User Guide

Analytica 4.5

5 Aug, 2013
Copyright Notice

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

This document is © 1993-2013 Lumina Decision Systems, Inc. All rights reserved.
The software program described in this document, Analytica, includes code that is copyrighted:
Analytica was written using MacApp®: © 1985-1996 Apple Computer, Inc.
Analytica incorporates Mac2Win technology, © 1997 Altura Software, Inc.
Analytica incorporates the Reprise License Manager libraries licensed from Reprise Software, Inc.
Analytica incorporates the Premium Solver SDK licensed from Frontline Systems, Inc.
Analytica incorporates the PCRE library, © 1997-2008, University of Cambridge.
The Analytica® software contains software technology licensed from Carnegie Mellon University exclusively to Lumina Decision Systems, Inc., and includes software proprietary to Lumina Decision Systems, Inc. The MacApp software is proprietary to Apple Computer, Inc. The Mac2Win technology is technology to Altura, Inc. Both MacApp and Mac2Win are licensed to Lumina Decision Systems only for use in combination with the Analytica program. Neither Lumina nor its Licensors, Carnegie Mellon University, Apple Computer, Inc., and Altura Software, Inc., make any warranties whatsoever, either express or implied, regarding the Analytica product, including warranties with respect to its merchantability or its fitness for any particular purpose.
Lumina Decision Systems is a trademark and Analytica is a registered trademark of Lumina Decision Systems, Inc.

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, Isaac Henrion, Sarah Hitzeman, Lynda Korsan, Randa Mulford, Paul Sanford, Rich Sonnenblick, Brian Sterling, and Eric Wainwright.
## Contents

**About Analytica** ................................................................. 1  
Welcome! ....................................................................................... 1  
If you don’t read manuals ................................................................. 2  
Hardware and software requirements ................................................. 2  
Installation and licenses ................................................................... 3  
Editions of Analytica ................................................................. 6  
Help menu and electronic documentation ........................................... 8  
Normally, usually, and defaults ...................................................... 10  
Typographic conventions in this guide ............................................. 10  
User guide Examples folder ......................................................... 11  
What's new? ............................................................................... 11  

**Chapter 2: Examining a Model** ............................................... 13  
To open or exit a model .............................................................. 14  
Diagram window ........................................................................... 15  
Classes of variables and other objects ............................................ 17  
Selecting nodes ........................................................................... 17  
The toolbar .................................................................................... 18  
Browsing with input and output nodes ............................................ 19  
The Object window ......................................................................... 20  
The Attribute panel ......................................................................... 20  
Showing values in the Object window ........................................... 22  
Printing ....................................................................................... 23  

**Chapter 3: Result Tables and Graphs** .................................... 25  
The Result window ......................................................................... 26  
Viewing a result as a table ........................................................... 28  
Viewing a result as a graph .......................................................... 28  
Uncertainty views .......................................................................... 29  
Comparing results .......................................................................... 34  

**Chapter 4: Analyzing Model Behavior** .................................. 37  
Varying input parameters ............................................................ 38  
Analyzing model behavior results ................................................ 39  

**Chapter 5: Creating and Editing a Model** .............................. 43  
Creating and saving a model ....................................................... 44  
Creating and editing nodes .......................................................... 45  
Drawing arrows ............................................................................ 47  
How to draw arrows between different modules .............................. 49  
Alias nodes .................................................................................. 50  
To edit an attribute ......................................................................... 52  
To change the class of an object .................................................... 53  
Preferences dialog ........................................................................ 54  

**Chapter 6: Building Effective Models** .................................. 57  
Creating a model .......................................................................... 58  
Testing and debugging a model .................................................... 60  
Expanding your model ................................................................... 62  

**Chapter 7: Creating Lucid Influence Diagrams** ...................... 65  
Guidelines for creating lucid and elegant diagrams .......................... 67  
Arranging nodes to make clear diagrams ....................................... 68  
Organizing a module hierarchy .................................................... 71
## Contents

Color in influence diagrams.............................................................................................................. 73  
Diagram Style dialog............................................................................................................................ 74  
Node Style dialog................................................................................................................................. 75  
Taking screenshots of diagrams............................................................................................................ 76  

### Chapter 8: Formatting Numbers, Tables, and Graphs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77  
Number formats ...................................................................................................................................... 78  
Date formats............................................................................................................................................... 80  
Display of constraint results .................................................................................................................. 82  
Multiple formats in one table ................................................................................................................... 83  
Graphing roles.......................................................................................................................................... 83  
Graph setup dialog................................................................................................................................. 85  
Graph templates...................................................................................................................................... 92  
XY comparison......................................................................................................................................... 95  

### Chapter 9: Creating and Editing Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103  
Creating or editing a definition ................................................................................................................ 104  
The Expression popup menu..................................................................................................................... 107  
Expression Assist...................................................................................................................................... 108  
Object Finder dialog............................................................................................................................... 110  
Using a function or variable from the Definition menu ........................................................................... 112  
The domain of possible values.................................................................................................................. 113  
Checking for valid values ........................................................................................................................ 117  

### Chapter 10: Creating Interfaces for End Users . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121  
Using input nodes ....................................................................................................................................... 122  
Creating a choice menu ........................................................................................................................... 123  
Using output nodes ................................................................................................................................... 124  
Input and output nodes and their original variables ............................................................................. 125  
Using form modules.................................................................................................................................... 125  
Adding icons to nodes............................................................................................................................. 126  
Graphics, frames, and text in a diagram.................................................................................................... 128  
Default and XML model file formats...................................................................................................... 129  
Hyperlinks in model documentation ....................................................................................................... 130  
Sharing your model.................................................................................................................................... 131  

### Chapter 11: Using Expressions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137  
Expressions.................................................................................................................................................. 137  
Numbers..................................................................................................................................................... 138  
Date and Time Values............................................................................................................................... 139  
Boolean or truth values............................................................................................................................ 139  
Text values................................................................................................................................................ 139  
Operators................................................................................................................................................... 140  
IF a THEN b ELSE c ................................................................................................................................... 142  
Function calls and parameters.................................................................................................................. 143  
Math functions........................................................................................................................................... 143  
Numbers and text....................................................................................................................................... 145  
Exception values INF, NAN, and NULL ..................................................................................................... 146  
Warnings................................................................................................................................................... 147  
Datatype functions...................................................................................................................................... 148  

### Chapter 12: Arrays and Indexes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149  
Introducing indexes and arrays................................................................................................................ 150  
IF a THEN b ELSE c with arrays................................................................................................................ 167
Contents

Sensitivity analysis functions .......................................................................................................................... 299
Tornado charts ................................................................................................................................................ 301
X-Y plots ....................................................................................................................................................... 304
Scatter plots .................................................................................................................................................. 306
Regression analysis ....................................................................................................................................... 307
Uncertainty in regression results .................................................................................................................. 308
Logistic regression functions ........................................................................................................................ 309
Stochastic Information Packets (SIPs) ........................................................................................................... 312

Chapter 18: Dynamic Simulation .................................................................................................................. 315

The Time index ................................................................................................................................................. 316
Using the Dynamic() function ...................................................................................................................... 316
More about the Time index ............................................................................................................................. 318
Initial values for Dynamic ............................................................................................................................... 321
Using arrays in Dynamic() ............................................................................................................................. 321
Dependencies with Dynamic .......................................................................................................................... 322
Uncertainty and Dynamic .............................................................................................................................. 324
Dynamic on non-Time Indexes ....................................................................................................................... 324

Chapter 19: Importing, Exporting, and OLE Linking Data ............................................................................ 327

Copying and pasting ....................................................................................................................................... 328
Using OLE to link results to other applications ............................................................................................ 328
Linking data from other applications into Analytica ...................................................................................... 331
Importing and exporting ................................................................................................................................ 334
Printing to a file ............................................................................................................................................... 335
Edit table data import/export format ........................................................................................................... 336

Chapter 20: Working with Large Models ...................................................................................................... 339

Show module hierarchy preference ................................................................................................................ 340
The Outline window ....................................................................................................................................... 341
Finding variables ............................................................................................................................................. 342
Managing attributes ..................................................................................................................................... 343
Invalid variables ........................................................................................................................................... 345
Using filed modules and libraries .................................................................................................................. 345
Adding a module or library ............................................................................................................................ 346
Combining models into an integrated model .................................................................................................. 347
Managing windows ....................................................................................................................................... 349
Large Memory Usage .................................................................................................................................. 350

Chapter 21: Building Functions and Libraries ............................................................................................. 353

Example function .......................................................................................................................................... 354
Using a function .............................................................................................................................................. 355
Creating a function ........................................................................................................................................ 355
Attributes of a function ................................................................................................................................ 355
Parameter qualifiers ...................................................................................................................................... 356
Libraries ........................................................................................................................................................... 361

Chapter 22: Procedural Programming ........................................................................................................... 363

An example of procedural programming ....................................................................................................... 364
Summary of programming constructs ........................................................................................................... 365
Begin-End, (), and “;” for grouping expressions ............................................................................................ 366
Declaring local variables and assigning to them ............................................................................................ 366
For and While loops and recursion ................................................................................................................ 369
Local indexes ................................................................................................................................................... 373
This introduction explains:

• How to use this manual
• System requirements
• 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 4.5
This User Guide describes how to use Analytica 4.5. 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, webinars, models, 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.

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 57.

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 65.

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 149.

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 245.

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 363.

Hardware and software requirements
To use Analytica, you need the following quite modest minimum configuration:

- Any Intel or AMD processor that can run Windows.
- 60 MB disk space.
- 1 GB RAM (16 GB recommended for large models with 64-bit edition).
- Color display with Mouse, touchpad or other pointer input device.

It helps to have a faster CPU, and, especially, more RAM for large models. Analytica 32-bit will benefit from up to 3 GB RAM if you have it, and Analytica 64-bit will benefit from all additional RAM. It is also handy to have a large screen, or even multiple screens, when working with a large
model. With working with very large models in Analytica 64-bit, performance can be improved by configuring Windows to maintain its Virtual Memory page file on a Solid State Drive.

**Installation and licenses**

After downloading the Analytica 4.5 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 3.1 or 4.4, the installer provides you the option to leave it there, so you can run either version.

We recommend that you run the Analytica installer from the account where you will eventually be using Analytica, rather than changing to an administrator account first. However, if you change to an admin account first, or if you have an IT administrator install it for you, then you (or they) should not enter your activation key into the installer (just leave the field blank), since you want the license to be activated for your user account, not your administrator’s account. When the activation key is not provided during the install, you will be asked to enter it the first time you launch Analytica.

The setup program asks you to confirm the directory name in which to install Analytica, by default, C:\Program Files\Lumina\Analytica 4.5. Most users can accept the default. If you do not have sufficient permissions to write to this directory, then you may want to use [My Documents]\Lumina\Analytica 4.5.

**Licenses**

You need a license to use the software. The Analytica_free101 license is automatically included with the downloaded software, and may be used by anyone.

Two types of Analytica licenses may be purchased: individual and centrally-managed. Individual licenses are installed directly on your computer, while centrally-managed licenses (which includes floating and site licenses) are hosted by a Reprise License Manager (RLM) server.

**Individual licenses**

When you purchase an *individual license*, Lumina emails you an activation key. You may enter the activation key either during installation or after installation from the Update License... dialog on the Analytica Help menu. When you activate a key, a license file for your computer and user account is downloaded from the Lumina’s internet-based activation server and placed on your
computer. If you are using a computer without direct access to the web, an activation key can be manually activated at http://lumina.com/ana/manualActivation.htm. With manual activation, the license file is emailed to you and you must save it into the Analytica install directory, and then select the license in the **Update License...** dialog.

**Centrally-managed licenses**

If your organization has installed a centrally-managed license on an RLM server, such as a floating license, you will need to know the name of the RLM server computer and possibly the port number (if your IT manager has configured a non-default port number). During installation or after installation from the **Update License...** dialog on the Analytica Help menu, select the option to use a centrally-managed license and enter the server name (and port if required). If multiple licenses (e.g., for different editions) are available, you will also have the opportunity to select which license to use.

If you are using a 1-seat floating license, the license seat will be unavailable to other people while you are running Analytica. Once you exit Analytica, the license is returned to the pool for others to use. Likewise, if others are currently using Analytica, the floating license may be unavailable to you. With an N-seat floating license, up to N people may use Analytica simultaneously. For instructions on how to roam a floating license (i.e., when you take your notebook computer out of the office), see [http://lumina.com/wiki/index.php/License_Roaming](http://lumina.com/wiki/index.php/License_Roaming).

**Wiki Access**

The Analytica Wiki ([http://lumina.com/wiki](http://lumina.com/wiki)) is a must for Analytica modelers. During installation you have an opportunity to supply your end-user information (name, email), which is used to set up a Wiki account if your support is current. When the account is set up, the login and password is emailed to you. With an account, you can contribute to and make edits to the Wiki, which we encourage you to do. An account is not required for read-only access.

**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, the Free 101 edition remains available, so you can still open, view, and evaluate your models, and edit models with fewer than 101 user objects. To reactivate the previous edition of 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 using the Free 101 edition (or any other Trial or Player edition). Just select **Update License...** from the Help menu in Analytica and enter your new activation key 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., 4.5.1 to 4.5.2), simply run the installer. The fields you entered originally will be filled in, which you can leave unless you wish to make changes. You do not need to re-enter an activation key that has been previous activated, and can leave that field blank. The installer replaces the older release and reuses your existing license.

**To upgrade to a minor or major release**

You can install Analytica 4.5 and retain an earlier release, such as Analytica 4.4, 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 4.5 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 4.4 (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.

**Free 101**
Allows you to build models with up to 101 user-defined objects (variables, indexes, modules, functions, etc), including the ability to create, edit and save models. You can also load, browse, change designated inputs and evaluate models having more than 101 user objects, but you cannot make changes to those models or save changes you make to inputs. Includes the functions and features found in the Professional edition. As the name implies, this edition is free and may be used for academic, non-commercial and commercial use.

**Professional**
Provides most features, including the ability to create, edit, and save models.

**Power Player**
Lets you review models, change inputs, and view results, and does not let you create or edit models. Unlike the Free edition, it does let you save large models with changed inputs. It also supports models that use Enterprise features, including database access, Huge Arrays, and the Profiler. See Chapter 23, “Analytica Enterprise” for details.

**Enterprise**
Offers all the features of Analytica Professional, plus support for Huge Arrays, reading and writing external data sources, running and communicating with other applications, profiling for analysis of computational effort by variable, and encryption (obfuscation) of sensitive model elements. See Chapter 23, “Analytica Enterprise” for details.

**Optimizer**
Offers all the features of Analytica Enterprise, plus the Optimizer Library that provides powerful solver and structured 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.

