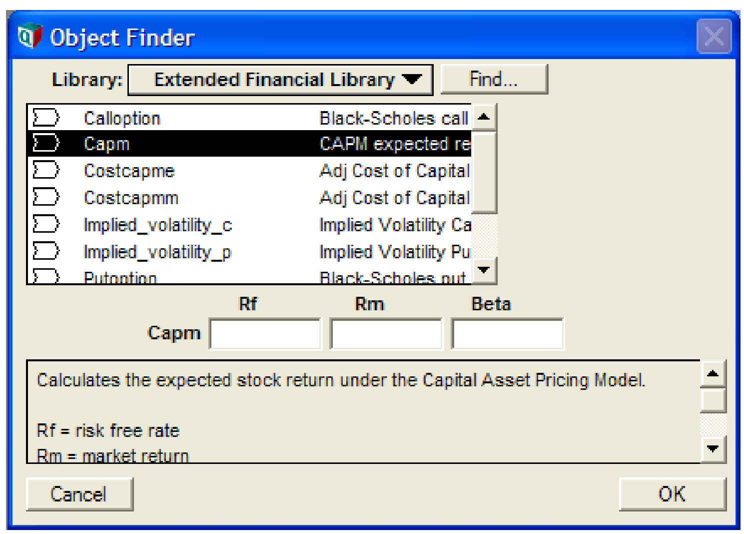
Using a library

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 ***expr*** menu to open the **Object Finder**.
- Paste from the library under the **Definition** menu.

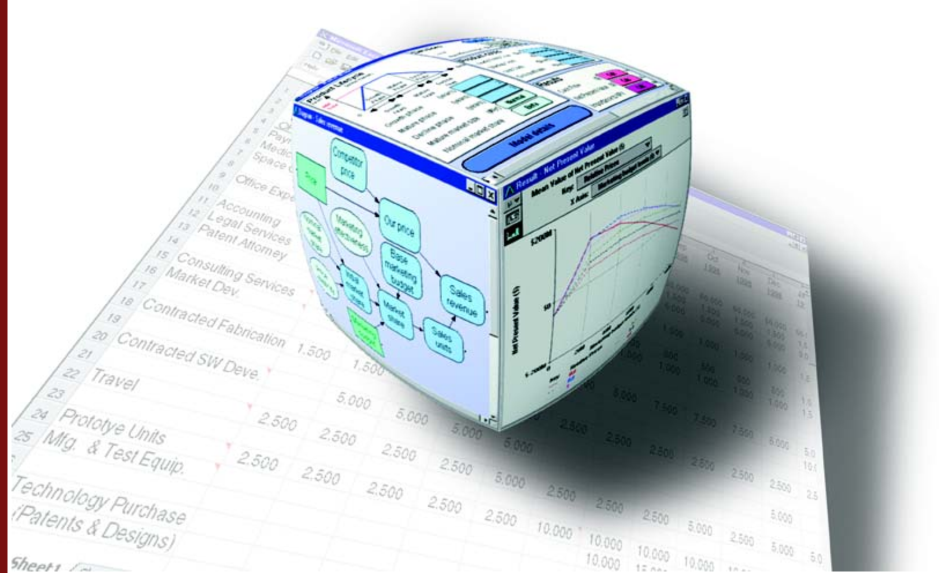


Example Compare the way the **Capm()** function is displayed in the **Object** window (see “Libraries” on page 323) to the way it is displayed in the **Object Finder**.



Chapter 21 *Procedural Programming*

This chapter shows you how to use the procedural features of the Analytica modeling language, including:



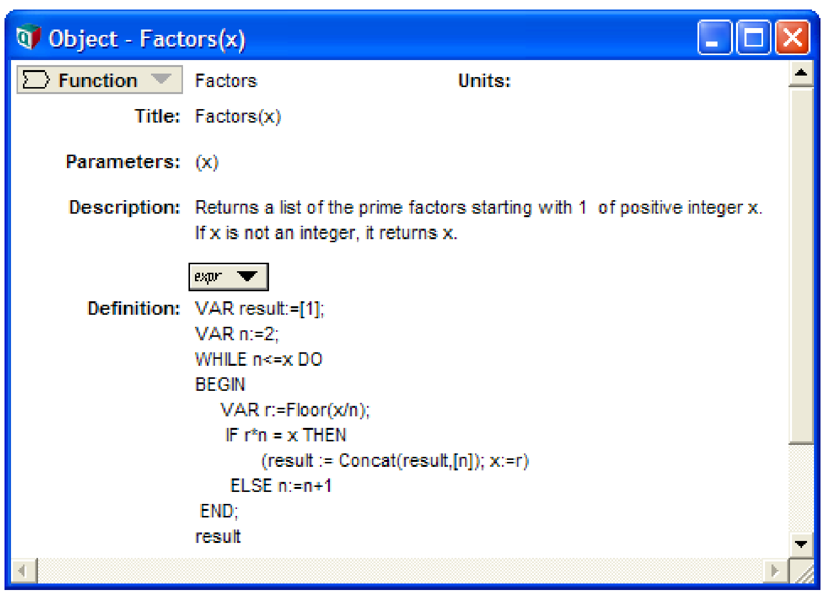
A ***procedural program*** is list of instructions to a computer. Each instruction tells the computer what to do, or it might change the sequence to execute the instructions. Most Analytica models are ***non-procedural*** — that is, they consist of an *unsequenced* set of definitions of variables. Each 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 selects 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. Controlling the evaluation sequence via conditional statements and loops is a large part of programming in a language like in Fortran, Visual Basic, or C++. Non-procedural languages like Analytica free you from having to worry about sequencing. Non-procedural models or programs are usually much easier to write and understand than procedural programs because you can understand each definition (or formula) without worrying about the sequence of execution.

However, procedural languages enable you to write more powerful functions that are hard or impossible without their procedural constructs. For this reason, Analytica offers a set of programming constructs, described in this chapter, providing a general procedural programming language for those who need it.

You can use these constructs to control the flow of execution only within the definition of a variable or function. Evaluating one variable or function cannot (usually) change the value of another variables or functions. Thus, these procedural constructs do not affect the simple nonprocedural relationship among variables and functions. The only exception is that a function called from a button can change the definition of a global variable. See “Creating buttons and scripts” on page 363.

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 “Defining a local variable: Var v := e” on page 328 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 “Begin-End, (), and “;” for grouping expressions” on page 328.
3. **`While test Do body`** construct tests condition **`Test`**, and, if True, evaluates **`Body`**, and repeats until condition **`Test`** is False. See “While(Test) Do Body” on page 333.
4. Begin *e1*; *e2*; … End groups several expressions separated by semicolons “;” — in this case as the body of a While loop. See “Begin-End, (), and “;” for grouping expressions” on page 328.
5. (*e1*; *e2*; …) is another way to group expressions — in this case, as the action to be taken in the Then case. See “Begin-End, (), and “;” for grouping expressions” on page 328.
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 “Assigning to a local variable: v := e” on page 329.
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 parentheses marks **`(`** 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 328
BEGIN *e1; e2*; … ***ei* END**	A group of expressions to be evaluated in sequence.	page 328
(***e1; e2***; … ***ei***)	Another way to group expressions.	page 328
m .. n	Generates a list of successive integers from m to n.	page 337
Var ***x*** := ***e***	Define local variable ***x*** and assign initial value ***e***.	page 328
Index ***i*** := ***e***	Define local index ***i*** and assign initial value ***e***.	page 335
x := ***e***	Assigns value from evaluating ***e to*** local variable ***x***. Returns value ***e***.	page 329
While ***Test*** Do ***Body***	While ***Test*** is **`True`**, evaluate ***Body*** and repeat. Returns last value of ***Body***.	page 333

Construct	Meaning	For more see
{ ***comments*** } /* ***comments*** */	Curly brackets { } and /* */ are alternative ways to enclose comments to be ignored by the parser.	page 327
' ***text*** **'** "***text***"	You can use single or double quotes to enclose a literal text value, but they must match.	page 133
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 339
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 339
\ ***e***	Creates a reference to the value of expression ***e***.	page 340
\ [***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 342
# ***r***	Returns the value referred to by reference ***r***.	page 340

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 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 parentheses characters **(** 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 also tolerates a semicolon (;) 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 (page 265) 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 only changes the parameter and does 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 can 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 are a few exceptions to this rule of no assignments to globals: You can assign to globals in button scripts or functions called from button scripts. See "Creating buttons and scripts" on page 363 for details. You can also assign to a global variable **V** from the definition of **X** when **V** is defined as **ComputedBy(X)**.

ComputedBy(x)

This function indicates that the value of a variable is computed as a side-effect of another variable, **x**. Suppose **v** is defined as **ComputedBy(x)**, and the value of **v** needs to be computed, then Analytica will evaluate **x**. During the evaluation of **x**, **x** must set the value of **v** using an assignment operator.

Even though **v** is a *side-effect* of **x**, its definition is still *referentially transparent*, which means that its definition completely describes its computed value.

ComputedBy is useful when multiple items are computed simultaneously within an expression. It is particularly useful from within an **Iterate()** function when several variables need to be updated in each iteration.

```
Variable rot := ... {a 2-D rotation matrix indexed by Dim and Dim2}
Variable X_rot := ComputedBy(Y_rot)
Variable Y_rot :=
    BEGIN
    Var v := Array(Dim,[X,Y]);
    Var v_r := sum( rot*v, Dim );
    X_rot := v_r[Dim2='x'];
    v_r[Dim2='y'];
    END
```

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 **x** was already indexed by **i**, it sets **x[i=n]** to the value of **e**. For other values of **i**, **x** retains its previous value. All other slices of **x** over **i** retain their previous values. If **x** was indexed by other indexes, say **j**, the result is indexed by **i** and **j**. The assigned slice **x[i=n]** has the value **e** for all values of the other index(es) **j**.

You can index by position as well as name in a slice assignment, for example:

```
x[@i = 2] := e
```

This assigns the value of **e** as the second slice of **x** over index **i**.

Slice assignment, e.g., **x[i = n] := e**, has three limitations:

- **x** must be a local variable.
- **n** must be an atom, not an array.
- You can use only one index. For example, you cannot use an expression like **x[i = a, j=b] := e**, with two index expressions. If **x** has two (or more) dimensions, you can create and assign a slice (e.g., a row) to **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 might be a sign that you are not using the Intelligent Arrays effectively. If so, please (re)read the sections on **Intelligent Arrays** (page 144 and page 160).

For i := a Do expr

The **For** loop successively assigns the next atom from array **a** to local index **i**, and evaluates expression **expr**. **expr** might refer to **i**, for example to slice out a particular element of an array. **a** might be a list of values defined by **`m..n`** or **`Sequence(m, n, dx)`** or it might be a multidimensional array. Normally, it evaluates the body **expr** once for each atom in **a**.

The result of the **For** is an array with all the indexes of **a** containing the values of each evaluation of **expr**. If any or all evaluations of **expr** have any additional index(es), they are also 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 **expr** that might 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 cannot avoid the warning since **`Sqrt(x)`** is evaluated before **`If-Then-Else`** selects which elements of **`Sqrt(x)`** to include. To avoid the warning (assuming **x** is indexed by **i**), the expression can be rewritten as:

```
For j:=I do
   If x[i=j]<0 then 0 else Sqrt(x[i=j])
```

Or as (see next section):

```
Using y:=x in i do
   If y<0 Then 0 else Sqrt(y)
```

Situations like this can often occur during slicing operations. For example, to shift **x** one position to the right along **i**, the following expression would encounter an error:

```
if i<2 then x[i=1] else x[i=i-1]
```

The error occurs when **`x[i=i-1]`** is evaluated since the value corresponding to *i-1=0* is out of range. The avoid the error, the expression can be rewritten as:

```
For j:=i do
   If j<2 then x[i=1] else x[i=j-1]
```

Out-of-range errors can also be avoided without using **For** by placing the conditional inside an argument. For example, the two examples above can be written without **For** as follows:

```
Sqrt(if x<0 then 0 else x)
x[i=(if i<2 then 1 else i-1)]
```

Dimensionality reduction

For can be used to apply a function that requires 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)`**, and 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**, page 318). 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 that *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], ','))
```

This results in:

`Index_1` ▼ **`, Index_2`** ▶

	1	**2**	**3**
1	A	B	C
2	D	E	F
3	G	H	I

Reducing memory requirements

In some cases, it is possible to reduce the amount of memory required for intermediate results during the evaluation of expressions involving large arrays. For example, consider the following expression:

`MatrixA:` A two dimensional array indexed by **`M`** and **`N`**.
`MatrixB:` A two dimensional array indexed by **`N`** and **`P`**.