*Note: The Free 101 edition now takes the place of the Player and Trial editions that were offered in earlier releases.*
Compare Analytica features by edition

<table>
<thead>
<tr>
<th>Features</th>
<th>Editions of Analytica</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Free 101</td>
</tr>
<tr>
<td>Open models, change inputs, and view results</td>
<td>✓</td>
</tr>
<tr>
<td>Save model with changed inputs</td>
<td>✓</td>
</tr>
<tr>
<td>Create and edit moderate-sized models</td>
<td>✓</td>
</tr>
<tr>
<td>Create and edit large models (more than 101 objects)</td>
<td>✓</td>
</tr>
<tr>
<td>No watermarking of printouts and images</td>
<td>✓</td>
</tr>
<tr>
<td>Hierarchical influence diagrams</td>
<td>✓</td>
</tr>
<tr>
<td>Monte Carlo uncertainty analysis</td>
<td>✓</td>
</tr>
<tr>
<td>Intelligent Arrays, see page 149</td>
<td>✓</td>
</tr>
<tr>
<td>Procedural programming, see page 363</td>
<td></td>
</tr>
<tr>
<td>OLE linking with Excel, see page 328</td>
<td>✓</td>
</tr>
<tr>
<td>Outline Window, see page 341</td>
<td>✓</td>
</tr>
<tr>
<td>Create input and output controls and forms, see page 122</td>
<td>✓</td>
</tr>
<tr>
<td>General function libraries: Math, Array, Distributions, Special, Statistical, Text, Advanced math, Financial, and Matrix</td>
<td>✓</td>
</tr>
<tr>
<td>Save browse-only models and hide sensitive model details, see page 404</td>
<td></td>
</tr>
<tr>
<td>Huge Arrays™ — dimension up to 100 million, see page 407</td>
<td>✓</td>
</tr>
<tr>
<td>ODBC database access, see page 398</td>
<td>✓</td>
</tr>
<tr>
<td>Integration functions: RunConsoleProcess, Excel read/write functions, text file read/write functions</td>
<td>✓</td>
</tr>
<tr>
<td>Time and memory profiling, see page 411</td>
<td>✓</td>
</tr>
<tr>
<td>Optimization engine (LPs, QPs, NLPs)</td>
<td>✓</td>
</tr>
<tr>
<td>Optimizer extension available (at extra cost)</td>
<td>✓</td>
</tr>
<tr>
<td>Application programming interface (see ADE User Guide)</td>
<td></td>
</tr>
</tbody>
</table>

**Analytica Cloud Player**

The Analytica Cloud Player, or ACP, provides a web site where you can upload and share your models with others. Models are viewed through a web browser, so that your colleagues do not need to install any software and can view models from non-Windows computers.
With active support, you are entitled to moderate usage of ACP free of charge on an individual account, with an opportunity to purchase additional credits when a greater usage load is required. Group plans are also available for collaborative project groups where saving of inputs and access control is required.

**Cube Plan**

The CubePlan system provides the Enterprise-wide deployment of Analytica-based models, with advanced access control, web-based access and model modification, collaboration, dashboarding, advanced charting facilities, and GIS integration. Unlike ACP, CubePlan servers can be deployed on client servers within the organization and integrated with other enterprise systems. For more information, see [http://CubePlan.com](http://CubePlan.com).

**Help menu and electronic documentation**

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

![Help menu](image)

**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.

- **User Guide F1**
  Opens this Analytica User Guide as a PDF document in Adobe Reader. Press the function key F1 as a shortcut (see “Online help and electronic documentation” on page 9).

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

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

- **Analytica Wiki**
  Opens the Analytica Wiki home page in your default web browser.
Wiki login info  
Dialog allows you to enter your Analytica Wiki login credentials so that when you click on a link for more information that jumps into the Analytica Wiki, Analytica is able to log you in automatically.

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.

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 select a different license or edition or activation a new activation key to upgrade your license 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.
Introduction

About Analytica

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.

The Analytica Wiki

The Analytica Wiki is the repository of resources for active Analytica users. The Wiki is located at the URL:

http://wiki.Lumina.com

There is no need to commit this URL to memory -- there is a link on the Help menu.

The Wiki contains extended reference materials in greater detail than available here in the user guide, additional example models and libraries, Excel to Analytica equivalents, dozens of hours of recorded webinars, lists of published research publications that used Analytica, FAQs, in-depth articles on Analytica and modeling concepts, categorized function indexes, and much more. In addition, whenever you encounter an error while using Analytica, a hyperlink on the error dialog takes you to the Wiki where additional details and examples about that particular error message are described.

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

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

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” on page 45). You would enter the expressions, \([1, 2, 3, 4, 5]\) and \(N^2\) into their definitions (see “Creating or editing a definition” on page 104). You would not enter the assignment “:=”. The last line says that the expression \(\text{Sum}(\text{Squares}, \ N)\) 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 9 of the Tutorial for a summary of the models in the Examples folder.

What’s new?

We are always working to make Analytica better. As new releases become available, you can review the list of new and improved features introduced into each release from

Chapter 2

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 345). 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.

Example models
Several example models spanning several categories are installed with Analytica. You can quickly get to these either from the Analytica / Example Models shortcut on the Windows’ Start / Programs menu, or from within Analytica from File → Open dialog by pressing the Analytica Example Models button at the upper left.

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 7, “Creating Lucid Influence Diagrams.”

Balloon Help
Node titles are typically kept terse to promote quick visual interpretation of influence relationships, but because they are terse, you may be left with questions about what they mean or represent. For example, Appreciation rate refers to the appreciation of what? Over what time period? Is it inflation adjusted? What units are used? What is the underlying identifier for that node that is used in definition expressions? What is the source for this estimate?

The author of the model can address these types of questions with a more detailed Description for each variable or module. While you are viewing an influence diagram in browse mode,
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 3, “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 20).
- Double-click a module node (thick outline) to see its Diagram window, showing the next level of detail of the model.

Going to the parent diagram
To see the diagram that contains the active module or variable, click 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.

- An hourglass shape depicts a **constraint** — a relationship utilized when solving constrained optimization problems in the Analytica Optimizer edition. The constraint node appears on the toolbar only when using Analytica Optimizer. Optimization is covered in the [Optimizer user guide](#).

- 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” on page 150).

- 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 21, “Building Functions and Libraries.”

This node is a **button** — when you click a button (in browse mode), it executes its script to perform some useful action. You can use buttons with any edition of Analytica, but you need Analytica Enterprise or Optimizer to create a new button (see “Creating buttons and scripts” on page 407).

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 15) 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 341). The outline highlights the object you were previously looking at. The keyboard shortcut is *F3*.

- **Object button**: Click to open the Object window (page 20) 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 26) (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 110); if the variable is an editable table (edit table, subtable, or probability table), it opens the Edit Table (page 178) window. Otherwise, an Attribute panel (page 20) or an Object window (page 20) opens, depending on the Edit Attributes setting in the Preferences dialog (page 54). 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” on page 19.

- **Edit tool**: Lets you create new objects, and move and edit existing objects. See “Creating and editing nodes” on page 45.

- **Arrow tool**: Lets you draw arrows (influences) between nodes on a diagram. See “Drawing arrows” on page 47.
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” on page 171.

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” on page 178.

Click to view and edit a probability distribution in the Function Finder. For more, see “Probabilistic calculation” on page 250.
Chapter 2  

Examining a Model

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 3, “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 3, “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” on page 20).

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 341).
- 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.

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, which you can click to compute and display the array in a separate Result window.

For more about the Result window, see Chapter 3, “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.
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.
Chapter 3  Result Tables and Graphs

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 29), 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” on page 95.
- **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 54).

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

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

Click Calculate to compute and display the new value.

Viewing a result as a table

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

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 78) 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 \( \iff \) or \( \implies \) 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 85) 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 \textit{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 \textit{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.

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:

\[
\text{Chance Rate of inflation := Normal}(2.5, 1)
\]

**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 416) and sampling method in the Uncertainty setup dialog (page 251) 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 254) 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 254) 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 density graph](image)

### 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.

![Probability mass graph](image)

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” on page 265, “Is the quantity discrete or continuous?” on page 246, and “Probability density and mass graphs” on page 263 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.

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

Sample  A sample is an array of the random values from the distribution generated by the Monte Carlo sampling process. The sample is the underlying form used to represent each uncertain quantity. All the other uncertainty views use statistics estimated from the sample. The sample view gives more detail than you usually want. You will likely want to view it mainly when verifying or debugging a model.
Like any other graph, you can display a sample as a table by clicking to see the underlying numerical values.

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, \texttt{Compare1}, with a list of the selected variables.

The result of \texttt{Compare1} is a graph containing an index containing the titles of the variables being compared. This is the \texttt{Self} index of the \texttt{Compare1}. It also includes all the indexes of the array variables being compared — in this case, \texttt{Time} and \texttt{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 4

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 17, “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.
Chapter 4  Analyzing Model Behavior

Analyzing model behavior results

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" on page 171.

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

\[
\text{Decision Year} := 2000 \ldots 2010
\]

If you want to create a list of evenly spaced numbers, use the \textbf{Sequence}(x1, x2, dx) function (page 173), for example:

\[
\text{Decision Quarters} := \text{Sequence}(2000, 2010, 0.25)
\]

**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.
Chapter 4 Analyzing Model Behavior

Analyzing model behavior results

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” on page 27) to get additional insights into model behavior.
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.

![Graph 1](image1)

- 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.

![Graph 2](image2)
A bent or curved line shows that there is a nonlinear dependence. (If you have only two values for the input, the graph will be a straight line even if there is a nonlinear dependence.)
Chapter 5

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.

Tip

When you want to change the model name, title or description, return to the model’s Object Window by pressing the Object window button on the toolbar from the top-level diagram window with no nodes selected.
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” on page 17.

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.
Creating and editing nodes

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.

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 54).

Automatic update when identifier changes

If an identifier changes, Analytica automatically updates any definitions referring to that identifier it uses 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” on page 103).

**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 74) and **Node Style dialog** (page 75).

**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 104), 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 \( A \rightarrow A \), or indirectly so that the arrows form a directed circular path, e.g., \( A \rightarrow B \rightarrow C \rightarrow A \).

If you try to draw arrows that would make a cycle, it 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 18, “Dynamic Simulation”).

Arrows linking to module nodes When there are arrows 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 Indicates that the target variable depends on the origin variable.

Arrow from variable node to module node Indicates that at least one variable in the target module depends on the origin variable.

Arrow from module node to variable node Indicates that the target variable depends on at least one variable in the origin module.
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.

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.

![Diagram showing alias nodes](image)

### 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.

![Diagram showing make alias command](image)

#### 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**.

![Diagram showing arrow between variable and module](image)
It displays an arrow between the nodes.

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

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 Val, in the first module Cost to Buy, with an alias of Val in the second module Total Cost.

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 122) and output nodes (page 124) 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 20) or in the Object window (page 20). 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 \[\times\] 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.

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” on page 17.

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” on page 54), uncertainty options (see “Uncertainty Setup dialog” on page 251), 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 345.

- **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 361.

- **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 345.

- **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 10, “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” on page 349).

- **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 45 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.
Preferences dialog

Opens

These radio buttons control where you view the definition of a selected object, when you click \( \text{\textbullet} \) 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 20) and select the definition text.
- **Diagram attribute panel**: Open the **Attribute panel** (page 20) on the appropriate **Diagram window** and select the definition text.

Personal Preferences

The checkboxes in this cluster are distinguished from all other preferences in that they are user settings, not stored with the model, and persist across Analytica sessions.

- **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.

- **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.

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

- **Expression assist**: When this is checked (the default), pop-ups appear while typing expressions in definition attributes, providing identifier auto-completion and function parameter help. When this is unchecked, pop-ups don’t appear continuously while typing, but you can still access the assist once by typing Ctrl+space, or use Ctrl+? to toggle it on. See “Expression Assist” on page 105.

- **Help balloons**: When checked, a help balloon pops up when the user hovers the mouse over a node for one second. For a balloon to pop up, the node must have a Help or Description attribute. The balloon provides a convenient way for end-users to browse a model.

  By default, balloons are only active in browse mode. When you uncheck **only in browse mode**, balloons also appear in edit mode.

- **Large text in attributes & tables**: Enlarges the font size used in the attribute panel, object window and table cells. This does not determine the size of fonts used in influence diagrams, for that see “Diagram Style dialog” on page 74 and “Node Style dialog” on page 75.

Default result view

Select the radio button to specify which view you prefer as the default when you first display the **Result window** (page 26) 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**
Enables out-of-range warnings when:
- The computed value of a variable falls outside the range specified by the Domain attribute.
- The computed value fails to satisfy the expression specified in the Check attribute. See page 117.

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

<table>
<thead>
<tr>
<th>Node is filled with diagonal pattern:</th>
<th>the definition is missing or is syntactically incorrect</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>Node is flagged with a red triangle to indicate that it has a description</th>
</tr>
</thead>
</table>

**Show module hierarchy**
Show a hierarchy bar at the top of each `Diagram` window showing its nesting level. See page 340.

**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.

**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 327.

**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, it uses January 1, 1900, the same as Excel for Windows.

**Domain acts as self index**
In Analytica 4.2 and earlier, when the domain was defined as a list of values, that list also acted as the self-index of the variable. Analytica 4.3 separates the self-index from the domain index of a variable, which enables the two to be distinct and leads to a cleaner semantics for the domain attribute. Checking this preference preserves the pre-4.3 semantics, which ensures backward compatibility when it is required.

When you load a legacy model into Analytica 4.3, this preference will be checked initially.

**Proactively evaluate indexes**
Specifies whether index objects should be evaluated immediately when a model is loaded. Turning this off speeds up the time required to load a model, postponing the evaluation of indexes until they are needed (such as when an edit table is viewed or a result is computed).
Chapter 6

Building Effective Models

This chapter shows you how to build models that are:

- Focused
- Simple
- Clear
- Comprehensible
- Correct
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” on page 436).

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.

Move from the qualitative to the quantitative

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.

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.
Testing and debugging a model

Chapter 6 Building Effective Models

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 4, “Analyzing Model Behavior”), and to perform sensitivity analysis (see Chapter 17, “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 297) 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.
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**
  - LOAN
  - Rent 1
  - rent 2
  - maint.
  - APR

- **Good object titles**
  - Loan
  - Rent from house
  - Current rent
  - Maintenance costs
  - Mortgage Annual Percentage Rate

**Use consistent node sizes**

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

- **Inconsistent node sizes**
  - Loan
  - Rent from house
  - Current rent
  - Maintenance costs
  - Mortgage Annual Percentage Rate

- **Consistent node sizes**

Node sizes will be uniform if you set the default minimum node size in the **Diagram Style dialog** (page 74) 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” on page 68.

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.
Arranging nodes to make clear diagrams

Arranging 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.

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 (Ctrl+T) from the Diagram menu.

Adjust size examines the node’s contents (its title and picture), font, where word wraps would occur, the the default minimum node size (as set on the Diagram Style... dialog), the current height and width, the edge alignments of others nodes also selected, containment relationships of nodes that grouped within text or picture nodes, locations of corners and edges relative to the grid (when Snap to grid is on), and aspect ratios, and makes tradeoffs between these considerations to select a good size for the selected node(s). When you select Adjust size several times in succession (or by press Ctrl+T multiple times), it cycles through different tradeoffs options, so if you don’t like the sizes selected at first, just press Ctrl+T several more times until you see an arrangement of node sizes that works well.

It often works good to select many nodes simultaneously, especially if your diagram contains inputs fields with left or right alignments, or fields that are grouped within text or picture nodes. With multiple nodes selected, Adjust size generally attempts to preserve containment and alignment relationships.

Adjust size uses utilizes these various criteria, and avoids silly word wrap boundaries, only when the diagram uses ClearType fonts. This is the case for all models created in Analytica 4.4 or later.

---

1. Or right to left for models in Arabic or Hebrew.
Arranging nodes to make clear diagrams

If you load a model created in Analytica 4.3 or earlier, you will need to enable ClearType fonts by selecting Set Diagram Style... on the Diagram menu with the top-level model diagram in focus. The ClearType fonts checkbox appears on that dialog only when you are editing a legacy model. When you turn ClearType fonts on for the first time, the font width and spacing changes slightly, which usually causes some nodes to word wrap in new, undesirable places. A good trick at that point is to select all nodes by pressing Ctrl+A, then select Adjust size repeatedly until a satisfactory appearance is reached.

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 last node selected will be the only node with solid selection handles and is called the reference node. The options are:

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

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.

<table>
<thead>
<tr>
<th>Poor alignment</th>
<th>Good alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
Arranging nodes to make clear diagrams

- 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.

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). To ensure that the nodes end up on the grid, use Space Evenly → Across, on grid and Space Evenly → Down, on grid. The “on grid” variations may change the begin-to-end span of the nodes, while the non-on-grid variants preserve the span but may place nodes off grid.