```
Average(MatrixA * MatrixB, N)
```

During the calculation, Analytica needs memory to compute **`MatrixA`** * **`MatrixB`**, an array indexed by **`M`**, **`N`**, and **`P`**. If these indexes have sizes 100, 200, and 300 respectively, then **`MatrixA`** * **`MatrixB`** contains 6,000,000 numbers, requiring over 60 megabytes of memory at 10 bytes per number.

To reduce the memory required, use the following expression instead:

```
For L := M Do Average(MatrixA[M=L]*MatrixB, N)
```

Each element **`MatrixA[M=L]*MatrixB`** has dimensions **`N`** and **`P`**, needing only 200x300x10= 600 kilobytes of memory at a time.

Tip

For the special case of a **dot product** (page 203), 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** <> 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** continues to loop indefinitely. If you suspect that might 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 "While and array abstraction" on page 338 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(initial, expr, until*, maxIter, warnFlag*)

Suppose the definition of variable **x** contains a call to **Iterate()**. **Iterate()** initializes **x** to the value of **initial**. While stopping condition **until** is False (zero), it evaluates expression **expr**, and assigns the result to **x**. Given the optional parameter **maxIter**, it stops after **maxIter** iterations and, if **warnFlag** is True, issues a warning — unless it has already been stopped by **until** becoming True. If **until** is array-valued, it only stops when *all* elements of **until** 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 can, and usually does, contain nested expressions as some of its parameters. **Iterate()** (again like **Dynamic()** and unlike other functions) can, and usually does, mention the variable **x** that it defines within the expressions for **initial** and **until**. These expressions can 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 **until**. Nevertheless, it is usually better to use **While** because you want it to do an appropriate number of iterations for each element of **until**, rather than continue until all its elements are True. But, with **While** you need to use one of the tricks described on and after "While and array abstraction" on page 338 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:

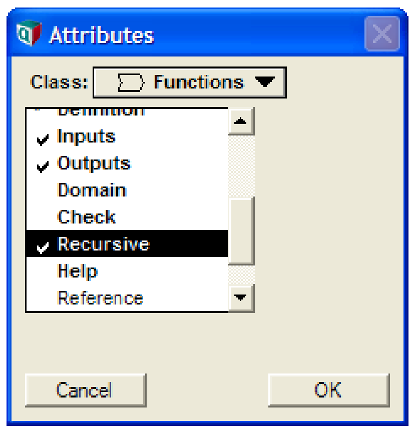
```
Function Factorial2(n: Positive Atom)
Definition: IF n > 1 THEN N*Factorial2(n-1) ELSE 1
```

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 complains 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.

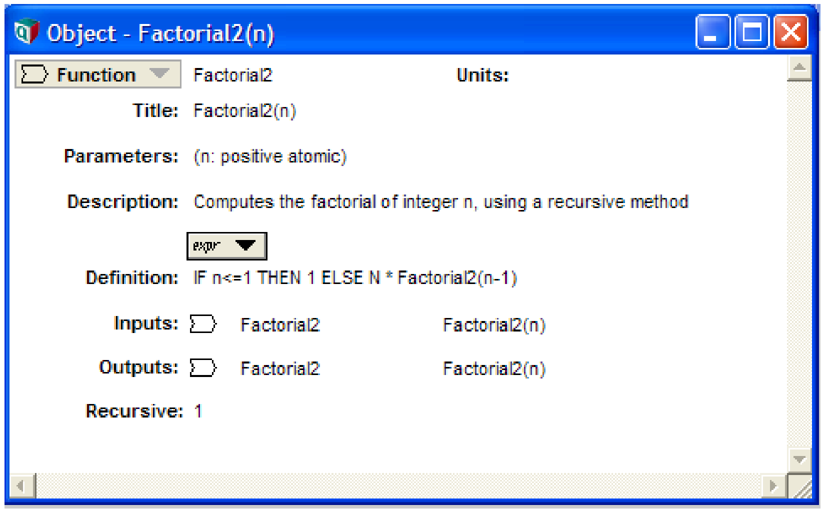


2. Select **Functions** from the **Class** menu in this dialog.
3. Scroll down the list of attributes and click **Recursive** *twice*, so that it shows √, meaning that the recursive attribute is displayed for each function in its **Object** window and the **Attribute** panel.
4. Check **OK** to close **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.



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 to 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 stops and gives a warning if the stack reaches a depth of 256 function calls.

Local indexes

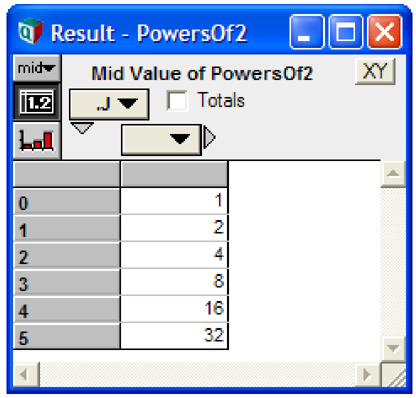
You can declare a local index in the definition of a variable or function. It is possible that the value of the variable or value returned by the function is an array using this index. This is handy because it lets you define a variable or function that creates an array without relying on an externally defined index.

The construct, **Index *i* := *indexExpr*** defines an index local to the definition in which it is used. The expression ***indexExpr*** can 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 →
```



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 can 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 dot (**.**) operator, but do not need to repeat the name of the variable:

```
PowersOf2[.j=5] → 32
```

Any other variables depending on **PowersOf2** can 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., **A** x **A'**, where **A'** is the transpose of **A**. The result is a square matrix. Rather than require a third index as a parameter, **MatSqr** creates the local index, **i2**, as a copy of index **i**.

```
Function MatSqr(a: 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**, and **SplitText**. See below.
- 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** (page 338), which requires its termination condition to be an atom.

- **If b Then c Else d** (page 338), when condition ***b*** is an array, and ***c*** or ***d*** can give an evaluation error.
- Functions with an optional index parameter that is **omitted** (page 339), such as **Sum**(***x***), **Product**, **Max**, **Min**, **Average**, **Argmax, SubIndex**, **ChanceDist**, **CumDist**, and **ProbDist**.

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. In this version, however, using a **For** loop works fine:

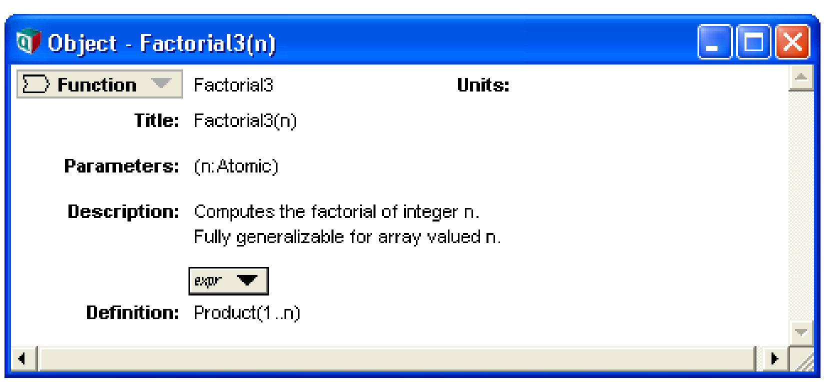
```
Function Factorial3
Parameters: (n)
Definition: FOR m := n DO Product(1..m)
```

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`**.

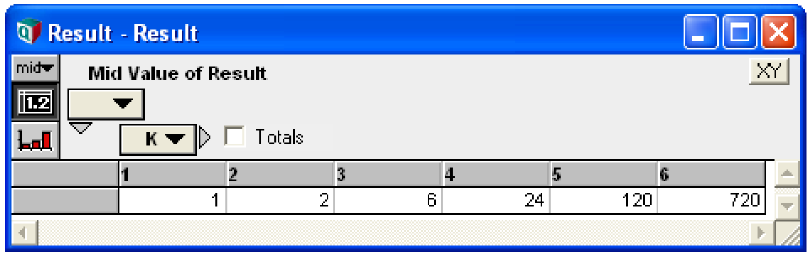
Atom parameters and array abstraction

Another way to ensure array abstraction in a function is to use the **Atom** qualifier for its parameter(s). When you qualify a parameter **`n`** as an Atom, you are saying that it must be a single value — not an array — when the function is evaluated, but not when the function is used:

```
Function Factorial3
Parameters: (n: Atom)
Definition: Product(1..n)
```



```
Index K := 1 .. 6
Factorial3(K) →
```



Notice that **Atom** does not require the actual parameter **K** to be an atom when the function is called. If **K** is an array, as in this case, it repeatedly evaluates the function **Factorial3(n)** with **n** set to each atom of array **K**. It then reassembles the results back into an array with the same indexes as parameter **K**, 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 might require a parameter to be an vector (have only one index), or have multiple dimensions with specified indexes. You can use “Array qualifiers” on page 319 to specify this. With this approach, you can ensure your function array abstracts 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 *a* Then *b* Else *c* 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 **Y**:

```
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 **`Sqrt(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, ',', ''))
```

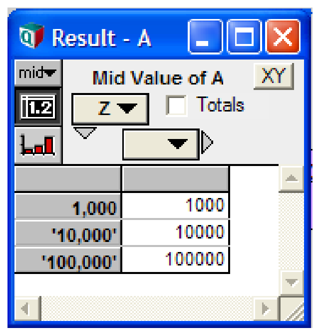
```
RemoveCommas(Z) →
Evaluation Error: The parameter of Pluginfunction TextReplace must
be a text while evaluating function RemoveCommas.
```

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) →
```



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 **Remove-Commas**, **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 sums 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 non-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*	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***	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*	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	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

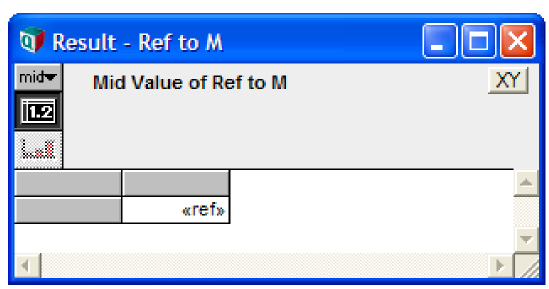
A ***reference*** is an indirect link to a value, an atom or an array. A variable can contain a single reference to a value, or it can contain an array of references. Variables and arrays can themselves contain references, nested to any depth. This lets you create complex data structures, such as linked 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.

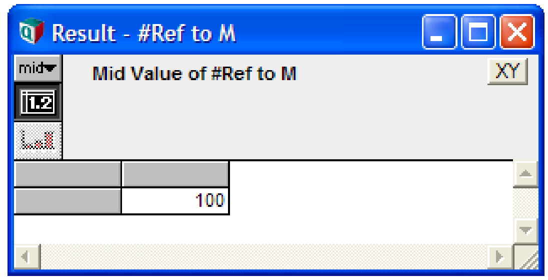
An example:

```
Variable M
Definition: 100

Variable Ref_to_M
Definition: \ M
```

The result of `Ref_to_M` looks like this:

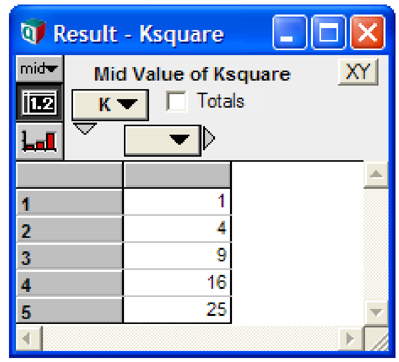


You can double-click the cell containing «ref» to view the value referenced, in this case:

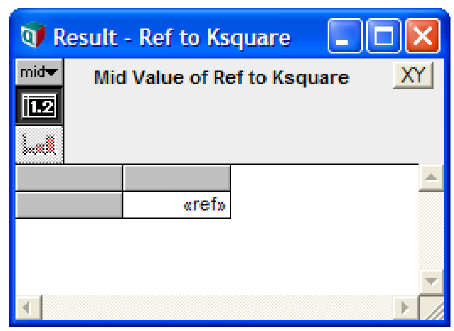


You can also create an array of references. Suppose:

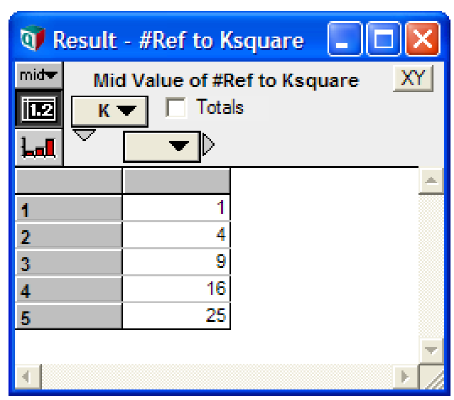
```
Index K
Definition: 1..5

Variable Ksquare
Definition: K^2

Ksquare →
```




```
Variable Ref_to_Ksquare
Definition: \ Ksquare

Ref_to_Ksquare →
```



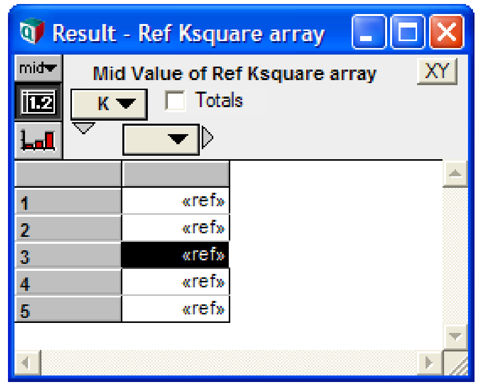
If you click the «ref» cell, it opens:



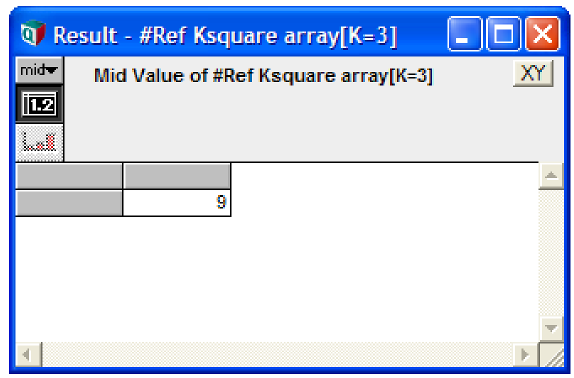
You can also create an array of references from an array, for example:

```
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.



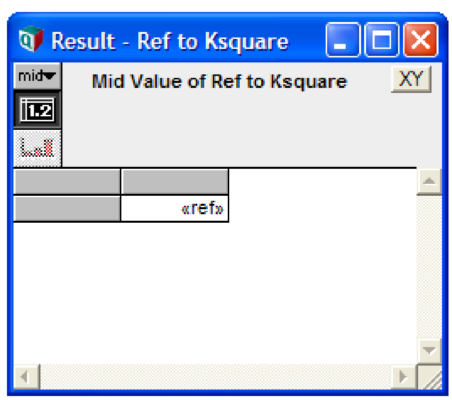
Clicking the third cell, for example, gives:



Managing indexes of referenced subarrays: \ [*i, j*,...] 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) are used as indexes for the referencing array. Thus, in the example above, since there were no indexes in square brackets, the index **K** was used as an index of the reference array. If instead we write:

```
\ [K] Ksquare →
```



It creates a similar result to `\ Ksquare`, since **K** is the only index of `Ksquare`.

To summarize:

\ ***e***	Creates a reference to the value of expression ***e, whether it is an atom or an array***.
\ [] ***e***	Creates an array indexed by all indexes of ***e containing references*** to all atoms from ***e***.
\ [*i*] ***e***	Creates an array indexed by any indexes of ***e*** other than ***i of references*** to subarrays of ***e*** each indexed by ***i***.
\ [*i*, *j* ...] ***e***	Creates an array indexed by any indexes of ***e*** *other than* ***i***, ***j ... of references*** to subarrays of ***e*** each indexed by ***i***, ***j ...***.

In general, it is better to include the square brackets after the reference operator, and avoid the unadorned reference operator, as in the first row of the table. Being explicit about which indexes to include generally leads 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 Library.ana** in the **Libraries** folder for these and other functions for working with linked lists.

Handles to objects

A handle is a pointer to a variable, function, module, or other object. Using a handle lets you write variables or functions that work with the object itself, for example to access its attributes — instead of just its value which is what you usually get when you mention a variable by identifier in an expression.

Viewing handles In a table result, a handle in an index or content cell usually shows the title of the object. If you select **Show by identifier** from the **Object** menu (or press *Control+y*) it toggles to show identifiers instead of titles (as it does in the node diagrams). If you double-click a cell containing a handle (title or identifier) it opens its **Object** window (as it does when you double-click a node in a diagram).

Attributes that contain handles The attributes, **inputs**, **outputs**, and **contains** (the list of objects in a module) each consist of a list of handles to objects. The attribute **isIn** is a single handle to the module that contains this object — the inverse of **contains**.

List of variables: [v1, v2, ... vn]

If you define a variable as a list of variables, for example,

```
Variable A := [X, Y, Z]
```

the variable will have a *self index* that is a list of handles to those variables. In a table result view of **A** (or other variable that uses this index), the index **A** will usually show the titles of the variables. See “List of variables” on page 167 for more. In an expression, the handles in the self index can be accessed using **IndexValue(A)**. The main value of A (either mid value or a probabilistic view of A) contains the results of evaluating **X**, **Y** and **Z**.

Handle(o)

Returns a handle to an Analytica object, given its identifier **o**.

```
Handle(Va1) → Va1
```

HandleFromIdentifier(text)

Returns a handle to global object (i.e., not a local variable or parameter), given its identifier as **text**.

```
Variable B := 99
HandleFromIdentifier("B") → Va1
```

The dependency maintenance is unaware of the dependency on the object. Hence, any changes to the variable **B** above will not cause the result to recompute.

Indexes of Handles

MetaOnly attribute When an index object is defined as a list of identifiers, the **MetaOnly** attribute controls whether it is treated as a ***general index*** or a ***meta-index***. Meta-indexes are useful when reasoning about the structure or contents of the model itself. A general index evaluates the variables appearing in its definition to obtain its mid or sample value, and the values that are recognized by **Subscript** (i.e., **a[i=x]**), while a meta-index (having its **metaOnly** attribute set to 1) does not evaluate the objects in the list. The following comparisons demonstrates the similarities and differences.

```
Constant E:= exp(1)
Variable X := -1
```

General index	*Meta-Index*
`Index I0 := [E,X,Pi,True]` `MetaOnly of I0 := 0 {or not set}`	`Index I1 := [E,X,Pi,True]` `MetaOnly of I1 := 1`
`Variable A0 := Table(I0)(1,2,3,4)`	`Variable A1 := Table(I1)(1,2,3,4)`
`IndexValue(I0) → [E,X,Pi,True]`	`IndexValue(I1) → [E,X,Pi,True]`
`Mid(I0) → 2.718,-1,3.142,1,0]`	`Mid(I1) →[E,X,Pi,True,False]`
`A0[I0=Handle(E)] → 1` `A0[I0=Handle(True)] → 4`	`A1[I1=Handle(E)] → 1` `A1[I1=Handle(True)] → 4`

`A0[I0=E] → 1`
`A0[I0=-1] → 2`
`A0[I0=True] → 4`
	`A1[I1=E] →` ***`Error:-2.718 not in I1`***
	`A1[I1=-1] →` ***`Error:-1 not in I1`***
	`A1[I1=True] →` ***`Error:1 not in I1`***

MetaIndex..Do The construct, **`MetaIndex i :=`** ***`indexExpr`***, declares a local meta-index (see "Local indexes" on page 335). This should generally be used in lieu of the **`Index i :=`** ***`indexExpr`*** construct when ***`indexExpr`*** evaluates to a list of handles.

```
MetaIndex I := contains of Revenue_Module;
Description of (I)
```

IndexesOf(X) Returns the indexes of an array value as a list of handles. The first element of the list is null, rather than a handle, when **`X`** has an ***implicit dimension*** (also known as a ***null-index***).

Dialog functions

Dialog functions display dialog boxes to give special information, warnings, or error messages, or to request information from the user. Dialogs are ***modal*** — meaning that Analytica pauses evaluation while showing the dialog until the user closes the dialog. (**ShowProgressBar** is an exception in that it continues evaluation while it displays the progress bar.) If the user clicks **Cancel** button, it stops further evaluation — as if user pressed *Control+*. (*Control+period*).

Dialog functions display their dialog when evaluated. If the definition of a variable **`A`** calls a dialog function, it will display the dialog when it evaluates **`A`**. If it evaluates **`A`** in mid and prob mode, it displays the dialog each time. It does not display the dialog again until it evaluates **`A`** again — for example, because one of its inputs changes.

MsgBox(message, *buttons, title*)

Displays a dialog with the text **message**, a set of **buttons** and an icon (according to numerical codes below), with **title** in the dialog header bar. Analytica pauses until the user clicks a button. If the user clicks the **Cancel** button, it stops evaluation. Otherwise 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 in the dialog, add one of these 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

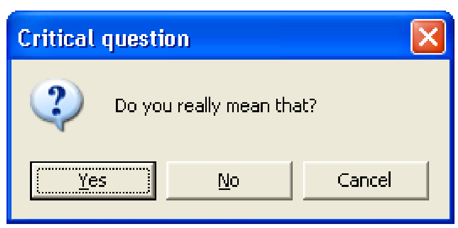
7 = No

Here are some examples.

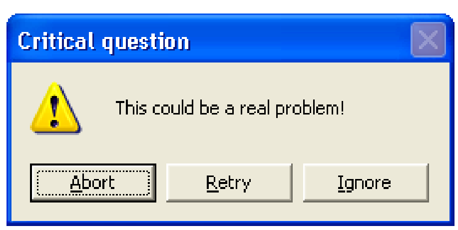
```
Msgbox('OK, I''m done now.', 0+64,'Information') →
```


```
Msgbox('Uh uh! Looks like trouble!', 5+16, 'Disaster') →
```


```
Msgbox('Do you really mean that?', 3+32, 'Critical question') →
```



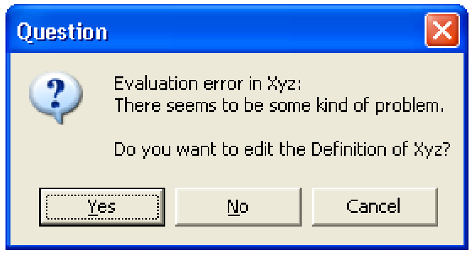

```
Msgbox('This could be a real problem!', 2+48, 'Critical question') →
```



Error(message)

Displays an evaluation error in a dialog mentioning the variable whose definition calls this function, showing the **message** text:

```
Variable Xyz := Error('There seems to be some kind of problem')
Xyz →
```



If you click **Yes**, it opens the definition of the variable or function whose definition (or Check attribute) calls **Error()** in edit mode (if the model is editable). If you click **No** or **Cancel**, it stops evaluation.

Error in check If you call **Error()** in a check attribute (page 115), it shows the error message when the check fails *instead of* the default check error message, letting you tailor the message.

AskMsgText(question*, title, maxText, default*)

Opens a dialog displaying **question** text with a field for the user to provide an answer, which it returns as text.

If you specify **title** text it displays that in the title bar of the dialog. If you specify **maxText** as a number, it will accept only that many characters. If you specify **default** text, it displays that as the default answer.

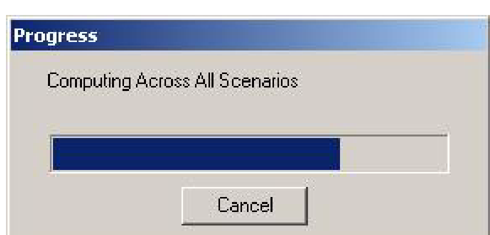
Example
```
AskMsgText("Enter your model access key", title: "License Entry",
maxText: 15)
```

AskMsgNumber(question, *title, default*)

Displays a dialog showing **question** with **title**, if given. It shows a field for user to enter a number, containing **default** number if given. When the user enters a number into the dialog, and clicks **OK**, it returns the number.

ShowProgressBar(title, text, p)

Displays a dialog with the **title** in title bar, a **text** message and a progress bar showing fraction **p** of progress along the bar. The dialog appears the first time you call it with **p<1**. As long as **0<=p<1**, it shows a **Cancel** button, and continues evaluation. If you click **Cancel**, it stops further computation, as if the user had pressed *Control+*. (*Control+period*). If **p=1**, it shows the **OK** button and stops further computation. If you click **OK**, it closes the dialog. The dialog also closes if called with **p>1** or when the computation completes.



Declaration
```
ShowProgressBar(title, text: Text atomic; p: number atomic)
```

Example In this 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
```

Miscellaneous functions

CurrentDataDirectory(*filename*)

Sets the current data directory to **filename**. The ***current data directory*** is the directory used by **ReadTextFile()** and **WriteTextFile()**, if their filename parameter contains no other path. When starting a model, it is the current model directory that contains the model. Specifying a path as a parameter to the function changes the current data directory to that path. If **filename** is omitted, it returns the path to the current data directory.

CurrentModelDirectory(*filename*)

Sets the current model directory to **filename**. The ***current model directory*** is the directory into which the model (and submodules) are saved, by default. When starting a model, it is the directory containing the model. You can change it by selecting a different directly using the directory browser from **Save as**, or by using this function. If **filename** is omitted, it returns the path to the current model directory.

Evaluate(e)

If **e** is a text value, **Evaluate(e)** tries to parse **e** as an Analytica expression, evaluates it, and returns its value. For example:

```
Evaluate('10M /10') → 1M
```

One use for **Evaluate(e)** is to convert a number formatted as text into a number it can compute with, for example:

```
Evaluate('1.23456e+10') → 12.3456G
```

If **e** is an expression that generates a text value, it evaluates the expression, and then parses and evaluates the resulting text value. For example:

```
(VAR x := 10; Evaluate(x & "+" & x)) → 20
```

If **e** is a number or expression that is not a text value, it just returns its value:

```
Evaluate(10M /10) → 1M
```

If **e** is a text value that is not a valid expression — for example, if it has a syntax error — it returns `Null`.

Like other functions, it evaluates the parameter as mid (deterministic) or prob (probabilistic), according to the context in which it is called.

Evaluate(e) parses and evaluates text **e** in a global context. Thus, **e** cannot refer to local variables, local indexes, or function parameters defined in the definition that uses **Evaluate(e)**. For example, this would give an evaluation error:

```
Variable A := (VAR r := 99; Evaluate('r^2') )
```

If **e** evaluates to a handle before it is passed to the function, then that object is evaluated and its (mid or sample) value is returned.

Evaluate and dependencies Analytica's dependency mechanism does not work with variables or functions whose identifiers appear inside the text parameter of **Evaluate**. For example, consider:

```
Variable B := Evaluate("F(A)")
Variable C := F(A)
```

Initially **B** and **C** compute the same value. If you then change the definition of function **F** or variable **A**, Analytica's dependency maintenance ensures that **C** is recomputed when needed using the new definition of **F** and **A**. But, **B** does not know it depends on **F** and **A**, so is not recomputed, and can become inconsistent with the new values for **F** and **A**. In rare cases, you might intentionally want to break the dependency, in which case **Evaluate** is appropriate; otherwise, use it only with care.

GetRegistryValue(root, subfolder, name)

Reads a value from the Windows system registry. This can be quite useful if you install your Analytica model as part of a larger application, and if your model needs to find certain data files on the user's computer (for example, for use with **ShowPdfFile**, **ReadTextFile**, or **RunConsoleProcess**). The locations of those files could be stored in the registry by your installer, so that your model knows where to look.

Example `GetRegistryValue("HKEY_CURRENT_USER", "Software/MyCompany/MyProduct", "FileLocation")`

IgnoreWarnings(expr)

Evaluates its parameter **expr**, and returns its value, while suppressing most **warnings** (page 387) 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** (page 58), because you do want to see warnings that might appear in other parts of the model.

IsResultComputed(x)

Returns 1 if the value of **x** is computed when the function is evaluated. To test whether the sample value of x has been computed, use `Sample(IsResultComputed(x))`, or to test the mid value use `Mid(IsResultComputed(x))`.