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 74). 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 75).
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.

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.
Chapter 7 Creating Lucid Influence Diagrams

Organizing a module hierarchy

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 71) with the spaghetti (page 66). 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 it's easier to see the black arrows and text over light backgrounds. Analytica's default colors provide a light neutral color for the background and a slightly stronger color for the nodes.

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.

![Color Palette](image)

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 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 \texttt{Resize centered} option from the Diagram menu.

4. With the node selected, open the \texttt{Set Node Style} dialog from the Diagram menu, check the \texttt{Border} and \texttt{Fill color} options (and \texttt{Bevel}, if you like), and click \texttt{OK}.

5. Select the \texttt{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 \texttt{Send to Back} from the right-click button menu.

\textbf{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 \texttt{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 \texttt{Diagram Style} dialog, select \texttt{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,\texttt{Indexes}, \texttt{Functions}, \texttt{Modules}, and \texttt{Dynamic}. \texttt{Dynamic} (page 316) controls the display of time-lagged dependencies to variables defined with \texttt{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 \texttt{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 54) 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 8

Formatting Numbers, Tables, and Graphs

This chapter shows you how to:

- Control the display of numbers and dates, including date-times, in tables, graphs and input/output fields.
- Select styles and options for graphs.
Chapter 8: Formatting Numbers, Tables, and Graphs

Number formats

The **Number format** dialog lets you control the format of numbers and dates displayed in tables, graphs, and input or output fields. 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.

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 79).

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.

You can change the default number format by pressing **Set Default**. The default format applies to all variables in your model whose number format has not been explicitly set.
Format types  Choose one of these number formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suffix</td>
<td>A letter after the number specifies powers of ten (see below for details)</td>
<td>12.35K</td>
</tr>
<tr>
<td>Exponential</td>
<td>Scientific or exponential notation, where the number after the “e” gives the powers of ten</td>
<td>1.235e04</td>
</tr>
<tr>
<td>Fixed Point</td>
<td>A decimal point with fixed number of decimal digits</td>
<td>12345.68</td>
</tr>
<tr>
<td>Integer</td>
<td>A whole number with no decimals</td>
<td>12346</td>
</tr>
<tr>
<td>Percent</td>
<td>A percentage</td>
<td>12%</td>
</tr>
<tr>
<td>Date</td>
<td>Controls the format used for date-time numbers (see below for details)</td>
<td>12 Jan 2007</td>
</tr>
<tr>
<td>Boolean</td>
<td>Displays 0 as False, any other number as True</td>
<td>True, False</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Power of 10</th>
<th>Suffix</th>
<th>Prefix</th>
<th>Power of 10</th>
<th>Suffix</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^3</td>
<td>K</td>
<td>Kilo</td>
<td>10^-2</td>
<td>%</td>
<td>percent</td>
</tr>
<tr>
<td>10^6</td>
<td>M</td>
<td>Mega or Million</td>
<td>10^-3</td>
<td>m</td>
<td>milli</td>
</tr>
<tr>
<td>10^9</td>
<td>G or B</td>
<td>Giga or Billion</td>
<td>10^-6</td>
<td>µ or u</td>
<td>micro (mu)</td>
</tr>
<tr>
<td>10^12</td>
<td>T</td>
<td>Tera or Trillion</td>
<td>10^-9</td>
<td>n</td>
<td>nano</td>
</tr>
<tr>
<td>10^15</td>
<td>Q</td>
<td>Quadrillion</td>
<td>10^-15</td>
<td>f</td>
<td>femto</td>
</tr>
</tbody>
</table>

**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.
Chapter 8
Formatting Numbers, Tables, and Graphs

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

Date and date-time values are displayed using the format shown when Date is selected. A date-number is a numeric value representing 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. Analytica distinguishes between real numbers and date-time numbers, so that if real numbers and dates appear in the same table, the dates will be shown as dates, the real numbers in the current numeric format. When you are viewing the Date format settings, the format used for numeric values remains italicized.

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

**Short**: e.g., 7/15/2011

**Abbrev**: e.g., 15-Jul-2011
Date formats

Long: e.g., Thursday, July 15, 2011
Time: e.g., 2:05:06 PM
Full: e.g., 15-July-2011 2:05:06 PM

Custom: Use an existing custom format or set up a new one, as shown in the table below.

<table>
<thead>
<tr>
<th>Date format</th>
<th>Displays as</th>
</tr>
</thead>
<tbody>
<tr>
<td>dd-MM-yy</td>
<td>05-08-08</td>
</tr>
<tr>
<td>‘Q’Q YYYY</td>
<td>Q2 2008</td>
</tr>
<tr>
<td>www, d MMM yyyy</td>
<td>Thu, 5 Aug 2008</td>
</tr>
<tr>
<td>wwww, d of MMMM, yyyy</td>
<td>Thursday, 5 of August, 2008</td>
</tr>
<tr>
<td>d-MMM-yyyy hh:mm:ss tt</td>
<td>5-Aug-2008 03:45:22 PM</td>
</tr>
<tr>
<td>MM/dd/yy H:m:s</td>
<td>08/05/08 15:45:22</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>numeric day of the month as one digit</td>
<td>1, 2, ... 31</td>
</tr>
<tr>
<td>dd</td>
<td>numeric day of the month as two digits</td>
<td>01, 02, ... 31</td>
</tr>
<tr>
<td>ddd</td>
<td>ordinal day of month in numeric format</td>
<td>1st, 2nd, ... 31st</td>
</tr>
<tr>
<td>dddd</td>
<td>ordinal day of month in text format</td>
<td>first, second, ... thirty-first</td>
</tr>
<tr>
<td>Dddd</td>
<td>capitalized ordinal day of month</td>
<td>First, Second, ... Thirty-first</td>
</tr>
<tr>
<td>www</td>
<td>weekday in three letters</td>
<td>Mon, Tue, ... Sun</td>
</tr>
<tr>
<td>wwww</td>
<td>weekday in full</td>
<td>Monday, Tuesday, ... Sunday</td>
</tr>
<tr>
<td>M</td>
<td>month as a number</td>
<td>1, 2, ... 12</td>
</tr>
<tr>
<td>MM</td>
<td>month as two-digit number</td>
<td>01, 02, ... 12</td>
</tr>
<tr>
<td>MMM</td>
<td>month as three letter name</td>
<td>Jan, Feb, ... Dec</td>
</tr>
<tr>
<td>MMMM</td>
<td>month as full name</td>
<td>January, February, ... December</td>
</tr>
<tr>
<td>q</td>
<td>quarter as one digit</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>yy</td>
<td>year as two digits</td>
<td>e.g., 99, 00, 01</td>
</tr>
<tr>
<td>yyyy</td>
<td>year as four digits</td>
<td>e.g., 1999, 2000, 2001</td>
</tr>
<tr>
<td>h</td>
<td>hour on a 12-hour clock</td>
<td>1, 2, ... 12</td>
</tr>
<tr>
<td>H</td>
<td>hour on a 24-hour clock</td>
<td>0, 1, ..., 23</td>
</tr>
<tr>
<td>hh</td>
<td>hour on a 12-hour clock as two digits</td>
<td>01, 02, ... 12</td>
</tr>
<tr>
<td>HH</td>
<td>hour on a 24-hour clock as two digits</td>
<td>00, 01, ... 23</td>
</tr>
<tr>
<td>m</td>
<td>minutes</td>
<td>0, 1, ... 59</td>
</tr>
<tr>
<td>mm</td>
<td>minutes as two digits</td>
<td>00, 01, ... 59</td>
</tr>
<tr>
<td>s</td>
<td>seconds</td>
<td>0, 1, ... 59</td>
</tr>
<tr>
<td>ss</td>
<td>seconds as two digits</td>
<td>00, 01, ... 59</td>
</tr>
<tr>
<td>tt</td>
<td>AM or PM</td>
<td>AM, PM</td>
</tr>
</tbody>
</table>

Tip

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

Interpreting input dates

A value entered into a definition, input control, or table cell, is interpreted as a date (or time) when the entire expression can be interpreted as a date or time. For example, 1/5/2012 and 1-5-2012 are interpreted as 5 January, 2012 on a computer set to USA region or 1 May, 2012 elsewhere. All common formats for dates and times are recognized.

Within a sub-expression, the only date format recognized is d-MMM-yyyy, where the month is the three-letter English month name, e.g., 1-May-2008. A hh:mm:ss time format is also recognized. Thus, when you enter (1/5/2012) into a definition or table cell, it is parsed as two consecutive divisions, (1/5) /2012; however, (1-Jan-2012 14:15:00) is parsed as a date-time value.
When **Display dates as numbers** in the number format is checked, then the generalized date parsing is not applied to entries, so that a definition 1–5–2012 would be interpreted as two consecutive subtractions. The canonical format, e.g., 1-May-2008, continues to be recognized.

### 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, 04 de Febrero de 2012
```

where

```
Variable StartDate := MakeDate(2012, 2, 4)
```

### 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.

If you want to paste or link values from Excel or other Windows software to or from Analytica, you should check this option.

### Display dates as numbers

Occasionally you may want to display the numeric value (the number of days since the date origin) for a date-time value. The **Display dates as numbers** checkbox, which displays when viewing the non-date format options, forces the real number value to display for dates and times. The setting also suppresses the more liberal interpretation of expressions as dates and times during parsing (see "Interpreting input dates" above).

### Display numbers as dates

Occasionally you may want to display a real number in date or time format (interpreting it as the number of days since the date origin). The **Display numbers as dates**, visible from the Date format type, forces this to happen.

### 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. Numerous functions exist for working with dates and times, including the **MakeDate()**, **MakeTime()**, **DatePart()**, **DateAdd()**, **Today()**, **ParseDate()**, **Sequence()**, **Floor()**, **Ceil()** and **Round()** functions (page 233).

### Display of constraint results

Constraint nodes are used in Analytica Optimizer models, and are defined as a comparison such as \( x+y \leq z \). When the result of a constraint node is displayed, both the left-hand and right-hand side values of the comparison are displayed using the selected number format. For example, the constraint might contain the definition:

```
GetFract(Expenses,75%) \leq GetFract(Revenue,25%)
```

but when the result of the constraint is displayed, a cell in the result displays might display as:

```
845.1 \leq 12.3K
```

Not that in a non-constraint node, the result of a comparison would display as a boolean value, i.e., 0 or 1.
When the constraint is violated the display is proceeded by {!, e.g., {!}28<=21. This display is helpful when debugging optimization models, since it allows you to see the actual values of each side of the constraint. When the Boolean number format is selected, the comparison display is not used -- True is displayed when the constraint is satisfied, False when it is violated. Beware that the optimizer solver engines generally find solutions that are within an epsilon of satisfying each constraint, but when displaying the constraint result, you will see False when the constraint is violated only by epsilon and the solver engine considers it satisfied.

### 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:

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

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 85) or for **XY graphs** (page 95).

**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 85). 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 85) (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.