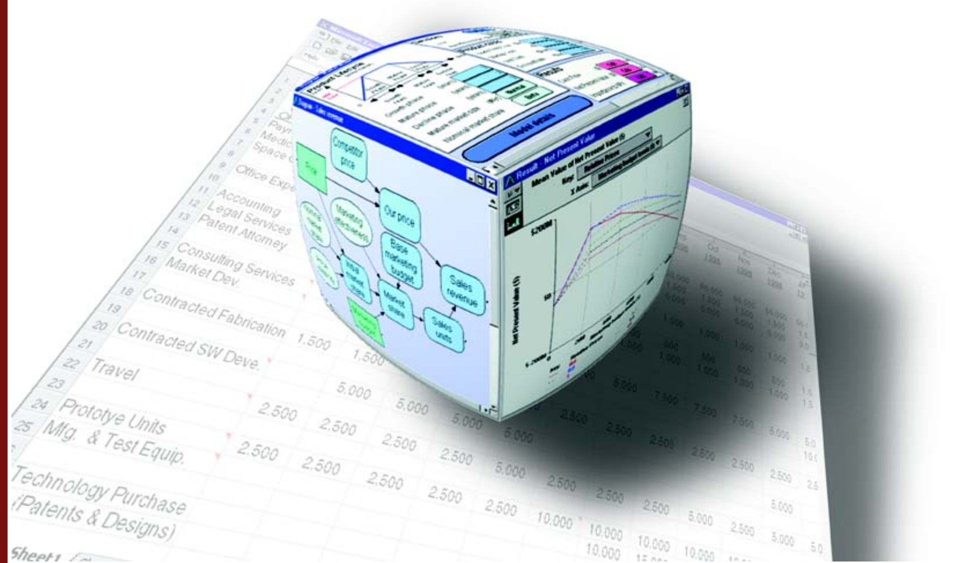
ShowPdfFile(filename)

Opens **filename** using Adobe Reader or Acrobat if one is installed on this computer and the file is a PDF document. **ShowPdfFile** is most useful when called from a button script, for example, as a way to provide the user of your model with a way to open a user guide for your model.

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
- Reading and writing text files
- 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
- Huge arrays: Expand arrays with indexes of over 30,000 elements
- Creating buttons and scripts: Objects that users click to run scripts that can change the model
- 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, and can change them using Analytica Decision Engine. You can use *any* edition of Analytica to run a model that uses buttons, or was saved as browse-only with hidden definitions.

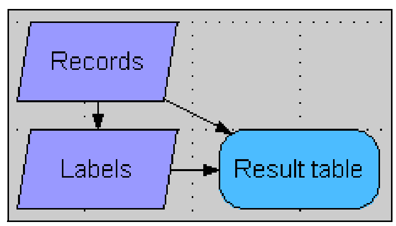
Accessing databases

Analytica Enterprise provides several functions for querying external databases using Open Database Connectivity (ODBC). ***ODBC*** 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, and 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, with its rows indexed by a ***record index***, and columns indexed by a ***label index***. So, the basic structure of an Analytica model for retrieving a result table is this.



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 might 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. These are 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 can contain the Data Source 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 might be as simple as:

```
'DSN=Automobile Data'
```

If a default data source is configured on your machine (consult your database administrator), you can specify it as:

```
'DSN=DEFAULT'
```

Some systems might require one login and password for the server, and another login and password for the DBMS. In this case, both can be specified as:

```
'DSN=Automobile Data; UID=John;
PWD=Lightning; UIDDBMS=JQR; PWDDBMS=Thunder'
```

You can use the **DRIVER** attribute to specify explicitly which driver to use, instead of letting it be determined automatically by the data source type. 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 might wish to connect directly to a driver rather than a registered data source. Some drivers allow this as a way to access a data file directly, even when it is not registered. Also, some drivers 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 can specify:

```
'DRIVER={Paradox Driver};DIRECTORY='D:\CARS'
```

The specific fields used here (UID, PWD, UIDDBMS, PWDDBMS, DIRECTORY, etc.) are interpreted by the ODBC driver, and therefore depend on the specific driver used. Any fields interpreted by your driver are allowed.

If you do not wish to embed the full DSN in the connection text, a series of dialogs 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 prompts you to enter the required information. Explicitly placing information in your model eliminates the extra dialog. A blank connection text can even be used, in which case you 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 might be able to store the information in a DSN file (depending on which drivers and driver manager you are using). You might also be able to register data sources on your machine from that interface.

Configuring a DSN

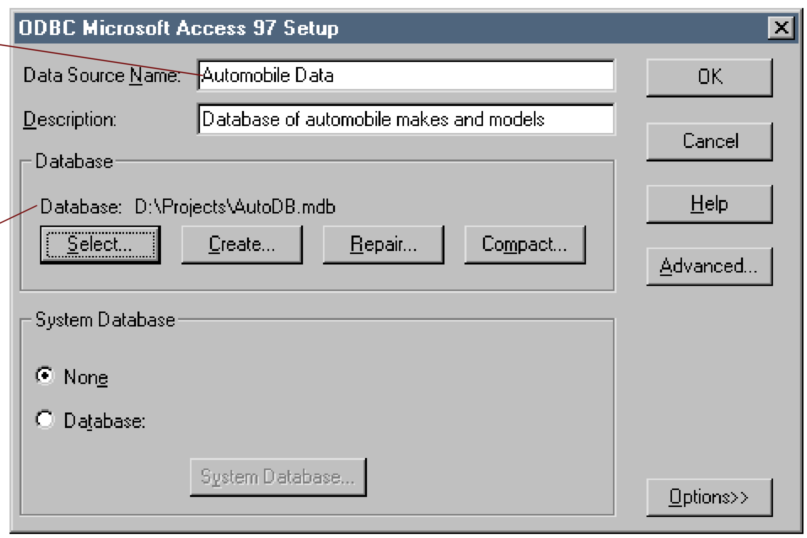
To access a database using ODBC, you must have a Data Source Name (DSN) already configured on your machine. In general, configuring a DSN requires substantial database administration expertise as well as the appropriate access permissions on your computer and network. To configure a data source, you should consult with your Network Administrator 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 the Microsoft Access Driver and click **Finish**.
4. You are led through a series of dialogs specific to the driver you selected. These include dialogs that 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.

The DSN used in your Analytica queries

The actual location of the database



Specifying an SQL query

You can use any SQL query as a text parameter within an Analytica database function. SQL queries can be very powerful, and can 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 can only 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:

```
'SELECT * FROM myQuery'
```

Retrieving an SQL result table

To retrieve a result table from a data source, you need:

1. The data source connection text.
2. The SQL query. These are discussed in the previous two sections. For illustrative purposes, suppose the connection text is **'DSN=Automobile Data'**, and the SQL statement is **'SELECT * FROM vehicles'**. Obtain the relational **Result_table** thus:

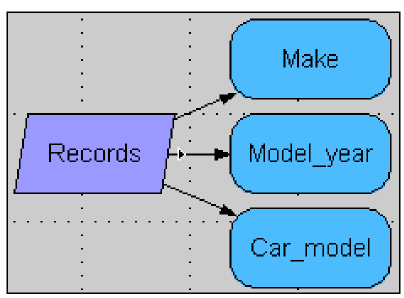
```
Index Records := DBQuery('DSN=Automobile Data',
    'SELECT * FROM vehicles')
Index Labels := DBLabels(Records)
Variable Result_table := DBTable(Records, Labels)
```

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 of a database table

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 is 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 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.

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 the 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 in “MDArrayToTable(a, i, l)” on page 194. **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 1D 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 2D 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, so after the operation, the database table is 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 adds 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 semicolon)

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 semicolon separates multiple statements. For example:

```
'DELETE * FROM TableA ;; SELECT * FROM TableA'
```

This 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 constructs 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 shows 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 can 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()** is 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 1D 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 2D 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 2D table with columns indexed by **columnIndex**, and rows indexed by **dbIndex**. With this last form, **columnIndex** can 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) might be patterns if your ODBC driver manager is ODBC 3 compliant. Use the percent symbol (%) 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** might be a comma-delimited list of table types. Your data source and ODBC driver might or might 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.

MdxQuery(connectionString, mdx)

MdxQuery lets you read or write multidimensional data on an OLAP server database, returning or sending a multidimensional Analytica array. It uses the standard query language, MDX. MDX is analogous to SQL, but where SQL accesses any standard relational database, MDX accesses multidimensional "hypercube" databases. **MdxQuery()** works with Microsoft SQL Server Analysis Services.

connectionStr is the standard text used to identify and connect with the database, similar to that used in other database functions, such as **DBQuery()**. **mdx** is text containing the query in the MDX language.

MdxQuery() creates a local index for each dimension. The local indexes are named **.Axis1**, **.Axis2**, **.Axis3**, etc., and contain the cube member captions as elements. Some cube axes returned from MDX queries are hierarchical, and for these, **MdxQuery** concatenates member captions, separated by commas. For example, if a particular hierarchical axis included calendar year and quarter, an element of **.Axis1** might be "2003,1", i.e., Calendar year 2003, quarter 1. To use a separator other than comma, specify an optional parameter, **sep**, to **MdxQuery**.

For additional usage information and examples, please refer to **MdxQuery** on the Analytica Wiki.

SqlDriverInfo(driverName)

Returns a list of attribute-value pairs for the specified driver. If **`driverName=''`** (an empty text value), returns a list of the names of the drivers. **driverName** must be a text value — it cannot be a list of text values or an index that is defined as a list of text values. This statement would not normally be used in a model, but might 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 tries to read from the current folder, usually the folder containing the current model file. If it doesn't find the file, it opens a Windows browser dialog 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 **`Chr(10)`** line feed character. It then defines a local index **`lines`**, to be used as the index of the array of lines that it returns.

The optional parameter **showDialog** controls whether the file dialog appears. If not specified, then the dialog appears only if the file does not exist. If you set **showDialog** to true (1), it always prompts for the file, even if it finds one by that name. This gives the user a chance to change the filename, while still providing a default name.

WriteTextFile(filename, text, *append, warn, sep*)

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 does 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 writes the corresponding slice of text to each file — following proper array abstraction.

Making a browse-only model and hiding definitions

When you are ready to let others use the models you have created, you might 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 might 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]
```

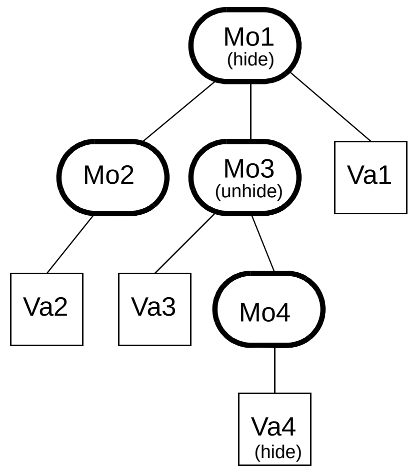
Tip The definition of a variable with an input node is always visible regardless of whether it or its parent module is marked as Hidden.

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 are not 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.

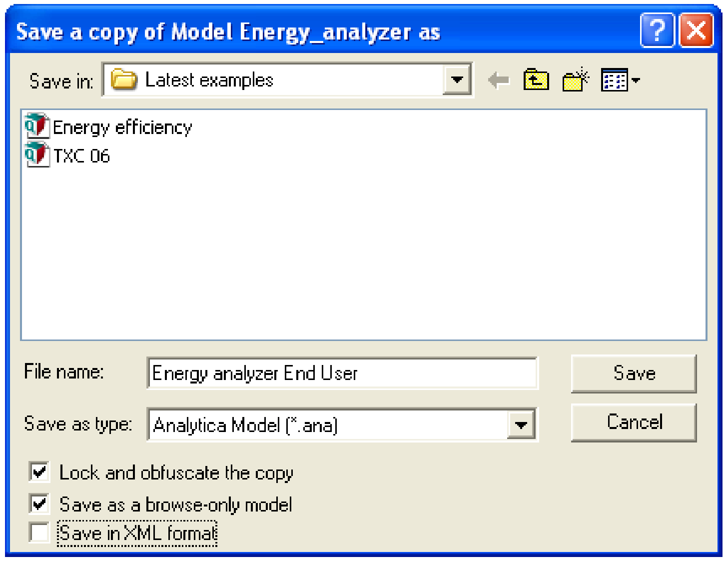


Tip The **Hide Definition(s)** and **Unhide Definition(s)** menu options are disabled if the current model, or any of its linked submodules, has been obfuscated. In this case, obfuscation has locked hiding in place.

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 makes 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 is 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 need the Power Player — or their own Enterprise edition.

When a browse-only model (saved as such from Enterprise) is loaded into 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 submodules 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 results in obfuscating the parent model, as well as any other separately filed submodules or libraries it might 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

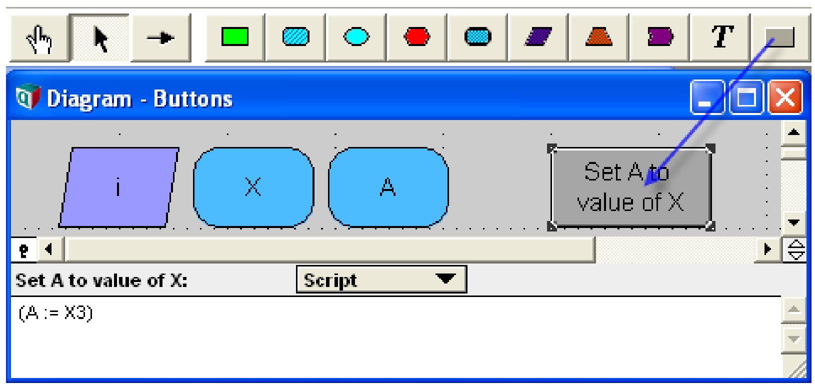
Analytica Enterprise, Optimizer, Power Player, and ADE can manage indexes and arrays of up to 100 million elements in any dimension. The only practical limit on model sizes is the amount of memory. Huge Arrays means they can also handle sample size for probabilistic simulation up to this size. (You can set this in the **Uncertainty Setup** dialog from the **Result** menu.) This also lets 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 semicolon (;) 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 can be called directly from a script, or indirectly from another function called from a script, and so on recursively. Such a function might *not* be from 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 can assign to a nonlocal (global) variable, for example:

```
A := 100
```

This is *not* permitted in the definition of a variable, which only assigns 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 329.

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 might 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 cannot assign to globals in a definition: You can assign to a global variable in a function that is called from a button script. It can 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`** retains 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`** does 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`** is an **edit table**, using those indexes. Any subsequent change to those indexes affect, and possibly invalidate the table. If you want to make sure this doesn't happen, you might want to save copies of the indexes, and transform the table to use the saved indexes.

Assign to an attribute

You can 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 can do it only in a function called from a script, directly or indirectly. You *cannot* assign to an attribute in the definition of a variable. The syntax is:

```
<attrib> OF <object> := <text>
```

Here **`<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 does this again each time you press the button. Notice that the object whose attribute you are resetting can 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 can assign to the definition thus:

```
Definition OF X := Y^2
```

You can 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)

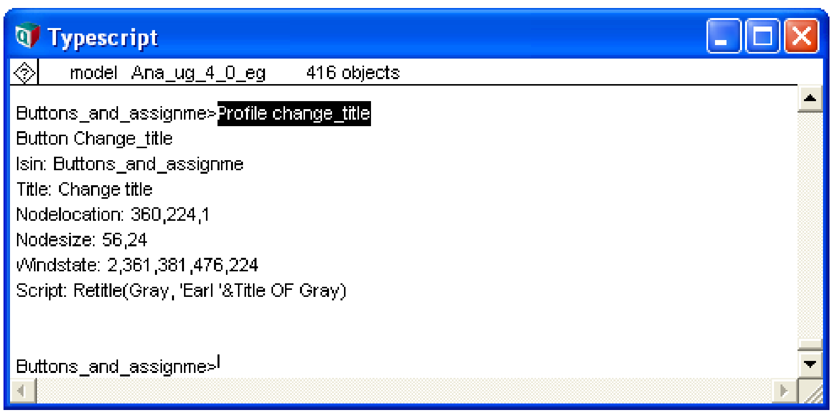
This function evaluates a text value **t** as if it was a script. This means **t** can contain script commands, assignments to globals, and other statements permitted in scripts.

Tip Avoid using **EvaluateScript(t)** except in script functions — that is, functions called from button scripts. This minimizes 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, showing a prompt — the title of the model or module — at the start of each line. You can type in a script command. It prints any results as text, and show another prompt. This window is occasionally useful for advanced users who wish to inspect internal details of a model. You can also use it to test out commands that you want to use in a button script.

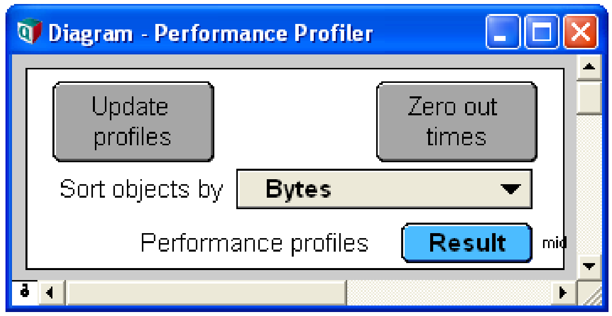
To open the **Typescript** window, press *Control+*`'` (single apostrophe).



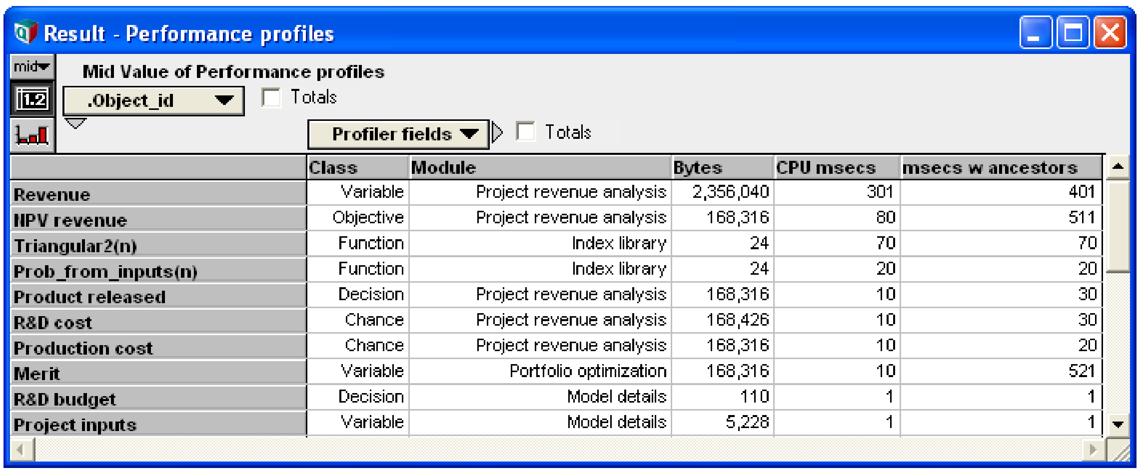
Performance Profiler library

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**.



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 might 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 **`Samplesize`** = 1000, would use about 20 x 30 x 1000 x12 = 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 can share slices. In such cases, it might 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**, which might cause multiple evaluations. A function can 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 GB of memory. If your computer doesn't have that much RAM installed, and it needs more than is available, it can 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 can speed up the application considerably. Another approach is to install more RAM, up to 4 GB.

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 can 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.

This function includes 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 can 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.

Integrating with other Applications

RunConsoleProcess(program)

This function lets an Analytica model run a *console process*, that is, start another Windows application. The application or program can 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	If you omit **block** or set it to True (1), **RunConsoleProcess()** *blocks* — 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 *Control+Break* (or *Control+.*) before the process terminates, it kills the process, and ends further computation by Analytica, just as when Analytica is computing without another process. 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 *Control+Break* (or *Control+.*), it interrupts and stops further computations by Analytica, but has no effect on the spawned process. An unblocking process might 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.
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 *error* message when the process terminates. If **showErr**=2 it shows any text in **stdErr** as a *warning* 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:

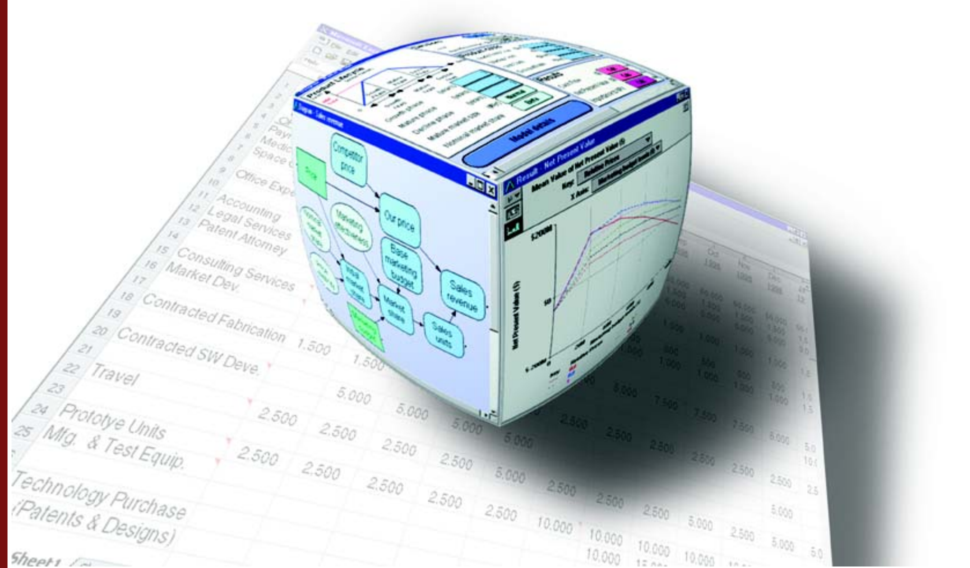
```
RunConsoleProcess("ReadURL.exe", "ReadURL " & url)
```

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 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 identifiers 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**. 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 **y**. Assume you have a random sample of `m` output values generated by Monte Carlo simulation:

$$(y_1, y_2, y_3, \ldots y_m) \tag{1}$$

You can estimate the mean and standard deviation of **y** using the following equations:

$$\bar{y} = \sum_{i=1}^{m} \frac{y_i}{m} \tag{2}$$

$$s^2 = \sum_{i=1}^{m} \frac{(y_i - \bar{y})^2}{(m-1)} \tag{3}$$

This leads to the following confidence interval with confidence α, where *c* is the deviation for the unit normally enclosing probability α:

$$\left(\bar{y} - c\frac{s}{\sqrt{m}}, \bar{y} + c\frac{s}{\sqrt{m}}\right) \tag{4}$$

Suppose you wish to obtain an estimate of the mean of **y** with an α confidence interval smaller than `w` units wide. What sample size do you need? You need to make sure that:

$$w > 2c\frac{s}{\sqrt{m}} \tag{5}$$

Or, rearranging the inequality:

$$m > \left(\frac{2cs}{w}\right)^2 \tag{6}$$

To use this, first make a small Monte Carlo run with, say, 10 values to get an initial estimate of the variance of **y** — that is, s^2. You can then use equation **(6)** to estimate how many samples 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 **y** are relabeled so that they are in increasing order:

$$y_1 \le y_2 \le \ldots y_m$$

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:

$$i = \lfloor mp - c\sqrt{mp(1-p)} \rfloor \tag{8}$$

$$k = \lceil mp + c\sqrt{mp(1-p)} \rceil \tag{9}$$

Note: *The brackets in equations* **(8)** *and* **(9)** *above mean round up* $\lceil$ *and round down* $\rfloor$*, since they are computing numbers that need to be integers.*

Suppose you want to achieve sufficient precision such that the α confidence interval for the *pth* fractile Y_p is given by (y_i, y_k), where y_i is an estimate of $Y_{p-\Delta p}$, and y_k is an estimate of $Y_{p+\Delta p}$. In other words, you want α confidence of Y_p being between the sample values used as estimates of the $(p-\Delta p)^{th}$ 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:

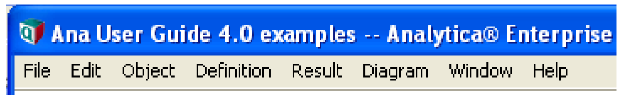
$$m = 0.90 \times (1 - 0.90) \times (2/0.05)^2 = 144 \tag{15}$$

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:

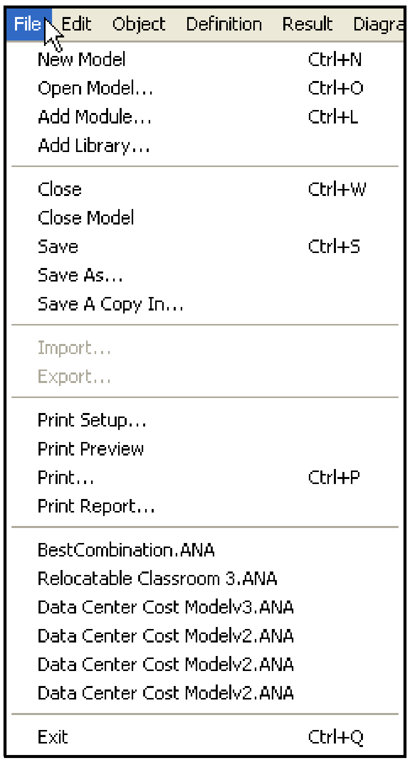
$$m = 0.5 \times (1 - 0.5) \times (2/0.01)^2 = 10,000 \quad \textbf{(16)}$$

These results are completely independent of the shape of the distribution. If you find this an appropriate way to state your requirements for the precision of the estimated distribution, you can determine the sample size before doing *any* runs to see what sort of distribution it might be.

Appendix B: Menus

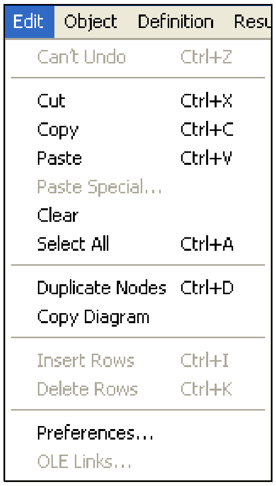


File menu



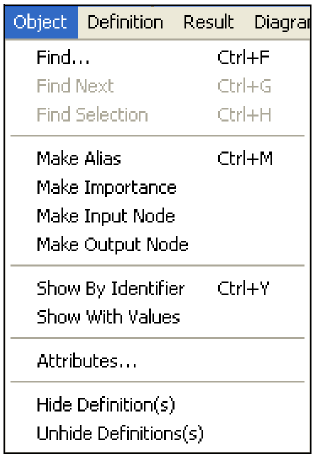
New Model	Starts a new model.
Open Model	Opens an existing, previously saved model.
Add Module	Adds a filed module to the active model.
Add Library	Opens file finder at Analytica Libraries folder to add a library module.
Close	Closes the active window.
Close Model	Closes the model after prompting you 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 298.
Export	Exports the contents of the selected field or cells into a file. See “Importing and exporting” on page 298.
Print Setup	Opens a dialog for selecting paper size, orientation, and scaling options for printing.
Print Preview	Opens a view showing where page breaks 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 27.
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



Undo	Undoes your last action. “Can’t Undo” appears in this location if an undo is not possible.
Cut	Cuts the selected text, node(s), 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 292.
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 292.
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 Diagram or Copy Table	When a **Diagram** window is active, **Copy Diagram** copies a picture of the diagram for pasting into a graphics application. When a table window is active, **Copy Table** copies the entire multidimensional object as a tab-delimited list of tables. See “Copying and pasting” on page 292.
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** (page 58) to examine or change various options.
OLE Links	Opens a dialog to let you change properties for OLE links from external applications into your model. See Chapter 18, “Importing, Exporting, and OLE Linking Data.”

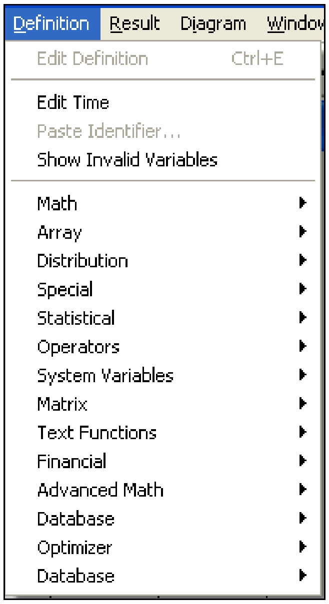
Object menu



Find	Opens a **Find** dialog to search for an object by its identifier or title. If a table is in focus, brings up the **Find in Table** dialog. See “Finding variables” on page 306.
Find Next	Finds the next object that partially matches the previously defined text value. See “Finding variables” on page 306.
Find Selection	Finds an object by its identifier that matches the currently selected text. See “Finding variables” on page 306.
Make Alias	Creates an alias for the selected object(s). See “Alias nodes” on page 54.
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 268.
Make Input Node	Creates an input node for the selected node(s).
Make Output Node	Creates an output node for the selected node(s). See “Using output nodes” on page 122.
Show By Identifier	Shows 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 26.
Attributes	Opens the **Attribute** dialog to set the visibility of attributes and define new attributes. See “Managing attributes” on page 306.
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)

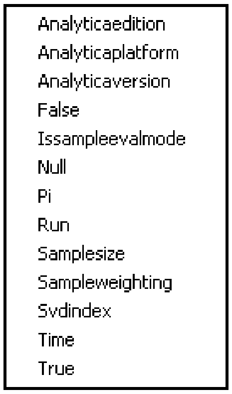
Definition menu

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 pastes it into the definition.



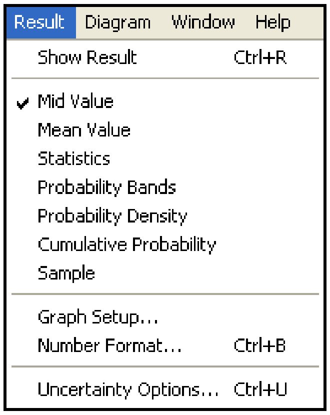
Edit Definition	Opens the appropriate view for editing the definition of the selected variable. If the variable is defined as a distribution or sequence, the **Object Finder** opens. If it is defined as a table or probability table, its edit table window opens. Otherwise, an **Object** window or **Attribute** panel opens, depending on the **Edit attributes** setting in the **Preferences dialog** (page 58).
Edit Time	Opens the **Object** window for the **`Time`** system variable. See "The Time index" on page 282.
Paste Identifier	Opens the **Object Finder** dialog for examining functions and variable identifiers, entering function parameters, and pasting them into definitions. See "Object Finder dialog" on page 112.
Show Invalid Variables	Displays a window listing all variables with invalid or missing definitions. See "Invalid variables" on page 309.
Math	See "Math functions" on page 136.
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 262.
Operators	Arithmetic, comparison, logical, and conditional operators. See "Operators" on page 133.
System Variables	System Variables submenu (see below).
Matrix	See "Matrix functions" on page 202.
Text Functions	See "Converting number to text" on page 138.
Financial	See "Financial functions" on page 210.
Advanced Math	See "Converting number to text" on page 138.
Database	Appears only in Analytica Enterprise. See "Database functions" on page 358.
Optimizer	Appears only if you have the Optimizer activated. See *Optimizer Guide* for more.
your libraries	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



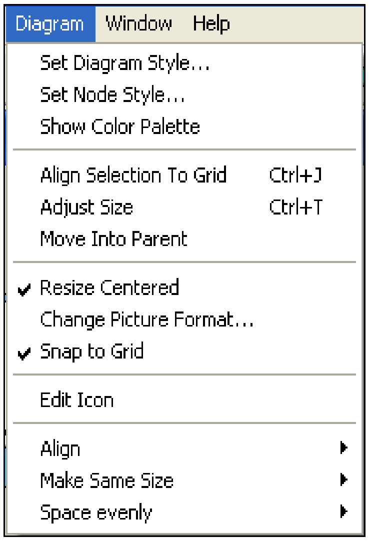
Analyticaedition	The edition of Analytica running, either “Optimizer”, “Enterprise”, “Professional”, “Power Player”, “Player”, “Trial”, “Lite”, “ADE” or “ADE Optimizer”.
Analyticaplatform	The operating system/platform. In Analytica for Windows, this is “Windows,” in Analytica for Macintosh, this is “Macintosh,” and in 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 4.1 subminor version 2 returns the value 40102.
False	The logical (Boolean) constant that evaluates numerically to zero.
Issampleevalmode	This is 1 when evaluated in Sample mode, or 0 when evaluated in Mid mode. You can use this in an expression when you need to compute a mid value differently than a probabilistic value.
Null	A special system constant, returned by various functions when data does not exist at a requested location, and ignored by array-reducing functions when present in the cells of an array. See “Exception values INF, NAN, and NULL” on page 138.
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” on page 225.
Sampleweighting	When this variable to an array indexed by Run, a different weight can be assigned to each probabilistic sample point. See “Importance weighting” on page 257.
Svdindex	The **SingularValueDecomp()** function returns three matrices, `'U'`, `'W'`, and `'V'`. To return all three at once, the return value is an array indexed by **`SvdIndex`**, which is equal to [`'U'`,`'W'`,`'V'`].
Time	The index variable identifying the dimension for dynamic simulation (the **`Dynamic()`** function). See “The Time index” on page 282.
True	The logical (Boolean) constant that evaluates numerically to nonzero.

Result menu



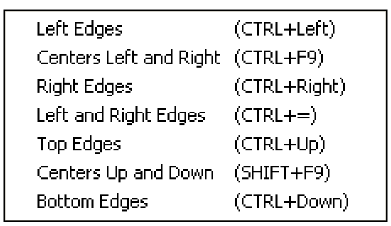
Show Result	Opens a **Result** window for the selected object. See “The Result window” on page 30.
Mid Value	Displays the mid or deterministic value. See “Uncertainty views” on page 33.
Mean Value	Displays the mean of the uncertain value. See “Uncertainty views” on page 33.
Statistics	Displays statistics of the uncertain value in a table as set in the **Uncertainty Setup** dialog. See “Uncertainty views” on page 33.
Probability Bands	Shows probability bands (percentiles) as set in the **Uncertainty Setup** dialog. See “Uncertainty views” on page 33.
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 33.
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 33.
Sample	Displays a table of the values determined for each uncertainty sample iteration. See “Uncertainty views” on page 33.
Graph Setup	Displays a dialog to specify the graphing tool, graph frame, and graph style. See “Graphing roles” on page 86.
Number Format	Displays a dialog to set the number format for displays of results. See “Number formats” on page 82.
Uncertainty Options	Displays a dialog 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” on page 225.

Diagram menu



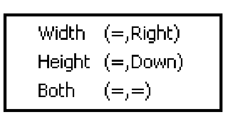
Set Diagram Style	Displays a **Diagram style** dialog to set default arrow displays, node size, and font for this diagram. See “Diagram Style dialog” on page 78.
Set Node Style	Displays **Node style** dialog to set arrow display and font for the selected node(s). See “Node Style dialog” on page 79.
Show 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 77.
Align Selection To Grid	Aligns selected node(s) to the diagram grid. See “Align to the grid” on page 73.
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 78.
Move Into Parent	Moves the selected object from the current diagram to its parent diagram. See “The Object window” on page 23.
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 73.
Change Picture Format	Opens a dialog that lets you convert the internal image format for any selected images to another image format.
Snap to Grid	Turns alignment to the diagram grid on or off in edit mode.See “Align to the grid” on page 73.
Edit Icon	Opens a window to draw or edit an icon for the selected node. See “Adding icons to nodes” on page 124.

Align submenu



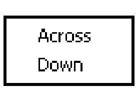
Left Edges	Aligns left edges.
Centers Left and Right	Aligns centers along the same horizontal line.
Right Edges	Aligns right edges.
Left and Right Edges	Moves and changes width so left and right edges are aligned vertically.
Top Edges	Aligns top edges.
Centers Up and Down	Aligns centers along the same vertical line.
Bottom Edges	Aligns bottom edges.

Make Same Size submenu



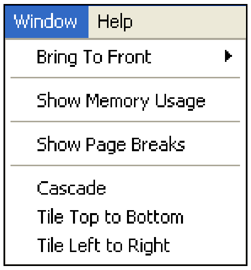
Width	Makes all nodes the same width.
Height	Makes all nodes the same height.
Both	Makes all nodes the same width and height.

Space evenly submenu



Across	Spaces nodes evenly horizontally between leftmost and rightmost node.
Down	Spaces nodes evenly vertically between top and bottom node.

Window menu



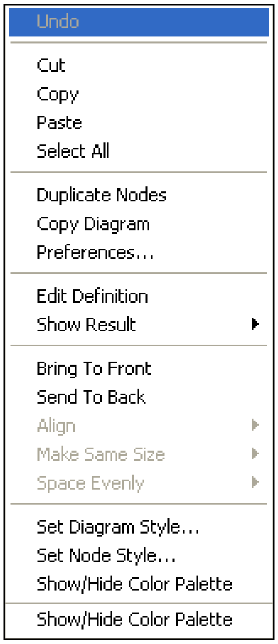
Bring To Front	Displays a list of the current windows; select one to display on top.
Show Memory Usage	Opens a window showing memory usage. See “Numbers and arrays” on page 384.
Show Page Breaks	Shows page breaks for the active diagram.
Cascade	Rearranges all open windows using a standard size, organized 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 *User Guid*e.
Optimizer	Opens the *Optimizer Manual* (only appears in Optimizer-enabled version of Analytica).
Tutorial	Opens the *Analytica Tutorial*.
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 vary depending on your computer setup and the version of Analytica you have.

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 is 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.

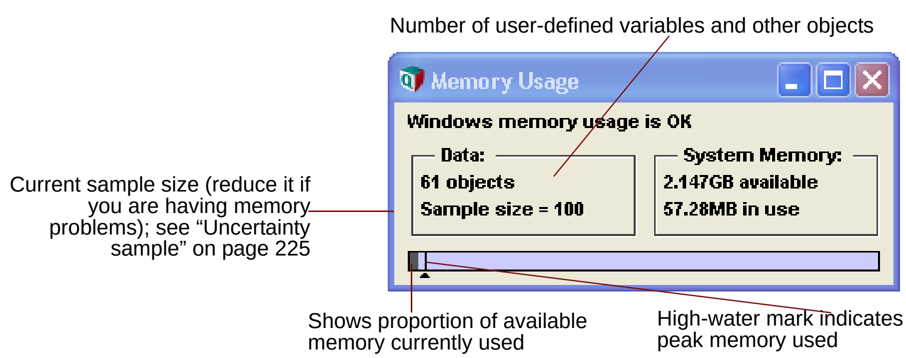
Appendix C: Analytica Specifications

Hardware and software	CPUs 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) *limited by available memory*
	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.



Tip 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.

   In Windows XP, right click on **My Computer** and select the **Advanced** tab. Select Performance **Settings**, **Advanced**, Virtual Memory **Change**. From the resulting dialog, increase maximum paging file size.

3. Analytica is a 32-bit process, and like all 32-bit process, the Windows operating system limits the maximum amount of memory that can be used by a 32-bit process to a default limit of 2GB. In Windows XP and Vista, this limit can be increased to 3GB by editing a system file **`C:\boot.ini`**. For the line corresponding to your operating in that file, append /3GB and /USERVA options. For example, after your edit, the line may be:

   ```
   multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows XP" /3GB /USERVA
   ```

   After making the edit, a system reboot is required for this to take effect.

4. Consider adding more physical memory to your computer.

   If you are limited by the 3GB per process ceiling, then adding more memory will not increase available space, although it may speed up execution when other applications are also open.

5. Consider ways to reorganize your model. Are there dimensions that can be removed from the model, or especially from problematic high-dimensional results. The Performance Profiler (in Analytica Enterprise) can help you pinpoint variables that consume a lot of memory.

For additional ideas for coping with memory limitations, see *Managing Memory and CPU Time for large models* on the Analytica Wiki (http://lumina.com/wiki).

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 warns you and append a digit to make it unique.

To see all identifiers currently in use:

1. Press *Control+'* (*Control*+single apostrophe), 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. Here is an example.