```
Result - Energy capacity and load

<table>
<thead>
<tr>
<th>Energy capacity and load (MW)</th>
<th>Key: Energy capacity and load (MW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>Mid Value of Energy capacity and load (MW)</td>
</tr>
<tr>
<td>0.4</td>
<td>Horizontal Axis: Infrastructure design life</td>
</tr>
</tbody>
</table>
```

- **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.

![Graph setup dialog](image)

**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” on page 263.

**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.
Graph setup dialog

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.
Graph setup dialog

- **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 4.5 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.
Chapter 8 Formatting Numbers, Tables, and Graphs

Graph templates

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 File Explorer, 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

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” on page 306 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:

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

For a standard graph of **V** against its index, **Degrees**, select **V** from the diagram and click the **Result** button (*Control+r*). Repeat with **U** to display the graph for **U** against **Degrees**.
For these graphs, we selected the symbol plus line line style (page 86) 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:

\[
\text{Variable } W := \sin(2\cdot\text{Degrees})
\]

The parameter \(2\cdot\text{Degrees}\) 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.
Creating and Editing Definitions

This chapter shows you how to:
• Create definitions
• Edit definitions
• Expression Assist
• Use the Object Finder
• The domain attribute
• Check the validity of a variable’s value
This chapter introduces the tools for creating and editing mathematical models by giving each variable a formula that defines how to compute its value in its definition. The definition of a variable can be a simple number, text, a probability distribution, or a more complicated expression. It can also be a list or table of numbers or other expressions. Subsequent chapters present more details about using mathematical expressions, arrays, and probability distributions.

Creating or editing a definition

To create or edit the definition of a variable, first be sure that the edit tool is selected. Select the variable of interest and do any of the following:

- Click in the toolbar, or press Control+e.
- Select Edit Definition from the Definition menu.
- Double-click the variable to open its Object window. Then click in the definition field.
- Click the key icon to open the Attribute panel of the diagram. Select Definition from the Attribute popup menu. Then click in the definition field.

If you have drawn arrows into this variable from other variables (Down_payment and Buying_price in this example), they appear in the 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 11, "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 107).
Expression Assist

As you type an expression into the definition, the Expression Assist pops up with each keystroke to show possible identifier completions, function parameters, function descriptions, or index values. Expression Assist may be turned on or off from the Preferences dialog (page 54). Even when it is turned off, you can press Ctrl+Space to display it temporarily. See “Expression Assist” on page 108.

Special editing key combinations

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

<table>
<thead>
<tr>
<th>Key or key combination</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>double-click</td>
<td>Selects the entire identifier containing the cursor.</td>
</tr>
<tr>
<td>Alt+click a node, Ctrl+click a node</td>
<td>Inserts identifier of the node at the cursor position. Works when nodes are in the same diagram window as the definition being edited.</td>
</tr>
<tr>
<td>left-arrow ←, right-arrow →</td>
<td>Moves cursor one character left or right.</td>
</tr>
<tr>
<td>up-arrow, down-arrow</td>
<td>When Expression Assist is displaying possible identifier completions, moves the selection. Otherwise, moves the caret one line up or down.</td>
</tr>
<tr>
<td>Control+left-arrow, Control+right-arrow</td>
<td>Moves to the beginning or end of the next word or identifier.</td>
</tr>
<tr>
<td>Alt+Control+left-arrow, Alt+Control+right-arrow</td>
<td>Moves the cursor from the adjacent parenthesis to the next matching parenthesis, left or right.</td>
</tr>
<tr>
<td>Esc</td>
<td>Temporarily removes expression assist popup.</td>
</tr>
<tr>
<td>Ctrl+space</td>
<td>Temporarily show expression assist.</td>
</tr>
<tr>
<td>Ctrl+?</td>
<td>Toggles Expression Assist on or off for the field being edited. Note: On most keyboards, ? requires the Shift key, so this is really Ctrl+Shift+?.</td>
</tr>
<tr>
<td>Tab</td>
<td>When Expression Assist displays possible completions, inserts the first (or selected) one.</td>
</tr>
</tbody>
</table>

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:

\[ c \times \left( -\left( \text{Ln(Uniform(1f,1))} \right) \right)^{(1/k)} \]

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 \( X := -b\times\text{Sqrt}(b^2 - 4\times A\times C)/A \) { Positive quadratic root }
Creating or editing a definition

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 110).
- 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 in different modules.

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

- Select Paste Identifier from the Definition menu to open the Object Finder dialog (page 110).
- Select the function from its library in the Definition menu (page 112).

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.

A definition’s syntax check can reveal syntax errors (page 432). 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.

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

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

Arrow updating

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

The Expression popup menu

Click \texttt{expr} to see the \textit{Expression popup menu}. The \texttt{expr} menu shows the type of the definition, which is an empty expression in the following figure.

Use this popup menu to change the definition to one of several common kinds of expressions. The entries in this menu depend on the class of the node being defined.
Expression Assist provides context-sensitive help information as you type an expression. The information updates with each keystroke as you type, displaying function parameters and possible identifier or index label completions. Identifier completion can also save you keystrokes and reduce the risk of typos.

Enabling or disabling  Expression Assist is turned on or off from the Preference dialog (page 54). When the preference is off, you can still access the feature while typing an expression by pressing Ctrl+Space or Ctrl+Shift+. Ctrl+Space displays the help only once (but only if the Expression Assist has something to display at the current context of the caret), while Ctrl+Shift+? temporarily toggles the feature for the field being edited.

When the preference is on, it automatically appears as you are typing an expression into a Definition or Check attribute, changing with each keystroke. Occasionally the Expression Assist popups will block something else on the screen that you wish to see. When this happens, press the Esc key to temporarily remove the popups. They will reappear was you start typing again. To remove them for a span of keystrokes, press Ctrl+Shift+? to toggle the pops off and on.
**Identifier completion**  
As you begin to type an identifier, a listing of possible identifier completions appears.

In this example, the list of shows identifiers starting with “D”, the part of the identifier typed so far. At the top above the separator line appears inputs to the variable, for which arrows had been previously drawn. The **Standard** tab is selected, so that the more esoteric identifiers are not included. The **Advanced** tab includes object identifiers that are only relevant to advanced metaprogramming topics, such as attribute and module names. The mouse wheel can be used to scroll the list. At this point, pressing the **Tab** key would insert the first item, **Depreciation**, into your definition. As you continue to type, the list of possible completions narrows.

Clicking on an entry, or pressing the down-arrow to select the entry of interest and pressing **Tab** or **Enter** inserts the identifier into your expression.

**Function parameter info**  
When you are editing a parameter argument to a function, Expression Assist displays the list of parameter names and the function’s description.

The first line of the popup shows the parameters to the function being entered, with the current parameter being entered is shown in bold and optional parameters in italics. You can click on the parameter name to insert the parameter name with a colon into the definition for named-parameter syntax.

**Subscript help**  
When entering a subscript or slice operation, \( X[I=v] \) or \( X[@I=n] \), Expression Assist can in some cases provide a listing of the indexes of \( X \) when you are about to type the index name, \( I \). When it can do this, the list of possible indexes appears immediately after you type the left bracket. Expression Assist will only provide the list of indexes when it can do so without initiating
any evaluation, which means that \( X \) must be a single identifier (not a computed expression) and the value of \( X \) must have been previously evaluated and cached.

Similarly, when you get to the index value, \( v \), Expression Assist can in many cases display the set of possible index labels as an autocompletion list. Again, suggested values are only displayed when it can do so without initiating evaluation. This is generally possible for indexes defined as lists, or for computed indexes that have been previously evaluated.

Because these suggestions can be extremely helpful when typing definitions, you may find it useful to evaluate parent variables before you start typing the definition.

**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.

![Image of the Find button]

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.

![Image of the Object Finder dialog]

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 \( << >> \).

![Image of item function]

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.

## The domain of possible values

In addition to specifying a variable's definition, it is also a good practice, and often advantageous, to also specify the space of possible values allowed for the variable’s result. The \textbf{Domain attribute} is used for this purpose. In fact, it is a good practice to define the space of possible values before you start to specify the definition, since this can help to encourage clear thinking and ensure that you are defining a variable that is clearly and unambiguously defined.

The domain attribute specifies the values allowed for each cell in a variable's result. When the variable evaluates to an array result, the domain specification applies individually to each cell of the result. The domain does nothing to constrain the permissible dimensionality.

In addition to promoting lucid thinking during the model building process, once specified the domain provides Analytica and its user interface with numerous clues about how variables should be treated, and can be instrumental in detecting modeling errors.

### Making the Domain visible

By default, the domain attribute is not shown in the Object Window (except that in Analytica Optimizer, it is always visible for Decision variables). To make it visible for greater convenience, you can turn it on by placing a check next to Domain in the Attributes... dialog, found on the Object menu. You must be in edit mode to make this change.
Domain Types

On the domain attribute pulldown, several categories of domain types are presented:

- **Automatic**: Signifies that you have not specified the domain explicitly. When the domain is needed, Analytica will use heuristics to automatically infer the likely domain based on the information at hand. For example, when displaying probability distributions, it makes a difference whether a variable is continuous or discrete, since in the former a probability density graph should be displayed, while in the latter a probability distribution should be depicted.

- **Continuous**: Indicates that the variable is real-valued. When Continuous is selected, you may optionally specify the lower and upper bounds:

  ![Domain Attributes](image)

  If you elect to specify bounds, you should treat these as hard bounds, meaning the final value should never fall outside the indicated range. If, for example, you have an uncertain variable with a right tail, you would leave the upper bound blank. The bounds are inclusive.

- **Integer**: Indicates that the variable is integer-valued, with optional lower and upper bounds.

- **Grouped Integer**: This option shows up only for decision variables, and only in Analytica Optimizer. Each grouped integer variable belongs to a named group. The optimal value for cells falling within a given group must have a value between 1 and N, where N is the total number of cells belonging to the same group, and such that all cells belonging to the same group have a different value. This category is used solely for formulating certain types of optimization problems. See the Optimizer User Guide.

- **Boolean**: Indicates that the variable is 0,1 valued (false, true).
The domain of possible values

**Discrete** Indicates that the variable is discrete (as opposed to continuous), but not captured by the above Integer or Boolean categories, and where the set of discrete values cannot be enumerated explicitly. The discrete type pulldown allows you to designate the values further as being textual, numeric, or more Any (unrestricted).

When an input control is set to **Discrete Text**, quotation characters do not display around the current value, and whatever a user types is taken directly as text. No attempt is made to parse the entry as a number, variable name or expression.

Some uncertain quantities result in discrete numeric (but non-integer) outcomes. When **Discrete Numbers** is selected, probability distribution graphs will be displayed, not probability density graphs.

**Explicit Values** This option allows you to list the set of possible (allowed) values explicitly. As with discrete, you can indicate whether the values are all numeric or all textual.

**Copy From Index** Indicates that the set of allowed values are the set of values found in another index variable. When you select this option, you are asked to select the index. To change the source index, click on the index name.

**Expression** (Advanced) This view allows you to enter the domain as an Analytica expression, or to view the expression for the currently selected domain option. This provides a high degree of flexibility for advanced scenarios, allowing the domains themselves to be computed, array-valued, and conditional. Expression mode is also required to access certain less common parameters of the above domain types.

There are five special domain functions which are used to specify each of the above domain types in expression view, **Continuous()**, **Integer()**, **GroupedInteger()**, **Boolean()**, and **Discrete()**. The simplest domain expressions consist of a single call to one of these functions, e.g.:

`Continuous(lb:0,ub:1000)`

These functions can be combined in arbitrary ways using Analytica expression syntax to create computed domains. For example:

`If Col="Id" then Integer(lb:1) Else Continuous(lb:low,ub:low+range )`

An expression can compute a set of discrete values as an unindexed list, or it can wrap them in a call to the **Discrete()** function, e.g.:

`Discrete("Accounting", "Engineering", "HR", "Marketing", "Sales")`

Each standard domain type considers null values to be acceptable. You can override this and indicate that null is not acceptable by specifying an optional **nullOk** parameter as **false** in expression view, e.g.:

`Continuous(nullOk:false)`

Each of the five functions are described in “Domain Functions” below.

As a stylistic guideline, domain computations should be designed to be computationally simple. For example, although possible, it is generally not a good idea to employ a complex and time consuming algorithm that produces a highly informed and nearly tight lower bound. Instead, utilize a “hard” bound that is relatively easy to obtain and save the complex algorithms for variable definitions.

**Uses of Domain**

When the domain is specified, Analytica makes use of the information in numerous ways.

**Bounds checking** When the value of a variable is computed, it is compared with the domain, and if the result is not consistent with the domain, an out-of-range warning is displayed. Bounds violation are often effective in detecting modeling mistakes. The **Check value bounds...against domain bounds** preference setting disables this feature (see “Preferences dialog” on page 54).
Chapter 9 Creating and Editing Definitions

The domain of possible values

Detecting Logical Errors
The domain enables Analytica to detect and warn of various inconsistencies in your model that may indicate an error in your modeling logic. Sometimes inconsistencies arise when one part of a model is changed, but you forget to change another part of your model to be consistent. Analytica will display warning messages when these situations are encountered, which can help to cut down on undetected modeling errors. An example is the following. Suppose an expression contains the following sub-expression

If X="Yes" Then...

where the domain of X is Boolean(), meaning that a value of X will be 0 or 1, not “Yes” or “No”. Analytica issues a warning (at evaluation time) that reveals this error in logic, which is made possible as a result of specifying the domain.

Interpretation of uncertainty
The computation of probability distributions or cumulative probability distributions requires the underlying sample to be interpreted as being either continuous or discrete. Although Analytica can usually correctly infer this from heuristics that make use of various other clues, an explicit domain provides this information directly.

Label ordering in graphs
In parametric graphs (e.g., XY graphs, histograms of discrete quantities, etc), the ordering for labels on the graph axes is not always obvious. In the absence of guidance from the domain, Analytica will usually use a lexical sort order; however, if you want to control the ordering of possible values, you can do so by using an explicit values domain. The order that possible values are listed is then used as the label ordering on graphs.

Search Space in Optimization
In Analytica Optimizer, the domain of decision variables defines the search space for constrained optimization problems. See the Optimizer User Guide.

List of Options for a self-Choice
A node with a definition of Choice(Self,...) uses the set of possible values listed in the domain as the options that appear on the Choice pulldown.

Indexes for a DetermTable or ProbTable
A DetermTable specifies a result for every possible value across a set of parent variables. The result of a determTable selects out the values that match the computed value. The domain is used to specify what the set of possible values are for each of these parent selector variables. A ProbTable behaves like a determ table, with its own domain included as a table index.

Validation of user input
When a user of a model enters values that are in disallowed by the domain, Analytica can provide the user with immediate feedback that his entry is out of the allowed range. This can be utilized in input controls or edit table cells.

Display and parsing of textual values
The domain influences how values are shown in input fields or cells. Should quotes be displayed around textual values? They will be if the domain indicates that both numeric and textual values are acceptable, but not if the variable is restricted to textual values. What if the user enters text that matches an existing variable name, or text that could be interpreted as a number? Should it be treated as a purely textual entry, or should it be parsed? Again, this is determined by the domain. When the domain is restricted to purely textual values, the entry is treated purely as text.

Domain Functions

(Advanced) The functions in this section are used in the Expression view of the Domain attribute to specify the space of possible values for a variable.

It would be highly unconventional to use any of these domain functions from outside of the domain attribute; nevertheless, it is possible to do so. When a domain function is evaluated, the return value is in the form of a parsed expression, consisting of a function call to the indicated function, but with each parameter evaluated and replaced by the literal values. Treating this result as an atomic value, expressions can assemble heterogeneous arrays of domain types that vary across multiple indexes.

Continuous(lb,ub,orZero,nullOk)
Used to specify a continuous domain. The optional lb and ub parameters specify lower and upper bounds respectively, and the expressions specified for lb and ub may evaluate to an array when the bounds vary along an index. The orZero parameter may be set to true to specify a semi-continuous domain, in which zero is acceptable along with any value consistent with lb and ub. By default, null values are considered acceptable, but when nullOk is specified as false, then null value will trigger out-of-bounds warnings.
Chapter 9  Creating and Editing Definitions

Checking for valid values

Integer(lb,ub,nullOk)
Used to specify an integer domain. The optional lb and ub parameters specify lower and upper bounds respectively. Values equal to the bounds are considered to be within the acceptable range. The nullOk parameter can be set to false to reject null values as out-of-range. When not specified, nullOk defaults to true.

GroupedInteger(groupName,..,nullOk)
Used to specify a grouped integer domain, in which all cells belonging to the same group are required to have a unique value in the final solution found by optimizer. This option is generally only used from Analytica Optimizer, and only in the context of an integer programming problems.

Boolean(nullOk)
Indicates a 0-1 valued domain. Any value other than 0, 1 or Null will trigger an out-of-bounds warning. When the optional nullOk parameter is specified as false, then Null will also trigger an out-of-bounds warning.

Discrete(x1,x2,x3,..., type, nullOk)
This function is used to encode both a Discrete Domain with unknown values, and the Explicit Values domain where the values are explicitly listed. For an Explicit Values domain, the values are listed as the parameters to the function, e.g.:

Discrete(2,3,5,7,9,11,13,17)

For an implicit discrete domain, no allowed values are listed. Either usage may include the type parameter, which must be specifying using the named parameter convention and may be specified as any of the following values, or a list of any of the following values: "Any", "Number", "Text" or "Handle", e.g.:

Discrete(Type:"Text")

The type setting is used by the user interface components when labels are being entered to determine whether the values should be parsed. Expressions are allowed in GUI list cells when the type is "Any".

Null values are also accepted without generating an out-of-bounds warning unless nullOk:False is specified.

Checking for valid values
Validity checking can be used to detect errors that get introduced into a model, or to validate value entered by users. When changes are made to parts of a model in the future, validity checking can often catch dumb mistakes that get inadvertently introduced.

Domain Range Checks
Computed values are compared against the space of possible values specified in the domain attribute (see “The domain of possible values” on page 113). This validation occurs as long as the Check value bounds ... against domain bounds preference is turned on (it is on by default).
Thus, validity testing requires only that you take the time to specify the appropriate domain attribute and bounds, something that is a recommended good practice anyway.
Domain range checks are also applied to input controls when a literal value is entered:

Percent damage: 25

Validation of edit table cell entry

Domain bounds are also applied to edit table cells. Cells that are not consistent with the domain bounds display with a pink background, and when values are entered, a tool tip describes the problem:

Check Attribute

The Check attribute provides an even more flexible mechanism for validating model results. In its simple usage, it can be utilized for range checking in the same fashion as the domain attribute, for example, to check that the value of Percent_damage is between 0 and 100, set its check attribute:

\[
\text{Check} := \text{Percent\_damage} \geq 0 \text{ AND } \text{Percent\_damage} \leq 100
\]

If the check attribute evaluates to False, whenever the variable is evaluated, it shows a warning dialog and the opportunity to edit the definition. Use of the check attribute has one further advantage that the domain attribute does not have -- you can customize the error message that displays when the criteria is violated (see “Custom error messages” on page 120).

Unlike the domain, the Check attribute can also be used to check consistency across multiple cells in results, or even across multiple result variables. For example, in some cases you might know that a certain invariant is required to hold, so that a violation of that invariant would signal a problem with the model. Such an invariant can apply array operations and multiple variables, e.g.:

\[
\text{Check} := C \leq \text{Max}(A,I) + \text{Sum}(B,I)
\]

While the check attribute has some flexibility not available from the domain attribute, when validating against simple bounds, the domain attribute tends to be more convenient.

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” on page 343).
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.

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

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

If the **Check** expression refers to another variable, it makes a dependency from the variable being checked to the variable mentioned. It 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.
Chapter 9  Creating and Editing Definitions

Checking for valid values

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 \( \text{Sum}(\text{Self}, \text{Projects}) < 5 \), 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 \( \text{Error()} \) function (page 388) in the check, it displays the message you pass to \( \text{Error()} \) 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 54) from the Edit menu. If you want to disable only Check attribute validation, but leave domain validation in tact, then uncheck the Check value bounds..against Check attribute preference. Validation preferences are on (checked) by default.
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.

To enable end-users to run your model, you may decide to publish your model to the Analytica Cloud Player and invite your colleagues to view it from their web browser. When publishing to the web, additional styles settings can be configured to make your interface more user-friendly from the confines of a web browser. The second part of this chapter, “Sharing your model” on page 131, shows you how to publish your model to the Cloud.
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 2, “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” on page 19). 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” on page 51).

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 248).

**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 117); 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 123.

List

A list or list of labels displays as a List button. See “Creating an index” on page 169.

Edit table

An edit table displays as an Edit Table button. See “Defining a variable as an edit table” on page 176.

Probability distribution

A probability distribution displays a button with the name of the distribution. See “Defining a variable as a distribution” on page 248.

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 variable, 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 l=Self and n=1. 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 183).
Creating Interfaces for End Users

Using output nodes

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) is created with a list of Explicit Values, with one element in the list.

To change the domain to List of numbers, press the Domain value type popup to the right of the Explicit Values pop-up and select Numbers.

Edit the list of values as you would edit a list of labels or list of numbers (see page 171).

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 29).

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 77).

Resizing controls

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

You can also drag the left edge of the control field, button, or menu left or right to change its width. This is especially useful for choice menus when you 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 52). 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 75 and “Recoloring nodes or background” on page 73). 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.
Creating input and output nodes in a form module

7. Select Form from the menu.

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 50). 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.
Adding icons to nodes

Chapter
Creating Interfaces for End Users

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+x) command or the **Delete** key.
Chapter 10 Creating Interfaces for End Users

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 4.5 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 Cloud Player (ACP) 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 ACP 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 75) 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 75) 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 75) 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 75) 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:

```plaintext
{ 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
<?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>
```
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.
Sharing your model

When you distribute your model to others, one option is to provide them with your model file, and ensure that they have Analytica installed on their computer. To use the model, they’ll need to install the free Analytica Player, Analytica Power Player, or a full Analytica edition on their Windows-based computer. A second option is to publish your model to the Analytica Cloud Player (ACP) and send them a URL to your model. When you do this, they only need a Flash-enabled web browser, and can view your model even from non-Windows computers. When you build a model with the intention of sharing it over the web, additional stylistic considerations that are discussed in this section can have a big impact on usability.

Analytica Player

The free Analytica Player allows users to browse models, change inputs, and evaluate and view results. Users can only make changes variables with input nodes, and cannot save their model after making changes to inputs. They also cannot utilize features that are not available with Analytica Professional such as database access, huge indexes, or Monte Carlo runs with more than 32000 iterations (see Chapter 23, “Analytica Enterprise” on page 391).

To distribute your model to someone who already has Analytica installed, either the free Player or another edition, you can simply attach the model to an email. When the user clicks on the attachment and selects Run, the model opens in Analytica or Analytica Player. You can also place the model on a web server with a hyperlink on web page using HTML such as:

Please view `<a href="myModel.ana">my Analytica model</a>.

If you do not have Analytica installed on your computer, please `<a href="http://www.lumina.com/products/analytica-editions/free-player/">download and install the free Analytica Player</a>` first.

When the user who has Analytica or Analytica Player installed clicks on the model link, Analytica launches with the model loaded.
When your model is spread across multiple linked modules, you will need to send all these module files to your end-user and have them save them all (usually in the same directory) before launching the main file. Be sure to tell them which model file is the top-level model that they should launch. To avoid the multiple-file complication, a better option is to save a copy of your model in a single file and send or post that single file (the linked-file structure is not preserved when you do this, hence, you send it as a copy). To do this, select **File → Save A Copy In...** and check the **Save everything in one file by embedding linked modules** checkbox.

Spreadsheet functions such as `SpreadsheetOpen`, `SpreadsheetCell`, etc., and OLE linking are not available in Cloud Player.

### Testing in Player

The user experience for Player users is essentially the same as viewing the model in Analytica Professional while being locked in browse mode. To experience your model in Analytica Player yourself before handing it off to your users, select **Help → Update License...**, and select `analytica_freeplayer` from the **License ID** dropdown. After pressing OK, you’ll need to restart Analytica. Repeat the process to select your other license when you are finished. If you do this often, you may wish to create a Windows desktop shortcut to eliminate the need for the above steps. The shortcut should run the following command line:

```bash
Analytica.exe /lic:analytica_freeplayer
```

The Analytica Power Player extends Player with the ability to utilize Enterprise-edition features, and with the ability to save models. Although a Power Player is not free, it is less expensive than the license for a full Professional edition. If your users require live access to your ODBC databases, huge arrays, or the ability to save their inputs, then Power Player should be used. Optimizer-enabled Power Player licenses are also available.

### Analytica Cloud Player

By publishing your model directly to the Analytica Cloud Player (ACP), you eliminate the need for your end-users to install Analytica Player on their computer before viewing your model. This also makes it possible for users on non-Windows platforms to use and view your model. The model is uploaded directly to Lumina’s Cloud server, and you simply send your end-users an invitation to view your model with a URL link to access the model.

#### ACP account types

Signing up for an **ACP individual account** is free. You can upload models to your account at any time, and as long as your Analytica support is active, your account allows 25 user sessions per month at no cost. You also receive 25 sessions credits to be used in the first 12 months when you sign up for an ACP individual account, even if you do not have active support. If you foresee higher user loads, you can pre-purchase additional session credits.

**ACP group accounts** are also available, and are appropriate for collaborative projects and higher usage situations. Group accounts include the ability for end-users to save changes to inputs in snapshot files, to manage multiple users and multiple projects, and to control access permissions to different models. Group accounts involve a monthly or annual subscription cost, and include a high level of user sessions.

For further information on ACP accounts types, please visit

```url
http://www.lumina.com/products/analytica-cloud-player/
```

---

1. A user-session starts when you or someone else views one of your published models in ACP. A single user-session allows up to 60 seconds of CPU computation time within the same session. Closing one model and re-opening or opening another is considered part of the same user session.
Sharing your model

Chapter 10

Creating Interfaces for End Users

Uploading / Publishing

To publish a model to ACP, open the model in Analytica, then select File → Publish to cloud.... If you already have an ACP account set up, the following dialog appears:

The ACP model name may optionally be different from your local file name, which you may take advantage of if you keep multiple local file revisions, or conversely want to publish different revisions. When you press Publish, the model is uploaded to the cloud. You have three options for what happens once the upload completes:

- Open ACP model folder page
- Open model in ACP
- Do nothing

Opening the model in ACP consumes a session credit, while viewing the model folder page does not. From the folder page, you have the option of launching the model.

You can also jump to the ACP model folder page by selecting File → Manage published models....

Setting up an ACP account

The first time you select Publish to cloud..., the dialog will open in the Configure account tab.

If you do not already have an account, click the sign up link to sign up for a free ACP individual account. Once you have an ACP account, enter your account information and press OK. For detailed steps on signing up for an ACP account, please see Chapter 8 of the Analytica Tutorial.

Designing models for viewing in web browsers

Some models are ill-suited for use from a web browser. When you create your model with the intention of eventually publishing it to Cloud Player, keeping certain stylistic and pragmatic considerations in mind can help you achieve a more effective presentation once the model is pub-
Limit computation times

People expect each operation in a web application to complete quickly. When operations take more than about 60 seconds, users will typically leave or close the page. ACP itself enforces a 60 second maximum duration on any single computational step (you can configure this maximum in an ACP group account, even though it isn’t highly recommended for these stylistic reasons). Models that inherently involve multi-minute computations or longer are not well-suited for publishing on the Cloud.

The ACP server is not a super computer, and is not intended for the evaluation of computationally excessive models. The server(s) are shared by all ACP users.

Use small diagrams

Even people with huge monitors make use of narrow web browser windows. A standard guideline is that web-apps should be a maximum of 800 pixels in width to display fully in the browser window. While ACP does not limit the actual displayed diagram size (scroll bars appear when required), you should keep diagrams within this size range to eliminate the need for users to scroll when viewing your diagrams.

Achieving small diagrams usually translates to keeping influence diagrams to 15 nodes or less, not exceeding 5 nodes wide. For input/output panels, limit these to two columns. It is also desirable to limit the length of diagrams and input/output panels similarly so that they be fully visible without scrolling for most users. These guidelines are highly recommended even when models will not be viewed on the web, since they also correspond to the amount of visual information the human brain can assimilate holistically in a glance.

To attain small diagrams, you should make use of modules to structure your models hierarchically. Each module should “tell” a small self-contained story. The default ACP presentation includes an outliner bar along the left side of the screen (which you can disable using an ACP style setting, if desired). Although this consumes some valuable screen width, it helps to make the module hierarchy more visible and accessible, and thus help you feel more comfortable with keeping each module small.

Single window design

Desktop Analytica uses a Multiple Document Interface (MDI) paradigm, in which you can have multiple windows open simultaneously, side-by-side or overlapping. This provides a tremendous flexibility to display many items of information simultaneously in flexible arrangements, with other information close at hand. However, ACP model users view one diagram, one result, or one table at a time. It thus becomes important to cluster information that is intended to be viewed together within the same diagram. Make use of ACP layout to help accomplish this.

Utilize layout styles

ACP provides several layout options that are not yet available in Analytica 4.5. These include tabbed module navigation, embedded tables and graphs, re-usable embedded space for result views, an outline tree or hierarchy header, control over what surrounding interface components display, and additional control over node-level appearance.
The full set of layout options and how to specify the settings are beyond the scope of this manual. The reference guide for these is on the Analytica Wiki, see CloudPlayerStyles attribute values, ACP rendering tables and graphs on the diagram, and Features of ACP not available in Analytica.

Use the Style Library
The ACP Style Library is a tool you can use when defining the ACP styles. It provides a set of control panels to select options with immediate preview displays to help you understand what each setting does. The Style Library is packaged as an Analytica library file, which you add to your model, then access the control panels by opening the library diagrams from within Analytica.

You can make use of the styles without using the Style Library. Styles are specified in the Cloud Player Styles attribute, as described at CloudPlayerStyles attribute values. However, the Style Library makes these far easier to use.

Instructions for obtaining the latest version of the ACP Style Library, and for using it, are located on the Analytica Wiki at http://wiki.lumina.com/index.php/ACP_Style_Library.

Summary
You can use input and output nodes to direct users of your model to relevant input and output variables. These nodes can be arranged into control panels for convenient presentation.

Only variables with an associated input node can be modified by users of Analytica Player, Power Player or Cloud Player.

Analytica Player is free to install, and hence provides way to share your models with others who do not have a licensed edition of Analytica. Power Player adds to Player functionality with an ability to save models, query databases, evaluate functions that require the Enterprise edition, and compute with huge indexes or sample sizes larger than 32000.

Analytica Cloud Player (ACP) is a convenient way to share models over the internet, and is usable by people who have not installed any edition of Analytica or Analytica Player, as well as people on non-Window’s platforms. ACP accounts are free, and a moderate level of usage is free as long as you keep your support active.

When sharing models from the Cloud Player, additional stylistic considerations are important for providing users of your model with a satisfactory browser-based experience. These include limiting evaluation times, keeping diagrams small, and making effective use of diagram real estate through embedded tables and graphs, tabbed panes and hierarchy headers. The Style Library provides an interface for selecting ACP styles.
The definition of each variable is an expression, such as

\[- \frac{B + \sqrt{B^2 - 4A*B}}{2A}\]

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 + - / \(^\wedge\) < > = 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 22, “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 78), 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” on page 79 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. Dates, date-times, or times-of-day must be typed in a very specific format when the number format is not set to date, or when they appear within an expression with other elements. See “Date and Time Values” below.

Integers, Fixed Point and Floating Point

Integer values that you enter between -9,223,372,036,854,775,807 and 9,223,372,036,854,775,807 are represented internally as integers, and numbers that you enter between -9,223,372,036,854.775807 and 9,223,372,036,854.775807 with 6 or fewer decimals are represented internally with full precision as fixed-point numbers. Numbers with more than 6 decimal digits of precision or outside these ranges are represented using 64-bit binary floating point numbers.

Precision

Integer and fixed-point numbers are exact representations of the numeric value, and hence, Analytica prefers these representations when possible. Floating point values are, in general, approximate values, accurate to about 15 significant digits. Many base-10 floating point numbers cannot be represented exactly in binary, and hence may have some rounding error in the 16th digit.

Round-off error

Some calculations, especially those that involve small differences between large numbers or large numbers of additions, might result in less precision than this maximum, or may magnify existing round-off error.

When calculations are performed with integer or fixed-point numbers, the result is represented internally as an integer or fixed-point number if Analytica is able to do so, which it usually is able to do with addition, subtraction, and multiplication. These results have zero round-off error. With many other operations, such as division or functions like \(\sqrt{()}\), this is not possible, so the result is floating point number. Operations involving floating-point numbers result in floating-point numbers even if the result appears to be an integer, since the result is no longer guaranteed exact, and always subject to the possibility that round-off error could accumulate.

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:

\[1/10^{-1000} \rightarrow 0\]

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

\[10^{-1000} \rightarrow \text{INF}\]
Complex Numbers

To enter the imaginary part of a complex number, append a lowercase j or i to the number, as in these examples:

1j, -1j, -2M+23Kj, 4-7e-8j

In the 4-7e-8j example, the imaginary part is -7e-8 (i.e., -70n). There cannot be a space between the digit and the j, and if you want just the imaginary number, you must write 1i or 1j, since without the preceding 1, you would be referring to a variable named i or j.

The real and imaginary components are both represented internally as 32-bit floating point numbers, each spanning the range from 3.4e+38 to 3.4e-38 with about 7 digits of precision. A complex number consumes the same amount of memory as a real number, but sacrifices range and precision to do so. As a result, you will experience greater round-off error in calculations that use complex numbers.

Date and Time Values

Dates and times are treated as numeric quantities, denoting the number of days elapsed since the date origin (usually 1-Jan-1904, or 1-Jan-1900 if the Use Excel date origin preference is checked). Dates can be entered directly into an expression, at any point in an expression and regardless of the number format setting, using the syntax d-MMM-yyyy, such as 27-Jan-2010, using a 3-letter English-language month abbreviation. Times can be entered in the format hh:mm:ss [tt], such as 10:27:00pm or 22:27:00, which becomes a fractional value between 0 and 1. A combined date-time can be entered as, e.g., 27-Jan-2010 10:27:00pm. In the general case, you cannot enter other date formats within an expression. For example, 1/27/2010 would be interpreted as two divide operators, (1/27)/2010, and regional (non-English) month names, e.g., 27-Ene-2010, are not recognized.

You can enter dates and times in more general formats, and in region/culture-specific styles, when the number format is set to Date and the date, time or date-time is the only element appearing in the expression. In this case, Analytica recognizes almost all commonly used formats, such as 27-January-2010, 1/27/2010, 1-27-2010, or Jan 27, 2010. When you enter a date in a format such as 1/2/2010 or 1-2-2010, the date is interpreted based on your regional setting. For example, in the US, these would parse to 2-Jan-2010, while in Europe they would generally parse as 1-Feb-2010. Non-English month names are also recognized in languages that use characters only from the standard Analytica character page (which includes most western European languages). For example, 27-Enero-2010 is recognized if you are in Spain.

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 78).

Text values

You specify a text value by enclosing text between single quotes, or between double quotes, for example:
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” on page 169 and “Expression view” on page 170) or edit table (see “Defining a variable as an edit table” on page 176). Analytica displays text values in results without the enclosing quotes. Also see “Converting number to text” on page 145.

For comparison and sort order for text, see “Alphabetic ordering of text values” on page 141.

For functions that work with text values, see “Text functions” on page 224.

For converting between numbers and text, see “Numbers and text” on page 145.

### 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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>x + y</td>
<td>plus</td>
<td>3+2 ( \rightarrow ) 5</td>
</tr>
<tr>
<td>x - y</td>
<td>binary minus</td>
<td>3 - 2 ( \rightarrow ) 1</td>
</tr>
<tr>
<td>-x</td>
<td>unary minus</td>
<td>-2 ( \rightarrow ) -2</td>
</tr>
<tr>
<td>x*y</td>
<td>product</td>
<td>3*2 ( \rightarrow ) 6</td>
</tr>
<tr>
<td>x/y or x÷y</td>
<td>division</td>
<td>3/2 ( (= \frac{3}{2}) ) ( \rightarrow ) 1.5</td>
</tr>
<tr>
<td>x**y</td>
<td>to the power of</td>
<td>3^2 ( \rightarrow ) 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4^{\frac{1}{2}} ( \rightarrow ) 2</td>
</tr>
</tbody>
</table>

#### Comparison operators

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Examples ( \rightarrow ) (1 = true, 0 = false)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>2 &lt; 2 ( \rightarrow ) 0 \text{ or } 'A' &lt; 'B' ( \rightarrow ) 1</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>2 &lt;= 2 ( \rightarrow ) 1 \text{ or } 'ab' &lt;= 'ab' ( \rightarrow ) 1</td>
</tr>
<tr>
<td>=</td>
<td>equal to</td>
<td>100 = 101 ( \rightarrow ) 0 \text{ or } 'AB' = 'ab' ( \rightarrow ) 0</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>100 &gt;= 1 ( \rightarrow ) 1 \text{ or } 'ab' &gt;= 'cd' ( \rightarrow ) 0</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>1 &gt; 2 ( \rightarrow ) 0 \text{ or } 'A' &gt; 'a' ( \rightarrow ) 1</td>
</tr>
</tbody>
</table>
Alphabetic ordering of text values

When applied to text values, the comparison operators, >, >=, <=, and <, use a region-specific collation order which takes into account conventions used in your own language and country. The specific collation order used is determined by the TextLocale system variable. If you want to change the collation order, bring up the Object Window for TextLocale by selecting it from the Definition→System Variables menu, with nothing selected and while in edit mode.

In general, the ordering of text may be different from the ordering implied by the numeric ASCII codes of the characters. For example, ‘de’ < ‘dé’ < ‘df’, whereas the ASCII code for ‘é’ comes after the ASCII code for ‘f’. If you want text comparisons to use simple ASCII ordering, you can set TextLocale to be ANSI.

Generally, comparisons follow the ordering you would find in a dictionary. For example:

'Analytica' < 'Excel'  \(\rightarrow\) 1 (True)

In almost all collation orders, digits precede letters:

'9' < 'A'  \(\rightarrow\) 1 (True)

The comparison operators are case-sensitive, so that uppercase letters precede lowercase letters:

'Analytica' > 'excel'  \(\rightarrow\) 0 (False)

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

\[
\text{TextUpperCase('Analytica')} < \text{TextUpperCase('excel')}  \rightarrow 1 \text{ (True)}
\]

When two text values do not contain identical character sequences, they are considered to be unequal, even if they represent the exact same text. This can happen in Unicode with combining characters. For example, an accented ‘é’ can be either \(\text{Chr(233)}\) or \(\text{Chr(769)}\&'e'\), where the Unicode \(\text{Chr(769)}\) is the combining acute accent, which modifies the character that follows. Although these alternate representations of ‘é’ are non-equal, they will have the same inequality comparisons with all other text, as if they are as closely adjacent in the sort order as is possible.

SortIndex(d, i), Sort(d,i) and Rank() also use the same collation ordering to determine the ordering of text values. Each also has an optional parameter to compare them in a case-insensitive fashion.

Logical operators

Logical operators apply to Boolean values and produce Boolean values.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b_1 \text{ AND } b_2)</td>
<td>true if both (b_1) and (b_2) are true, otherwise false</td>
<td>(5&gt;0 \text{ AND } 5&gt;10)  (\rightarrow) False</td>
</tr>
<tr>
<td>(b_1 \text{ OR } b_2)</td>
<td>true if (b_1) or (b_2) or both are true, otherwise false</td>
<td>(5&gt;0 \text{ OR } 5&gt;10)  (\rightarrow) True</td>
</tr>
<tr>
<td>(\text{NOT } b)</td>
<td>true if (b) is false, otherwise false</td>
<td>(\text{NOT } (5&gt;0))  (\rightarrow) False</td>
</tr>
</tbody>
</table>

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
.Then:
\[ \text{Irr}(\text{Payments, Time}) \]  
User's Irr function
\[ ::\text{Irr}(\text{Payments, Time}) \]  
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
15. Or
16. & (text concatenation)
17. :=
18. If ... Then ... Else, Ifonly ... Then ... Else, Ifall ... Then ... Else
19. 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:
\[ \text{IF } d + e < f ^ g \text{ or } a \text{ and } b > c \text{ THEN } x \text{ ELSE } y + z \]
is interpreted as:
\[ \text{IF } (((d + e) < (f ^ g)) \text{ or } (a \text{ and } (b > c))) \text{ THEN } x \text{ ELSE } (y + z) \]

**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 \( X := 1 \text{M} \)

Variable \( Y := 1 \)

IF \( X > Y \) THEN \( X \) ELSE \( Y \rightarrow 1 \text{M} \)

returns the larger of \( X \) and \( Y \).

It is possible to omit the ELSE clause:
Chapter 11 Using Expressions

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 167.

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 14, “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 353).

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. When x in complex, returns the magnitude, see “Complex number functions” on page 232.

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

**Ceil(x,digits)** Returns the smallest number with the indicated digits to the right of the decimal that is greater than or equal to x.

Ceil(Pi,4) → 3.1416    Ceil(-12345,-2) → -12300

**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

**Floor(x,digits)** Returns the largest number with the indicated number of digits past the decimal that is less than or equal to x.

Floor(Pi,4) → 3.1415    Floor(-12345,-2) → -12400

**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

**Round(x,digits)** Rounds the value of x to the indicated number of decimal digits to the right of the decimal point.

Round(Pi,1) → 3.100    Round(Pi,3) → 3.142
Round(14243.4,-2) → 14200

*Note:* The Number Format setting determines how many digits are included when a number is displayed, while Round(x,digits) returns a new rounded number so that the rounded value can be used in subsequent computations.

**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 unless the system variable EnableComplexNumbers is set.

Ln(150) → 5.011       Ln(Exp(5)) → 5

**Logten(x)** Returns the logarithm to the base 10 of x, which must be positive unless the system variable EnableComplexNumbers is set.

Logten(180) → 2.255    Logten(10 ^ 30) → 30

**Sign(x)** Returns -1 when x is negative, 1 when x is positive, 0 when x is zero, and NaN when x is NaN.

Sign(-15.2) → -1      Sign(7.3) → 1
Sign(0) → 0           Sign(0/0) → NaN

**Sqr(x)** Returns the square of x.

Sqr(5) → 25           Sqr(-4) → 16

**Sqrt(x)** Returns the square root of x. x must be positive unless the system variable EnableComplexNumbers is set.

Sqrt(25) → 5          Sqrt(-1) → NaN
Sqrt(-1) → 1j         *when EnableComplexNumbers is set*

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

Mod(7, 3) → 1         Mod(12, 4) → 0
**Factorial(x)** Returns the factorial of \( x \), which must be between 0 and 170.

\[
\text{Factorial}(5) \rightarrow 120
\]

\[
\text{Factorial}(0) \rightarrow 1
\]

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.

\[
\cos(180) \rightarrow -1
\]

\[
\cos(-210) \rightarrow -0.866
\]

\[
\sin(30) \rightarrow 0.5
\]

\[
\sin(-45) \rightarrow -0.7071
\]

\[
\tan(45) \rightarrow 1
\]

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

\[
\text{Arctan}(0) \rightarrow 0
\]

\[
\text{Arctan}(1) \rightarrow 45
\]

\[
\text{Arctan}(\tan(45)) \rightarrow 45
\]

See also “Arccos(x), Arcsin(x), Arctan2(y, x)” on page 231.

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

\[
\text{Degrees}(\pi/2) \rightarrow 90
\]

\[
\text{Degrees}(-\pi) \rightarrow -180
\]

\[
\text{Radians}(-90) \rightarrow -1.57079633
\]

\[
\text{Radians}(180) \rightarrow 3.141592654
\]

### 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:

\[
123456789 \& '' \rightarrow '123.5M'
\]

\[
123456789 \& '' \rightarrow '$123,456,789.00'
\]

\[
'The date is: ' \& 38345 \rightarrow 'The date is: Thursday, December 25, 2008'
\]

**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.

You can also control the format used when converting numbers to text with the \NumberToText\ function described on page 227.

#### Converting text to number
You can use the \Evaluate()\ function to convert a text representation of a number into an actual number, for example:

\[
\text{Evaluate}('12350') \rightarrow 12.35K
\]

\Evaluate()\ (page 388) 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 356). For example, you could define a user-defined function such as:

\[
\text{ParseNum}(X: \text{Coerce Number}) := X
\]
Exception values INF, NAN, and NULL

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

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

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:

\[10^{1000} \rightarrow \text{INF}\]

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

\[1/0 \rightarrow \text{INF}\]

INF can be positive or negative:

\[-1 \times 10^{1000} \rightarrow -\text{INF}\]

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

\[\text{INF} + 10 \rightarrow \text{INF}\]
\[\text{INF}/0 \rightarrow \text{INF}\]
\[10 - \text{INF} \rightarrow -\text{INF}\]
\[100/0 = \text{INF} \rightarrow \text{True}\]

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:

\[\text{INF} - \text{INF} \rightarrow \text{NAN}\]
\[0/0 \rightarrow \text{NAN}\]
\[\text{INF}/\text{INF} \rightarrow \text{NAN}\]
\[\text{Sqrt}(-1) \rightarrow \text{NAN}\]
\[\text{ArcSin}(2) \rightarrow \text{NAN}\]

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 147).

Any arithmetic operation, comparison, or function applied to NaN returns NaN:

\[0/0 <> \text{NAN} \rightarrow \text{NAN}\]

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 148).

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:

\[\text{Index I := 1..5}\]
\[X[\text{I=6}] \rightarrow \text{Null}\]

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 **NANs**, and so Analytica generates no warning when `Z` is evaluated.

You can also make use of the return value, even if it might be errant, as in the following example:

```
VAR x := Sqrt(y);
IF IsNaN(x) THEN 0 ELSE x
```

The 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 378 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, NAN, or complex.

- 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

**IsNull(x)**
To test if x is exactly Null. Returns false if x is an array.

**x = NULL**
To test if an atomic x is Null. When x is an array, returns True or False for each element of the array.

**IsUndef(x)**
Returns True if atomic x is Null or the internal value Undefined (usually indicating uncomputed). When x is an array, returns True or False for each element of the array.

**IsRealNumber(x)**
Returns True when x is a real number, including a boolean, date or INF. Returns False when x is a complex number or NAN.

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

**IsHandle(x)**
Returns True if x is a handle to an Analytica object.

**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 12

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:

- Conditional expressions (page 167)
- Creating an index (page 169)
- Editing a list (page 171)
- Functions that create indexes (page 172)
- Defining a variable as an edit table (page 176)
- Editing a table (page 178)
- Selecting a slice or subarray (page 181)
- Choice menus in an edit table (page 184)
- Shortcuts to navigate and edit a table (page 185)

For more, see Chapter 13, “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 187, 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 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 74.

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 177). 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:

\[
\begin{align*}
\text{Miles per year} & := 10K \\
\text{Fuel price} & := 3.00 \\
\text{Fuel cost} & := \text{Fuel price} \times \text{Miles per year} / \text{Miles per gallon}
\end{align*}
\]

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.

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.

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.
Chapter 12 Arrays and Indexes

Introducing indexes and arrays

The value of Year is now the sequence of years from 2008 to 2012. (See “Creating an index” on page 169 and “Functions that create indexes” on page 172 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 (CAGR) of 8% per year:

\[
\text{Fuel_price_cagr} := 8\%
\]
\[
\text{Fuel_price} := 3.00 \times (1 + \text{Fuel_price_cagr})^{(\text{Year} - 2008)}
\]

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 \((1 + 8\%)\) 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.
Chapter 12 Arrays and Indexes

Introducing indexes and arrays

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).

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 Repeat Values. 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.

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.
Combine two 2D arrays with the same indexes

Let’s look at the value of *Operating_cost* again.

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:

\[
Miles_{\text{per year}} := [5K, 10K, 15K]
\]

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 38.

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*. 

Introducing indexes and arrays

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 27.

**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:

\[
\text{Variable Total_operating_cost := Sum(Operating_cost, Year)}
\]
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`.

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 192). 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`:

```
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] → 1775
```

For more, see “x[i=v]: Subscript construct” on page 181.

### 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`).
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.

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.

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 167 for more.)
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 173).

The result for Cost_summary is four-dimensional, adding a new index, also labeled Cost_summary, showing the variables in the list.
Introducing indexes and arrays

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.

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 IndexValue() function (see "IndexValue(i)" on page 214).

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:

$$\text{Variable Total_cost_by_year := Sum(Cost_summary, Cost_summary)}$$

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:

$$\text{Variable Total_cost := Sum(Cost_summary, Year)}$$

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:

$$\text{Variable Total_miles := Sum(Miles_per_year, Year)}$$
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.
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, offset)” on page 235). 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**.
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 $\text{Run}$. The value of $\text{Run}$ is a sequence of integers from 1 to $\text{Sample\_size}$, a system variable specifying the sample size for simulation. See “Appendix A: Selecting the Sample Size” on page 416. For most operations and functions, $\text{Run}$ 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 29.
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 369), 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 374 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 mod-
els perform array abstraction conveniently when you add or modify dimensions. See “Ensuring array abstraction” on page 374 for details.

## IF a THEN b ELSE c with arrays

The **IF a THEN b ELSE c** (page 142) 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:

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

<table>
<thead>
<tr>
<th></th>
<th>-2</th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>'Negative'</td>
<td>'Negative'</td>
<td>'Zero'</td>
<td>'Positive'</td>
<td>'Positive'</td>
</tr>
</tbody>
</table>
```

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

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

<table>
<thead>
<tr>
<th></th>
<th>-2</th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>'Imaginary'</td>
<td>'Imaginary'</td>
<td>0</td>
<td>1</td>
<td>1.414</td>
</tr>
</tbody>
</table>
```

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 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 366.

### 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:

```analytica
If X >= 0 THEN Sqrt(X)

<table>
<thead>
<tr>
<th></th>
<th>-2</th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>«Null»</td>
<td>«Null»</td>
<td>0</td>
<td>1</td>
<td>1.414</td>
</tr>
</tbody>
</table>
```
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 366), 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:

```plaintext
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:

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

**IFONLY a THEN b ELSE c** 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:

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 20) and select Definition (page 104) from the Attribute menu.
5. Press the expr menu above the definition field, to see these options.

   ![Expr Menu]

   (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.)