```
Warning:

   Log of non-positive number.
```

A warning is reported during result evaluation to inform you that continuing can yield unexpected results.

You can suppress evaluation warnings for all variables by disabling the **Show result warnings** preference (see “Preferences dialog” on page 58). When **Show result warnings** is unchecked, any warning conditions encountered during result evaluation is ignored. You can also suppress warnings during evaluation of a single expression with the **IgnoreWarnings(expr)** function. See “IgnoreWarnings(expr)” on page 349 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 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 might 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. Here is an 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 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 might 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 can 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 384).

Appendix F: Forward and Backward Compatibility

Backward compatibility

Models created in earlier releases of Analytica can be loaded, viewed, evaluated, and modified with Analytica 4.1. 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 4.1.

When you are trying a model for the first time in 4.1, 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, any potential backward-compatibility issues are reported to you.

The most commonly encountered difference is the multiplication of **`NaN`** or **`INF`** by zero. In earlier releases of Analytica (prior to 4.0), 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 **`NaN`**s and **`INF`**s with a multiplication by zero. Now you might need to use IF-THEN-ELSE instead. If you find certain results have suddenly changed to NaN, this is the likely reason.

There have been many bug fixes in Analytica 4.1, 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 4.1 returns the same sample returned by Analytica 3.1 or 4.0; however, this is not guaranteed, and there are several cases where the sample is different. Although the samples in each release come from the same distribution, the precise points in the random samples might 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. These differences, when they occur, reflect 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 see all samples from that point on changed.

Forward compatibility

It is also possible to run models created or edited in Release 4.1 in earlier releases of Analytica, such as Analytica 4.0 or 3.1, provided you don't rely on functions, features, or functionalities new to Analytica 4.1. The models load into earlier releases of Analytica, although they might encounter problems during parsing or evaluation in the places where 4.1 features are used. A few 4.1 features might 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.

If you have pasted bitmap graphics onto a diagram in 4.1, these will not display when your model is loaded into Analytica 4.0 or earlier, due to a new feature in 4.1 that compresses these images into an internal PNG format. The **Change Picture Format** option on the **Diagram** menu in 4.1 can be used to convert these back to the Legacy Bitmap format so that they display in earlier releases (at a price of increased space consumed).

There are two issues related to edit tables that could potentially create a problem when loading a model edited with 4.0 or 4.1 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 might not yet be fully spliced. When loaded into Analytica 3.1, unspliced edit tables do not successfully parse. To avoid this, prior to saving the model from Analytica 4.1, 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 do 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 and 4.1, it is likely that you 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 4.1?" on page 10 and "What's new in Analytica 4.0?" on page 12 for information on the many things that are new in this release.

Appendix G: Bibliography

Morgan, M. Granger and Max Henrion. *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 in 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 J. Matheson. Influence Diagrams. In *Readings on the Principles and Applications of Decision Analysis*, eds. R. Howard and J. Matheson. pp. 721-762. Menlo Park, CA: 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, CA: Morgan Kaufmann (1988).

Function List

When viewing this list online, click the category or function name to see details.

Basic Math
Abs, **Mod**, **Sqr**, **Sqrt**, **Exp**, **Ln**, **Logten**, **Round**, **Ceil**, **Floor**, **Factorial**, **Radians**, **Degrees**, **Sin**, **Cos**, **Tan**, **Arctan**

Advanced Math
Arccos, **Arcsin**, **Arctan2**, **Bessel***, **BetaFn**, **BetaI**, **BetaIInv**, **Combinations**, **Cosh**, **CumNormal**, **CumNormalInv**, **Erf**, **ErfInv**, **GammaFn**, **GammaI**, **GammaIInv**, **Lgamma**, **Permutations**, **Regression**, **Sinh**, **Tanh**

Creating Arrays
[...], **m..n**, **Array**, **CopyIndex**, **DetermTable**, **Sequence**, **SubTable**, **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
x[i=v], **x[@i-n]**, **x[Time=n]**, **Choice**, **Slice**, **Subscript**

Interpolation
Cubicinterp, **Linearinterp**, **Stepinterp**

Other Array Functions
Concat, **ConcatRows**, **Size**, **Sortindex**, **Subindex**, **Subset**, **Unique**, **Rank**, **IndexesOf**, **IndexNames**, **IndexValue**

Relational to Array conversions
MDArrayToTable, **MDTable**

Matrix Functions
Decompose, **Determinant**, **DotProduct**, **Invert**, **MatrixMultiply**, **Transpose**, **EigenDecomp**, **SingularValueDecomp**

Continuous Distributions
Beta, **ChiSquared**, **Cumdist**, **Exponential**, **Gamma**, **Logistic**, **Lognormal**, **Normal**, **Probdist**, **Random**, **Shuffle**, **StudentT**, **Triangular**, **Truncate**, **Uniform**, **Weibull**

Discrete Distributions
Bernoulli, **Binomial**, **Certain**, **Chancedist**, **Geometric**, **Hypergeometric**, **Poisson**, **Probtable**, **Uniform**

Multivariate Distributions
BiNormal, Correlate_dists, Correlate_with, Dirichlet, Gaussian, Multinomial, MultiNormal, MultiUniform, Normal_correl, RegressionDist

Statistical Functions
CDF, **Correlation**, **Covariance**, **Frequency**, **Getfract**, **Kurtosis**, **Mean**, **Mid**, **PDF**, **Probability**, **Probbands**, **Rankcorrel**, **Regression**, **Sample**, **Sdeviation**, **Skewness**, **Statistics**, **Variance**

Text Functions
&, **Asc**, **Chr**, **FindinText**, **JoinText**, **SelectText**, **SplitText**, **TextTrim**, **TextUpperCase**, **TextLength**, **TextLowerCase**, **TextSentenceCase**, **TextReplace**

Sensitivity Analysis
Correlation, **Dydx**, **Elasticity**, **Rankcorrel**, **Regression**, **Whatif**, **WhatIfAll**

Special Functions
ComputedBy, **Dynamic**, **Error**, **Evaluate**, **EvaluateScript**, **IgnoreWarnings**, **Iterate**, **Subindex**, **Time**, **Today**, **Whatif**, **WhatIfAll**

Miscellaneous Functions
CurrentDataDirectory, **CurrentModelDirectory**, **GetRegistryValue**, **Handle**, **HandleFromIdentifier**, **RunConsoleProcess**

Financial Functions
Cumipmt, **Cumprinc**, **Fv**, **Ipmt**, **Irr**, **Nper**, **Npv**, **Pmt**, **Ppmt**, **Pv**, **Rate**, **Xirr**, **Xnpv**

Dates
DateAdd, **DatePart**, **MakeDate**, **MakeTime**, **Today**

Dialog Functions
MsgBox, **AskMsgNumber**, **AskMsgText**, **ShowProgressBar**, **Error**, **ShowPdfFile**

Operators
@, **+ - * / ^ < <= = <> >= > : & \ # NOT OR AND OF**

Database Access
DBLabels, **DBQuery**, **DBTable**, **DBTableNames**, **DBWrite**, **MdxQuery**, **SqlDriverInfo**, **ReadTextFile**, **WriteTextFile**

Data Types
IsNaN, **IsNumber**, **IsReference**, **IsText**, **IsUndef**, **TypeOf**

Control Constructs
(s1;s2;...), **Begin ... End**, **For...Do...**, **Index**, **If...Then...Else...**, **IfAll**, **IfOnly**, **IgnoreWarnings**, **Iterate**, **MemoryInUseBy**, **Var**, **While...Do...**

System Variables
AnalyticaEdition, **AnalyticaPlatform**, **AnalyticaVersion**, **IsSampleEvalMode**, **Run**, **Samplesize**, **SampleWeighting**, **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**, **Handle**, **Index**, **IsNotSpecified**, **Mid**, **Nonnegative**, **Number**, **Optional**, **OrNull**, **Positive**, **Prob**, **Reference**, **Sample**, **Text**, **Variable**

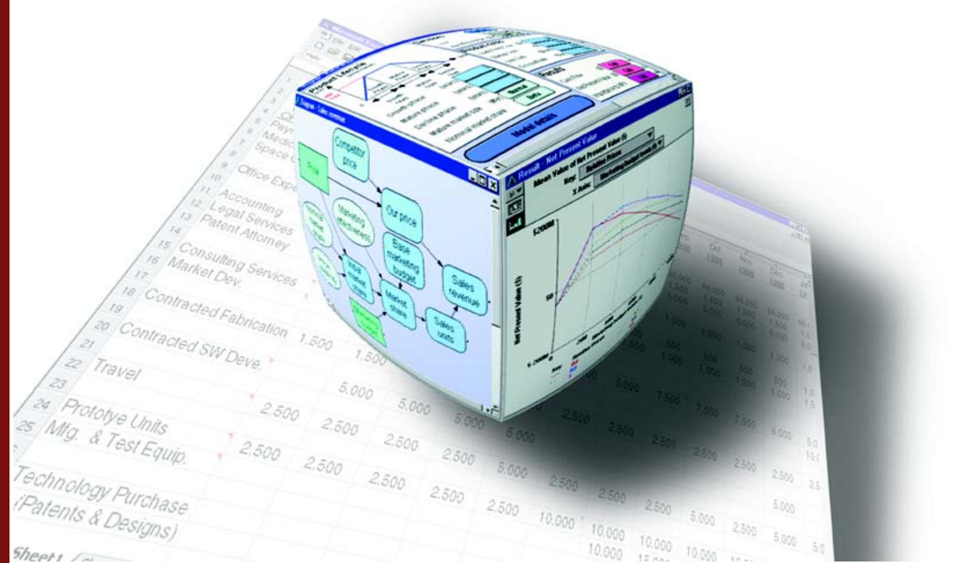
Optimizer Functions
See the *Optimizer Guide* for information on these functions.

LpDefine, **LpFindIIS**, **LpObjSA**, **LpOpt**, **LpRead**, **LpSolution**, **LpStatusNum**, **LpStatusText**, **LpWrite**, **LpWriteIIS**, **NlpDefine**, **QpDefine**

Logistic_regression, **Prob_regression**, **Poisson_regression**

Glossary

This glossary defines special terms used for Analytica and selected statistical terms.



Glossary

ADE See "Analytica Decision Engine."

Alias A node in a diagram that refers to a variable or other node located somewhere else, usually in another module. An alias permits you to display a variable in more than one module. An alias node is distinguished by having its title in italics. See "Alias nodes" on page 54.

Analytica Decision Engine An Edition of Analytica that runs Analytica models on a server computer. The Analytica Decision Engine (or ADE) provides an application programming interface (API) instead of Analytica's graphical end user interface. You can write custom applications in Visual Basic, C++, C#, Microsoft Office, or any language supporting ActiveX Automation or COM to access ADE via its API. For example, you could write a web application that lets you view and run an Analytica model from a web browser on a server. See "Editions of Analytica" on page 5.

Analytica Enterprise An edition of Analytica that includes all features of Analytica Professional, and adds functions for accessing databases, Huge arrays, creating buttons and scripts, model profiling, and the ability to save models that are browse-only and hide selected aspects of a model that are proprietary or confidential. See "Editions of Analytica" on page 5 and Chapter 22, "Analytica Enterprise" on page 351.

Analytica Optimizer An edition of Analytica that includes all features of Analytica Enterprise, and adds the Optimizer engine with functions for linear and nonlinear programming. See "Editions of Analytica" on page 5.

Analytica Player A free edition of Analytica that lets you open, view, and run a model. It lets you change variables designated as inputs, and generate corresponding results. It does not let you edit the model or save changes. See "Editions of Analytica" on page 5.

Analytica Power Player An edition of Analytica that lets you open, view, run a model, and change variables designated as inputs. Like the free Player edition, it does not let you edit the model other than inputs. But it does let you save changes, and it offers the database access and Huge Array features of Analytica Enterprise. See "Editions of Analytica" on page 5.

Analytica Professional The standard edition of Analytica. It provides all the features and functionality required to create, edit, and evaluate models. See "Editions of Analytica" on page 5.

Analytica Trial An edition of Analytica that offers all features of the Professional edition for a trial period, say of 15 days. You can download Analytica Trial from the Lumina web site (www.lumina.com) for a test drive. After expiration, Analytica Trial converts to Analytica Player edition. See "Editions of Analytica" on page 5.

Array A collection of values that can be viewed as a table or graph. An array has one or more dimensions, each identified by an index. See "Introducing indexes and arrays" on page 144.

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" on page 51.

Arrow tool A tool available from the edit tool in the toolbar in the shape of an arrow pointing right. In arrow mode, the cursor changes to this arrow. In this mode, you can draw arrows from one node to another to define influences. See "Drawing arrows" on page 51.

Attribute A property or descriptor of an object, such as its title, description, definition, value, or inputs. See "Managing attributes" on page 306.

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" on page 24.

Author An attribute recording the name(s) of the person or people who created the model, or other object. See "The model's Object window" on page 48.

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" on page 41.

Browse-only models A model that you can open, run, and change designated inputs, but not make other changes even if you have an edition of Analytica that is normally capable of editing a model. You can create a

Browse-only model with Analytica Enterprise. Browse-only models are also obfuscated, meaning that the model file is encrypted and not readable or editable. See "Making a browse-only model and hiding definitions" on page 360.

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 input variables. See "Browse mode" on page 23.

Chance variable An variable that is defined as uncertain by a probability distribution. A chance variable is depicted as an oval node. See "Classes of variables and other objects" on page 20.

Check The check attribute contains an expression that checks the validity of the value of a variable. It usually displays a warning message when the check fails. See "Checking for valid values" on page 115.

Class The type of Analytica object: decision, chance, objective, or index variable; function; module; library; form; model. See "Classes of variables and other objects" on page 20.

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** can be an index of its probability table. See "Add a conditioning variable" on page 239.

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" on page 20.

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" on page 241 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." See "Is the quantity discrete or continuous?" on page 220.

Created The date and time at which the model was first created. This model attribute is entered automatically, and is not user-modifiable. See "The model's Object window" on page 48.

Cumulative probability distribution A graphical representation of a probability distribution that plots the cumulative probability that the actual value of the uncertain variable **x** is 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" on page 36.

Data source A data source is described by a text value, which might contain the Data Source Name (DSN) of the data source, login names, passwords, etc. See "DSN and data source" on page 352.

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" on page 20.

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" on page 107.

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" on page 360.

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" on page 317.

Deterministic table A deterministic function that gives the value of a variable **x** conditional on the values of its input variables. The input must all be discrete variables. The table is indexed by each of its inputs, and gives the value of **x** that corresponds to each combination of values of its inputs. See "Creating a DetermTable" on page 200.

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. See "Uncertainty views" on page 33.

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” on page 20.

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 “Introducing indexes and arrays” on page 144.

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” on page 233.

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.” See “Is the quantity discrete or continuous?” on page 220.

Domain The possible outcomes of a variable. The domain has a type as well as value. The possible types are List of Labels, List of Numbers, or Continuous; the default type is Continuous, except for variables defined with the **Choice()**, **Probtable()**, and **Determtable()** functions. See “The domain attribute and discrete variables” on page 236.

DSN The Data Source 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 352.

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” on page 281.

Edit table A definition defined by the **Table** function, also called an edit table because it can be edited. See “Defining a variable as an edit table” on page 169 and “Editing a table” on page 171.

Editable table A table that the end user can edit directly when it is a model input, including an edit table (table), probability table (probtable), deterministic table (determtable), or subtable. See “Defining a variable as an edit table” on page 169

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” on page 49.

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” on page 131.

Expression type The ***expr*** (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” on page 111.

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” on page 309.

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” on page 309.

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.” See “GetFract(x, p)” on page 264.

Glossary

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" on page 20.

Graph Format for displaying a multidimensional result. To view a result as a graph, click the **Graph** button. See also "Table." See "Viewing a result as a graph" on page 32.

Graphing role 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. See "Graphing roles" on page 86.

Identifier A unique name for a variable used in expressions in definitions. An identifier must start with a letter, have no more than 20 characters, and contain only letters, numbers, and underscore (_) characters (which are used instead of spaces). Each identifier in a model must be unique. Compare to "Title." See "Identifiers and titles" on page 50.

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" on page 268.

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 "Introducing indexes and arrays" on page 144.

The plural, indexes, 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" on page 30.

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" on page 20.

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" on page 52.

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" on page 23.

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." See "Array(i1, i2, … in, a)" on page 183 and "Table(i1, i2, … in) (u1, u2, u3, … um)" on page 185.

Input node A node in a diagram that gives easy access to view and change the value of a variable. This can be a field, choice menu, or edit table. An input node is an alias to a variable. See also "Output node." See "Using input nodes" on page 120.

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. See "Arrows linking to module nodes" on page 52.

Inputs attribute A computed attribute listing the variables and functions used in the definition of this object. 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 attribute." See "Using input nodes" on page 120.

Intelligent array abstraction A powerful key feature of the Analytica Engine that automatically propagates and manages the dimensionality of multidimensional arrays within models. See "Summary of Intelligent Arrays and array abstraction" on page 160.

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. See "Graphing roles" on page 86.

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)" on page 263.

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 variables to be shared. See "Libraries" on page 323.

List A type of expression available in the ***expr*** menu consisting of an ordered set of numbers or expressions. A list is often used to define index and decision variables. See "Creating an index" on page 163.

List of labels A type of expression available in the ***expr*** 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" on page 163.

Matrix A two-dimensional array of numbers with indexes of equal length. See "Matrix functions" on page 202.

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. See "Mean(x)" on page 263.

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 "Prob value." See "Uncertainty views" on page 33.

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" on page 35.

Model The main module containing all the objects that comprise an Analytica model. A model can contain a hierarchy of modules and libraries. Between sessions, a model is stored in an Analytica document file with extension **.ana**. See "Models" on page 18.

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" on page 20.

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" on page 75 and "Show module hierarchy preference" on page 304.

Multimodal distribution A probability distribution that has more than one mode. See "How many modes does it have?" on page 221.

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" on page 20.

Normal distribution The bell-shaped curve, also known as a Gaussian distribution. See "Normal(mean, stddev)" on page 242.

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" on page 360.

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." See "Classes of variables and other objects" on page 20.

Object Finder A dialog used to browse and edit the functions and variables available in a model. See "Object Finder dialog" on page 112.

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" on page 23.

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" on page 20.

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 352.

OLE linking A standard in the Windows operating system for sharing data between applications. See "Using OLE to link results to other applications" on page 292.

Operator A symbol, such as a plus sign (+), that represents a computational process or action such as addition or comparison. See "Operators" on page 133.

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." See "Array(i1, i2, … in, a)" on page 183 and "Table(i1, i2, … in) (u1, u2, u3, … um)" on page 185.

Outline window A view of a model that lists the objects it contains as a hierarchical outline. See "The Outline window" on page 304.

Output node A node in a diagram that gives easy access to see the result of a variable, as a number, table, or graph. This can be a field, choice menu, or edit table. An output node is an alias to a variable. See also "Input node." See "Using output nodes" on page 122.

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. See "Arrows linking to module nodes" on page 52.

Outputs attribute A computed attribute listing the variables and functions that mention this variable in their definition. 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 attribute." See "Using output nodes" on page 122.

Parameters Values or expressions passed to a function, in parentheses after the function name, sometimes termed *arguments*. See "Function calls and parameters" on page 136.

Parametric analysis See "Behavior analysis."

Parent The parent of an object is the module that contains it.

Percentile The median is the fiftieth percentile (also written as 50%ile). More generally, there is probability **p** that the value is less than or equal to the **`pth`** percentile. Compare to "Fractile." See "GetFract(x, p)" on page 264.

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**. See "Probabilistic calculation" on page 224.

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” on page 228.

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** is 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” on page 35.

Probability distribution A probability distribution describes the relative likelihood of a variable having different possible values. See “Probability distributions” on page 232 and “Probabilistic calculation” on page 224.

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 takes 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” on page 35.

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 “Probtable(): Probability Tables” on page 238.

Probtable See “Probability table.”

Prob value The probabilistic value of a variable, represented as a random sample of values from the probability distribution for the variable. The prob value for a variable is based on the prob value for the inputs to the variable. See also “Probabilistic variable” and compare to “Mid value.” See “Uncertainty views” on page 33.

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” on page 185.

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” on page 20.

Result view A window that shows the value of a variable as a table or graph. See “Default result view” on page 59.

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” on page 37.

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” on page 226.

Scalar A value that is a single number. See “Input field” on page 120.

Scatter plot A graph that plots the samples of two probabilistic variables against each other. See “Scatter plots” on page 277.

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. See “Create a probability table” on page 238.
- Refers to the variable itself, instead of the variable’s identifier, in a check attribute or a **`Dynamic`** expression. See “Dynamic(initial1, initial2..., initialn, expr)” on page 282.

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” on page 270.

Side effects If evaluating the definition of variable **A** changes the value of variable **B**, the change to **B** would be a side effect of evaluating **A**. Unlike most computer languages, Analytica does *not* (usually) allow side effects, which makes Analytica models much easier to understand and verify. See “Assigning to a local variable: v := e” on page 329.

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?” on page 222.

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)” on page 263.

Slice A slice of an array is an element or subarray selected along a specified index dimension. A slice has one less dimension than the array from which it is sliced. See “XY comparison” on page 98.

Slicer A control on a graph or table result window that shows the value of a third (or higher) index dimension, not otherwise visible on the graph or table. You can press on the slicer to open a menu to select another value for the slicer index, or to step through other values. See “Slicers” on page 89.

Splicing Table splicing is the process of updating an editable table that depends on a computed index when that index changes. It can result in adding, deleting, or reordering subarrays of the table. See “Splice a table when computed indexes change” on page 173.

SQL Structured Query Language or SQL is a standard interactive and programming language for getting information from and updating a database. See “Accessing databases” on page 352.

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. See “Sdeviation(x)” on page 263.

Suffix notation The default number format, such as 10K, 123M, or 1.23u, where a 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” on page 83.

Symmetrical distribution A distribution, such as a normal distribution, that is symmetrical about its mean. See “Is the quantity symmetric or skewed?” on page 222.

System function A function available in the Analytica modeling language. See also “User-defined function.” See “Building Functions and Libraries” on page 315.

System variable A variable built in to the Analytica language, such as **`Samplesize`** or **`Time`**. See “Using a function or variable from the Definition menu” on page 114.

Table A two-dimensional view of an array. An array can have more than two dimensions, but usually you can only display two at one time. A definition that is a table is called an ***edit table***. In the **Result** window, click the **Table** button to select the table view of an array-valued result. See “Viewing a result as a table” on page 32.

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?” on page 222.

Title An attribute of an Analytica object containing its full name. The title usually appears in the diagram node for the object and in graphs and lists of inputs and outputs. It is limited to 255 characters. The title can contain any characters, including spaces and punctuation. Compare to “Identifier.” See “Edit a node title” on page 49.

Uncertain value See “Prob value.”

Uniform distribution A distribution representing an equal chance of occurrence for any value between the lower and upper values. See “Uniform(min, max, Integer: True)” on page 234 and “Uniform(min, max)” on page 241.

Units The increments of measurement for a variable. Units are used to annotate tables and graphs; they are not used in any calculation. See “Showing values in the Object window” on page 26.

User-defined function A function that the user defines to augment the functions provided as part of the Analytica modeling language. See “Building Functions and Libraries” on page 315.

Value See “Mid value.”

Variable An object that has a value, which can be text, a number, or an array. Classes of variable include decision, chance, and objective. See “The Object window” on page 23.

Variance A measure of the uncertainty or dispersion of a distribution. The wider the distribution, the greater its variance. See “Variance(x)” on page 263.

Index

Index

G

Index

Index

Windows and Dialogs

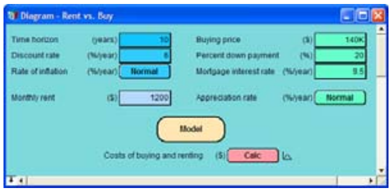


Diagram Window:
Inputs and Outputs

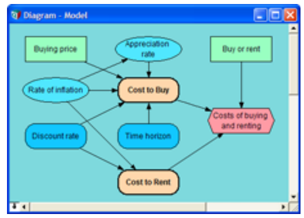
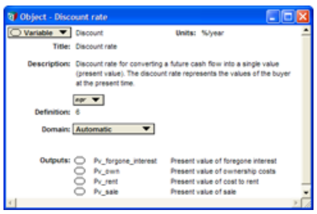
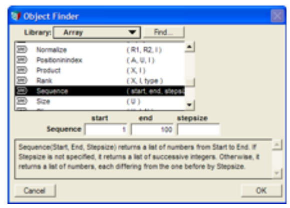


Diagram Window:
Influence Diagram

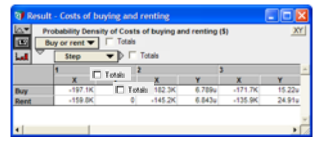
Result Window — Graph View



Object Window



Object Finder



Result Window — Table View

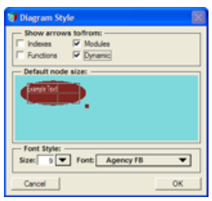
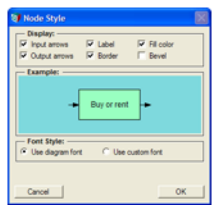
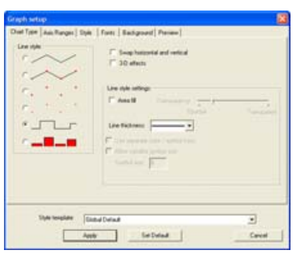


Diagram Style Dialog

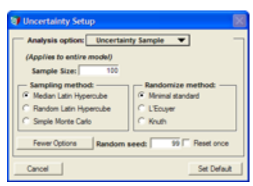


Node Style Dialog

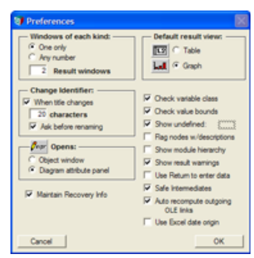
Number Format Dialog



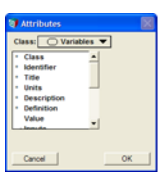
Graph Setup Dialog



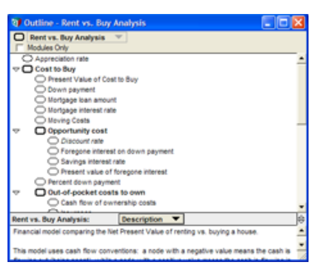
Uncertainty Setup Dialog



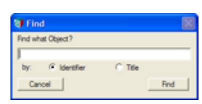
Preferences Dialog



Attributes Dialog



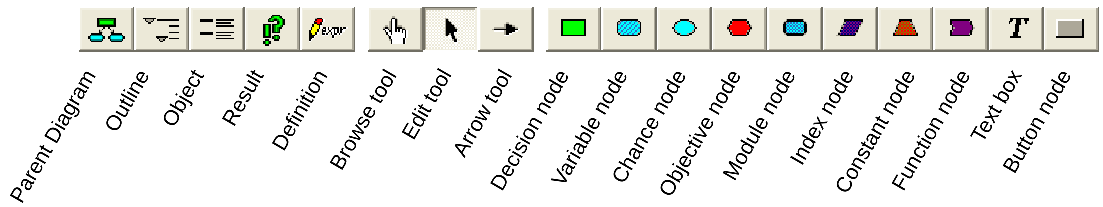
Outline Window



Find Dialog

Quick Reference

The Tool Bar



It displays the node palette when you select the edit tool or arrow tool.

Number Formats

Format	Description	Example
Suffix	letter denotes order of magnitude, such as M for 10^{-6} (see table below)	12.35K
Exponent	scientific exponential	1.235e+004
Fixed point	fixed decimal point	12345.68
Integer	fixed point with no decimals	12346
Percent	percentage	1234568%
Date	text date	12 Jan 2008
Boolean	true or false	True

Suffix format

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	Giga	10^{-9}	n	nano
10^{12}	T	Tera or Trillion	10^{-12}	p	pico
10^{15}	Q	Quad	10^{-15}	f	femto

For more, see “Number formats” on page 82.