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]

   List icon for the expr popup menu

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.
Creating an index

**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.

<table>
<thead>
<tr>
<th>List view</th>
<th>Expression view</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, 2, 3, 4, 5]</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>List view</th>
<th>Expression view</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>['Alabama', 'Alaska', 'Arizona', 'Arkansas']</td>
</tr>
<tr>
<td>Alaska</td>
<td></td>
</tr>
<tr>
<td>Arizona</td>
<td></td>
</tr>
<tr>
<td>Arkansas</td>
<td></td>
</tr>
</tbody>
</table>

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 139):

```
['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.

<table>
<thead>
<tr>
<th>List view</th>
<th>Expression view</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, 'Alabama', 2, 'Alaska']</td>
</tr>
<tr>
<td>'Alabama'</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>'Alaska'</td>
<td></td>
</tr>
</tbody>
</table>
If you attempt to mix numbers and text in a list of labels, all the values are treated as text, as shown below.

<table>
<thead>
<tr>
<th>List view</th>
<th>Expression view</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>['1', 'Alabama', '2', 'Alaska']</td>
</tr>
<tr>
<td>Alabama</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Alaska</td>
<td></td>
</tr>
</tbody>
</table>

**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 Backspace.

**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 178).

**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 173).
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, strict, dateUnit)” on page 173.

**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 173.

**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 169). 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.

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

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 169.
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 \( \text{CompareVars} \), 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 \( \text{CompareVars} \), it will become:

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

The result of \( \text{CompareVars} \) is an array of the values of the variables it contains, with a self index, also called \( \text{CompareVars} \), 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 \( \text{CompareVars} \) for \( A \) will have the same value of \( A \) repeated for each value of \( A \). See “Compare a list of variables” on page 161 for an example.

Self index

The result will contain an extra index, a self index of \( \text{CompareVars} \), 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 382 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:

\[
5 .. 1 \rightarrow [5, 4, 3, 2, 1]
\]

It is equivalent to \( \text{Sequence}(m, n) \).

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 375 on how to use this in a way that supports array abstraction.

Sequence(start, end, stepSize, strict, dateUnit)

Creates a list of numbers increasing or decreasing from \( \text{start} \) to \( \text{end} \) by increments (or decrements) of \( \text{stepSize} \), which is optional and defaults to 1. When the \( \text{strict} \) parameter is omitted or false, \( \text{stepSize} \) must be a positive number and the sequence will decrement by \( \text{stepSize} \) when \( \text{end} \) is less than \( \text{start} \), guaranteeing at least one element. When \( \text{strict} \) is specified as true, a positive \( \text{stepSize} \) increments and negative \( \text{stepSize} \) returns a decrementing sequence, possibly with zero elements if \( \text{end} \) would come before \( \text{start} \).

The optional \( \text{dateUnit} \) parameter is used when creating a sequence of dates, with increments in units of Years (\( \text{dateUnit}:'Y' \)), Months (\( \text{dateUnit}:'M' \)), Days (\( \text{dateUnit}:'D' \) or omitted), Weekdays (\( \text{dateUnit}:'WD' \)), Hours (\( \text{dateUnit}:'h' \)), minutes (\( \text{dateUnit}:'m' \)) or seconds (\( \text{dateUnit}:'s' \)).

All parameters must be deterministic scalar numbers, not arrays.
# Chapter 12: Arrays and Indexes

## Functions that create indexes

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 171.

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:

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

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

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

Unless `strict` is true:

```plaintext
Sequence(5, 1, strict:true) → []
Sequence(5, 1, -2, strict:true) → [5, 3, 1]
```

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

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

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

```plaintext
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 212 for how to concatenate two arrays.

```plaintext
Index Year1 := 2006 .. 2008
Index Years2 := 2009 .. 2010
Index YearsAll := Concat(Year1, Years2)
```

### 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.

The optional parameter `position:true` can be specified to return the positions along `d`'s index for which `d`'s values are true. You would need to use positions if your index might contain duplicate values.

The basic use of `Subset` does not allow `d` to contain more than one dimension. An extended use that can be applied to multi-dimensional array parameters is described further in "Subset(d,position,i,resultIndex)" on page 214.

#### When to use

Use `Subset()` to create a new index that is a subset of an existing index.

#### Library  Array

#### Example

```plaintext
```
Functions that create indexes

Subset(YearsAll>2007 and YearsAll<2010, position:true) → [3,4]

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
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 ▶

<table>
<thead>
<tr>
<th>VW</th>
<th>Honda</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1950</td>
<td>1800</td>
<td>2210</td>
</tr>
</tbody>
</table>

SortIndex(Maint_costs, Car_type) →
Car_type ▶

<table>
<thead>
<tr>
<th>VW</th>
<th>Honda</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Honda</td>
<td>VW</td>
<td>BMW</td>
</tr>
</tbody>
</table>

SortIndex(Maint_costs) →
SortIndex ▶

<table>
<thead>
<tr>
<th>Honda</th>
<th>VW</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Honda</td>
<td>VW</td>
<td>BMW</td>
</tr>
</tbody>
</table>

Define Sorted_cars as an index node:

INDEX Sorted_cars := SortIndex(Maint_costs)
Maint_costs[Car_type = Sorted_cars] →

<table>
<thead>
<tr>
<th>Honda</th>
<th>VW</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1800</td>
<td>1950</td>
<td>2210</td>
</tr>
</tbody>
</table>
**Unique(a, i)**

Returns a maximal subset of i such that each indicated slice of a along i is unique.

The optional parameter `position:true` returns the positions of element in i, rather than the elements themselves. Specifying `caseInsensitive:true` ignores differences in upper and lower case in text values when determining if values are 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

<table>
<thead>
<tr>
<th>LastName</th>
<th>FirstName</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smith</td>
<td>Bob</td>
</tr>
<tr>
<td>2</td>
<td>Jones</td>
<td>John</td>
</tr>
<tr>
<td>3</td>
<td>Johnson</td>
<td>Bob</td>
</tr>
<tr>
<td>4</td>
<td>Smith</td>
<td>Bob</td>
</tr>
</tbody>
</table>

`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.
Defining a variable as an edit table

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 178) 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 move 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 move 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 33) or the Sample() function (page 295).

Time is the index for dynamic simulation (page 315). 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 20)
- The Attribute panel (page 20) of the diagram

In the Attribute panel, select Definition (page 104) from the Attribute popup menu.

**The Edit Table window**

The Edit Table window looks much like the Result window table view (page 28). 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 185 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.

**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 \(\checkmark\) to accept all the changes you have made to the table. If you close a table, it also accepts the changes, unless you click \(\times\).

**Cancel**

Click \(\times\) to cancel all the changes you have made to the table since you opened it or last clicked \(\checkmark\).

**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.

Entering multi-line text

When entering multi-line text or multiple line expressions into single cells of an edit table, press \texttt{Ctrl+Enter} while editing the cell contents to insert a new line within the cell. Pressing just \texttt{Enter} when editing the contents of a cells accepts the changes for that cell and leaves the cell edit; hence, you need to hold Alt when inserting the new line. You can also select Add new line within cell from the right-mouse context menu while editing.

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 \texttt{m .. n} or \texttt{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 180 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 123.

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 or select Append Row from the Edit menu.

Append a column

Click the rightmost element of the column index to select the right column, and press the right-arrow key or select Append Column from the Edit menu.

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 \texttt{Sys\_tableCellDefault}. You can also set table-specific default values, using the \texttt{TableCellDefault} attribute. See Table Splicing on the Analytica Wiki for details.

Reordering items

Items in a list view can be reordered by selecting one or more items and dragging the selected region to a new position. There is also a Reorder command on the right-mouse context menu. After selecting Reorder, move the mouse or use the arrow keys to move the selected cells to a new position in the list.

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 177). 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 177). 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:

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

Suppose, you change:
Start_year := 2006

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

<table>
<thead>
<tr>
<th>Year</th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>200</td>
<td>300</td>
</tr>
</tbody>
</table>

Suppose, you change:
Start_year := 2006

Then by default, Revenues will change to:

<table>
<thead>
<tr>
<th>Year</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200</td>
<td>300</td>
<td>0</td>
</tr>
</tbody>
</table>
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 \( n \)th 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 162)

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

\[
(\text{Revenue} - \text{Cost})[\text{Time} = 2010]
\]

**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 $\text{Size}(i)$ 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:

```plaintext
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:

$$\text{Slice}(2000..2003, 4) → 2003$$

It also works to get the $n$th slice of a multidimensional array over an unindexed dimension, for example:

$$\text{Slice}(\text{Quarters} & ' ' & 2000..2003, 4) → \text{Array}(\text{Quarters}, ['Q1 2003', 'Q2 2003', 'Q3 2003', 'Q4 2003'])$$

### 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:

$$\text{Slice(Cost, Car_type, 1) → Mpg}$$

<table>
<thead>
<tr>
<th></th>
<th>26</th>
<th>30</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
<td>2185</td>
<td>1705</td>
<td>1585</td>
</tr>
</tbody>
</table>

Here, $n$ is an array of positions:

$$\text{Slice(Cost, Car_type, [1, 2]) → Mpg}$$

<table>
<thead>
<tr>
<th></th>
<th>26</th>
<th>30</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>VW</td>
<td>2185</td>
<td>1705</td>
<td>1585</td>
</tr>
<tr>
<td>VW</td>
<td>2810</td>
<td>2330</td>
<td>2210</td>
</tr>
</tbody>
</table>

### Preceding time slice: $x[\text{Time-1}]$

$x[\text{Time-n}]$ refers to the built-in index Time (see “The Time index” on page 316). 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 316).
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 \( \text{Size}(i) \). The first element of \( i \) is at position 1; the last element of \( i \) is at position \( \text{Size}(i) \). The position operator \( @ \) offers three ways to work with positions:

- \( @i \) → an array of integers from 1 to \( \text{Size}(i) \) indexed by \( i \).
- \( @[i=\text{x}] \) → the position of value \( \text{x} \) in index \( i \), or 0 if \( \text{x} \) is not an element of \( i \).
- \( e[@i=\text{n}] \) → the \( n^{th} \) slice of the value of expression \( e \) over index \( i \).

### Examples

<table>
<thead>
<tr>
<th>Index Car_type :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>@Car_type →</td>
</tr>
<tr>
<td>Car_type ▶</td>
</tr>
</tbody>
</table>

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

### More examples and tips

#### Index Time:

<table>
<thead>
<tr>
<th>Time := Time+2007 →</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>@Time →</td>
</tr>
<tr>
<td>Time ▶</td>
</tr>
</tbody>
</table>

| @Time=2 → 3 |
| @Time = 3 → |

| Time[@Time=3] → 2 |
| (Time+2007)[@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]

### Choice(i, n, inclAll, eval, result, separator)

Appears as a popup menu in the definition field, allowing selection of the \( n^{th} \) item from \( i \) (see “Creating a choice menu” on page 123). \( \text{Choice}() \) must appear at the topmost level of a definition or table cell. It cannot be used inside another expression. The optional \( \text{inclAll} \) parameter controls whether the "All" option (\( n=0 \)) appears on the popup (\( \text{inclAll} \) defaults to True). When using \( \text{Choice} \) in a table cell, it is a good practice to set \( \text{inclAll} \) to \( \text{false} \). When index \( i \) contains handles to objects, the optional boolean \( \text{eval} \) parameter controls whether the handle is returned (\( \text{eval:False} \)) or the result of evaluating the variable is returned (\( \text{eval:True} \)). The optional
result parameter can be an array indexed by i, which specifies the value returned and which may be different from the elements of i. You can include separators in the choice menu by specifying a value for the optional separator parameter. The elements of i matching separator will appear as non-selectable separators in the popup menu.

Examples

\[
\text{Choice}(\text{Years}, 2) \rightarrow 1986
\]

If \( n=0 \), and inclAll is true, it returns all values of i:

\[
\text{Choice}(\text{Years}, 0, 1) \rightarrow
\]

Years


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 183) 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.

Note: When you enter the cell definition, the choice control displays immediately. You may want to edit the textual expression to make a change, or to copy the textual expression for pasting into other cells. In edit mode you can toggle between the viewing the control and the textual expression using the expression/control selector. Note that the selector is not present in browse mode, or when no cells with controls are visible.
4. Copy and paste \texttt{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 containing drop-down menus in some but not all cells, with \textit{Show Expressions} selected.

5. Select \( X \), then select \textbf{Make Input} from the \textbf{Object} menu to make an input node for it. Move the input node to a good location. You will usually want an input node so the selections can be changed when the table is viewed in browse mode.

\textbf{Tip} The variable containing the edit table with menus \textit{should always} have an input node — otherwise, you won’t be able to select from the menus or edit other cells in browse mode.

\textbf{Note:} You can also insert checkbox controls into table cells using the same steps. Enter the expression \texttt{Checkbox(0)} or \texttt{Checkbox(1)} directly into a cell.

\section*{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. \texttt{Control+Page Up} and \texttt{Control+Page Down} are exceptions.

The \textit{current cell} is highlighted, or the first cell you selected in a highlighted rectangular region. In a region, the \textit{anchor cell} is the corner opposite the current cell. If you select only one cell, the \text{Anchor} and \textit{Current} are the same cell.

\textbf{Mouse operations}

- \textit{Mouse Click} Click in a cell to make it the current cell.
- \textit{Mouse Shift+Click} Select the region from the previous anchor to this cell.
- \textit{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.
- \textit{Mouse wheel} Scroll vertically without changing the selection.
- \textit{Control+mouse wheel} Scroll horizontally without changing the selection.

\textbf{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:

- \texttt{down-arrow} If you have selected the last row, add a row.
- \texttt{left-arrow} If you have selected the right column, add a column.
- \texttt{Control+i} If you have selected a row header, insert a row. If you have selected a column header, insert a column.
- \texttt{Control+k} Delete a selected row or column.
**Chapter 12 Arrays and Indexes**

**Shortcuts to navigate and edit a table**

**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).

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

- Create arrays (page 189)
- Reduce the number of dimensions in an array (page 192)
- Transform an array into another with the same dimensions (page 197)
- Interpolate values from arrays of x and y values (page 207)
- Fourier Transforms (page 208)
- Sets - collections of unique elements (page 209)
- Other array functions (page 212)
- DetemTables (page 215)
- SubTables (page 218)
- Work with matrices (page 218)
This chapter describes several classes of function and other constructs that work with arrays. If you have not already read it, we recommend that you read “Introducing indexes and arrays” on page 150 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 := 2005 .. 2009
- Index Time := 0 .. 4
- Index CarNum := 1 .. 7
- Index MaintType := ['Repair', 'Scheduled', 'Tires']

Variable Car_prices :=
Car_type ▼, Years ▲

<table>
<thead>
<tr>
<th></th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
<td>$16,000</td>
<td>$17,000</td>
<td>$18,000</td>
<td>$19,000</td>
<td>$20,000</td>
</tr>
<tr>
<td>Honda</td>
<td>$18,000</td>
<td>$19,000</td>
<td>$20,000</td>
<td>$22,000</td>
<td>$24,000</td>
</tr>
<tr>
<td>BMW</td>
<td>$25,000</td>
<td>$26,000</td>
<td>$28,000</td>
<td>$30,000</td>
<td>$32,000</td>
</tr>
</tbody>
</table>

Variable Miles :=
Car_type ▼, Years ▲

<table>
<thead>
<tr>
<th></th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
<td>8000</td>
<td>7000</td>
<td>10K</td>
<td>6000</td>
<td>9000</td>
</tr>
<tr>
<td>Honda</td>
<td>10K</td>
<td>12K</td>
<td>11K</td>
<td>14K</td>
<td>13K</td>
</tr>
<tr>
<td>BMW</td>
<td>5000</td>
<td>8000</td>
<td>8000</td>
<td>7000</td>
<td>10K</td>
</tr>
</tbody>
</table>

Variable Miles_per_gallon:=
Car_type ▲

<table>
<thead>
<tr>
<th></th>
<th>VW</th>
<th>Honda</th>
<th>MBW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32</td>
<td>34</td>
<td>18</td>
</tr>
</tbody>
</table>

Variable Rate_of_inflation :=
Years ▲

<table>
<thead>
<tr>
<th></th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1.01</td>
<td>1.02</td>
<td>1.03</td>
<td>1.04</td>
</tr>
</tbody>
</table>

Variable Cost_of_ownership :=
Car_type ▼, Time ▲

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
<td>2810</td>
<td>2951</td>
<td>3098</td>
<td>3253</td>
<td>3416</td>
</tr>
<tr>
<td>Honda</td>
<td>3535</td>
<td>3847</td>
<td>3897</td>
<td>4166</td>
<td>4365</td>
</tr>
<tr>
<td>BMW</td>
<td>3185</td>
<td>3294</td>
<td>3409</td>
<td>3529</td>
<td>3656</td>
</tr>
</tbody>
</table>

Variable NumMaintEvents :=
MaintType ▼, CarNum ▲

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repair</td>
<td>10</td>
<td>4</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Scheduled</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>
Chapter 13  More Array Functions

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 176). If you want to define a table by explicitly listing its indexes and providing expressions to generate its values or sub-arrays, you might find Array() more convenient.

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 expres-
sion returning an array which typically has 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. \texttt{Array()} is one of the few places where you actually need to worry about the order of the indexes in the array representation.

Use \texttt{Array()} to specify an array directly as an expression. \texttt{Array()} is similar to \texttt{Table() (page 191)}; 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:

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

Definition viewed as a table:

<table>
<thead>
<tr>
<th>Car_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
</tr>
<tr>
<td>32</td>
</tr>
</tbody>
</table>

**Example 2**  
If an array has multiple dimensions, then the elements are listed in nested brackets, following the structure of the array as an array of arrays (of arrays..., and so on, according to the number of dimensions).

Definition viewed as an expression:

```plaintext
Array(Car_type, Years, [[8K, 7K, 10K, 6K, 9K], [10K, 12K, 11K, 14K, 13K], [5K, 8K, 8K, 7K, 10k]])
```

Definition viewed as a table:

<table>
<thead>
<tr>
<th>Car_type, Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
</tr>
<tr>
<td>VW</td>
</tr>
<tr>
<td>Honda</td>
</tr>
<tr>
<td>BMW</td>
</tr>
</tbody>
</table>

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,listLen)” on page 214.

**Example 3**  
If an element is a scalar where an array is expected, \texttt{Array()} expands it to create an array with the scalar value repeated across a dimension.

Definition viewed as an expression:

```plaintext
Array(Car_type, Years, [[8K, 7K, 10K, 6K, 9K], [13K], [5K, 8K, 8K, 7K, 10k]])
```

Definition viewed as a table:

<table>
<thead>
<tr>
<th>Car_type, Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
</tr>
<tr>
<td>VW</td>
</tr>
<tr>
<td>Honda</td>
</tr>
<tr>
<td>BMW</td>
</tr>
</tbody>
</table>

**Example 4**  
Use \texttt{Array()} to change an index of a previously defined array.

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

Chapter 13  More Array Functions

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 189); Table() requires m numeric or text values.

A definition created as a table from the expr menu uses Table() in expression view.

Example 1

Definition viewed as an expression:
Table(Car_type) (32, 34, 18)

Definition viewed as a table:

Car_type ▶

<table>
<thead>
<tr>
<th>VW</th>
<th>Honda</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>34</td>
<td>18</td>
</tr>
</tbody>
</table>

Example 2

Definition viewed as an expression:
Table(Car_type, Years)
(8K, 7K, 10K, 6K, 9K, 10K, 12K, 11K, 14K, 13K, 5K, 8K, 8K, 7K, 10K)

Definition viewed as a table:

Car_type ▼, Years ▲

<table>
<thead>
<tr>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
<td>8000</td>
<td>7000</td>
<td>10K</td>
<td>6000</td>
</tr>
<tr>
<td>Honda</td>
<td>10K</td>
<td>12K</td>
<td>11K</td>
<td>11K</td>
</tr>
<tr>
<td>BMW</td>
<td>5000</td>
<td>8000</td>
<td>8000</td>
<td>7000</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Car_type</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IntraTable(i1, i2, ... in) (u1, u2, u3, ... um)

IntraTable generalizes the Table function to permit functional dependencies between cells. These dependencies can be denoted as intra-cell dependencies, and hence, the table as a whole as an IntraTable. An IntraTable behaves identically to an edit table in all other respects.

The presence of intra-cell dependencies produces a table that is reminiscent of a spreadsheet, with dependencies between individual cells. As such, it is advisable to avoid IntraTable if you don’t need to, since it can easily be abused and lead to inflexible and hard-to-maintain models.

To create an IntraTable, while editing a definition select Other... on the expr menu and select IntraTable in the Array Functions library.

Within cells, you can apply Slice and Subscript notation to the Self keyword to reference other cells within the table. The intra-cell dependencies must not form a directed loop, which would imply that a cell is defined indirectly in terms of itself, otherwise an error will result when the table is evaluated.

The following example contains two cells that illustrate intra-cell references, with examples of both subscript and slice notation.

<table>
<thead>
<tr>
<th>Car_type</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When evaluated, the following array results:

<table>
<thead>
<tr>
<th>Car_type</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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, \(\text{Sum}(x, i)\), \(\text{Product}(x, i)\), \(\text{Max}(x, i)\), \(\text{Min}(x, i)\), 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 181).

The function \(\text{Sum}(x, i)\) illustrates some properties of reducing functions.

<table>
<thead>
<tr>
<th>Car_type</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sum(Car_prices, Years) →
Car_type

<table>
<thead>
<tr>
<th>VW</th>
<th>Honda</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td>90K</td>
<td>103K</td>
<td>141K</td>
</tr>
</tbody>
</table>

Sum(Sum(Car_prices, Years), Car_type) → 334K

See “Example variables” on page 188 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 \(\text{Sum()}\) or any other array-reducing function.

Reducing over an unused index
If the index, \(i\), is not a dimension of \(x\), \(\text{Sum}(x, i)\) 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 \(x := 5\)
\[
\text{Sum}(x, \text{Car_type}) → 15
\]
This is because \(\text{Car_type}\) has three elements (\(3 \times 5 = 15\)). For Product:

\[
\text{Product}(x, \text{Car_type}) → 125
\]
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 \(\text{Car_type}\), we can redefine \(x\) as an edit table indexed by \(\text{Car_type}\). Any expression containing a \(\text{Sum()}\) or other reducing function on \(x\) works correctly whether it is indexed by \(\text{Car_type}\) or not.

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

When a \(\text{NaN}\) 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 \(\text{NaN}\) as well. \(\text{NaN}\) 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 \(\text{NaN}\) values in an array-reducing operation. The array-reducing functions \(\text{Sum, Product, Average, Min, and Max}\) all accept an optional parameter, \(\text{ignoreNaN}\) that can be set to \(\text{True}\). \(\text{IgnoreNaN}\) requires a named-parameter syntax, for example:

\[
\text{Max}(x, i, \text{ignoreNaN:True})
\]

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

\[
\text{Max}(x, i, \text{ignoreNonNumbers:True})
\]

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

\[
\text{Sum}(x, i, j, k)
\]
This is equivalent to:

\[
\text{Sum} (\text{Sum} (\text{Sum}(x, i, j), j), k)
\]

\(\text{Sum}(x, i)\)

Returns the sum of array \(x\) over the dimension indexed by \(i\).
Chapter 13  More Array Functions

Array-reducing functions

Library  Array

Examples  Sum(Car_prices, Years) → Car_type

<table>
<thead>
<tr>
<th></th>
<th>VW</th>
<th>Honda</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>90K</td>
<td>103K</td>
<td>141K</td>
</tr>
<tr>
<td>2006</td>
<td>103K</td>
<td>103K</td>
<td>141K</td>
</tr>
<tr>
<td>2007</td>
<td>103K</td>
<td>103K</td>
<td>141K</td>
</tr>
<tr>
<td>2008</td>
<td>103K</td>
<td>103K</td>
<td>141K</td>
</tr>
<tr>
<td>2009</td>
<td>103K</td>
<td>103K</td>
<td>141K</td>
</tr>
</tbody>
</table>

See “Example variables” on page 188 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(Car_prices, Car_type) → Years

<table>
<thead>
<tr>
<th></th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>7.2T</td>
<td>8.398T</td>
<td>10.08T</td>
<td>12.54T</td>
<td>15.36T</td>
</tr>
<tr>
<td>2006</td>
<td>8.398T</td>
<td>10.08T</td>
<td>12.54T</td>
<td>15.36T</td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td>10.08T</td>
<td>12.54T</td>
<td>15.36T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>12.54T</td>
<td>15.36T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>15.36T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Average(x, i)

Returns the mean value of all of the elements of array x, averaged over index i.

Library  Array

Examples  Average(Miles, Years) → Years

<table>
<thead>
<tr>
<th></th>
<th>VW</th>
<th>Honda</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>8000</td>
<td>12K</td>
<td>7600</td>
</tr>
<tr>
<td>2006</td>
<td>8000</td>
<td>12K</td>
<td>7600</td>
</tr>
<tr>
<td>2007</td>
<td>8000</td>
<td>12K</td>
<td>7600</td>
</tr>
<tr>
<td>2008</td>
<td>8000</td>
<td>12K</td>
<td>7600</td>
</tr>
<tr>
<td>2009</td>
<td>8000</td>
<td>12K</td>
<td>7600</td>
</tr>
</tbody>
</table>

Max(x, i)

Returns the highest valued element of x along index i.

Library  Array

Examples  Max(Miles, Years) → Car_type

<table>
<thead>
<tr>
<th></th>
<th>VW</th>
<th>Honda</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>10K</td>
<td>14K</td>
<td>10K</td>
</tr>
<tr>
<td>2006</td>
<td>10K</td>
<td>14K</td>
<td>10K</td>
</tr>
<tr>
<td>2007</td>
<td>10K</td>
<td>14K</td>
<td>10K</td>
</tr>
<tr>
<td>2008</td>
<td>10K</td>
<td>14K</td>
<td>10K</td>
</tr>
<tr>
<td>2009</td>
<td>10K</td>
<td>14K</td>
<td>10K</td>
</tr>
</tbody>
</table>

To obtain the maximum of two numbers, first turn them into an array:

Max([10, 5]) → 10

See “Example variables” on page 188 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(Miles, Years) → Car_type

<table>
<thead>
<tr>
<th></th>
<th>VW</th>
<th>Honda</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>6000</td>
<td>10K</td>
<td>5000</td>
</tr>
<tr>
<td>2006</td>
<td>6000</td>
<td>10K</td>
<td>5000</td>
</tr>
<tr>
<td>2007</td>
<td>6000</td>
<td>10K</td>
<td>5000</td>
</tr>
<tr>
<td>2008</td>
<td>6000</td>
<td>10K</td>
<td>5000</td>
</tr>
<tr>
<td>2009</td>
<td>6000</td>
<td>10K</td>
<td>5000</td>
</tr>
</tbody>
</table>

To obtain the minimum of two numbers, first turn them into an array:
Chapter 13  More Array Functions

Array-reducing functions

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(Miles, Car_type) →

<table>
<thead>
<tr>
<th>Years</th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Honda</td>
<td>Honda</td>
<td>Honda</td>
<td>Honda</td>
<td>Honda</td>
</tr>
</tbody>
</table>

**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) →

<table>
<thead>
<tr>
<th>Years</th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BMW</td>
<td>VW</td>
<td>BMW</td>
<td>VW</td>
<td>VW</td>
</tr>
</tbody>
</table>

**CondMin(x, cond, i)**  
**CondMax(x, cond, i)**

Conditional Min and Max. **CondMin()** returns the smallest, and **CondMax()** returns the largest values along a given index, $i$, that satisfies condition **cond**. When **cond** is never satisfied, **CondMin()** returns INF, **CondMax()** returns -INF.

**Library**  none

**Examples**  CondMin(Cost_of_ownership, Time>=2, Time) →

<table>
<thead>
<tr>
<th>Car_type</th>
<th>VW</th>
<th>Honda</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3098</td>
<td>3897</td>
<td>3409</td>
</tr>
</tbody>
</table>

**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**  Array

**Examples**  Subindex(Car_prices, 18K, Car_type) →

<table>
<thead>
<tr>
<th>Years</th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Honda</td>
<td>«null»</td>
<td>VW</td>
<td>«null»</td>
<td>«null»</td>
</tr>
</tbody>
</table>

Subindex(Car_prices, 12K, Years) →

<table>
<thead>
<tr>
<th>Car_type</th>
<th>VW</th>
<th>Honda</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2007</td>
<td>2005</td>
<td>«null»</td>
</tr>
</tbody>
</table>

If $u$ is an array of values, an array of index values is returned.

Subindex(Car_prices, [12K, 21K], Car_type) →
Subindex ▼, Years ▶

<table>
<thead>
<tr>
<th></th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td>18K</td>
<td>Honda</td>
<td>«null»</td>
<td>VW</td>
<td>«null»</td>
<td>«null»</td>
</tr>
<tr>
<td>19K</td>
<td>«null»</td>
<td>Honda</td>
<td>«null»</td>
<td>VW</td>
<td>«null»</td>
</tr>
</tbody>
</table>

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

Example

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.
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 183) returns the same result as \( \text{PositionInIndex}(x,i) \):

\[
\begin{align*}
\text{PositionInIndex}('B',I) & \to 2 \\
@[I = 'B'] & \to 2 \\
\text{PositionInIndex}('D',I) & \to 0 \\
@[I = 'D'] & \to 0
\end{align*}
\]

More examples and tips  When the array is multidimensional:

Taking the same example from above:

\[
\text{PositionInIndex}(\text{Car_prices}, 18K, \text{Car_type}) 
\]

```
<table>
<thead>
<tr>
<th></th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td>Years</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

\( \text{Area}(y, x, x1, x2,i) \)

Returns the area (sum of trapezoids) under the piecewise-linear curve denoted by the points \((x_i, y_i)\), landing in the region \( x_1 \leq x \leq x_2 \). The arrays \( x \) and \( y \) must share the common index \( i \), or when either \( x \) or \( y \) is itself an index, \( i \) can be safely omitted. \( x1 \) and \( x2 \) are optional; if they are not specified, the area is calculated across all values of \( x \).

If \( x1 \) or \( x2 \) fall outside the range of values in \( i \), the first value (for \( x1 \)) or last value (for \( x1 \)) are used. \( \text{Area()} \) computes the total integral across \( x \), returning a value with one less dimension than \( y \). Compare \( \text{Area()} \) to \( \text{Integrate()} \) (page 200).

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 \( \text{Cumulate}(x, i) \) illustrates some properties of transforming functions.

Example  \( \text{Cumulate}(\text{Car_prices},\text{Years}) \) →

```
<table>
<thead>
<tr>
<th></th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
<td>16K</td>
<td>33K</td>
<td>51K</td>
<td>70K</td>
<td>90K</td>
</tr>
<tr>
<td>Honda</td>
<td>18K</td>
<td>37K</td>
<td>57K</td>
<td>79K</td>
<td>103K</td>
</tr>
<tr>
<td>BMW</td>
<td>25K</td>
<td>51K</td>
<td>79K</td>
<td>109K</td>
<td>141K</td>
</tr>
</tbody>
</table>
```

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.

\( \text{Cumulate}(x, i, \text{passNull}, \text{reset}) \)

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 \( x \).

\( \text{Cumulate}(1,i) \) is equivalent to \( @i \), where each numbers the elements of an index.
The optional passNull parameter controls now null values in \( x \) are passed through to the result. If passNull is false or omitted, then null values in \( x \) are ignored and do not effect the cumulation. Leading null values will be passed through, but after a numeric value is encountered, null values in \( x \) will cumulate the same as zero.

The optional reset parameter, an array of boolean values along \( i \), can be used to indicate points along \( i \) where you want to restart the cumulation. For example, if you want to restart the cumulation following a state change, reset can be set to true each time a new state is entered.

### Library

**Array**

### Example

Cumulate(Cost_of_ownership, Time) →

\[
\begin{array}{cccccc}
 & 0 & 1 & 2 & 3 & 4 \\
VW & 2810 & 5761 & 8859 & 12.11K & 15.53K \\
Honda & 3535 & 7382 & 11.28K & 15.45K & 19.81K \\
BMW & 3185 & 6479 & 9888 & 13.42K & 17.07K \\
\end{array}
\]

Cumulate(Cost_of_ownership,Car_type, reset:Time=2) →

\[
\begin{array}{cccccc}
 & 0 & 1 & 2 & 3 & 4 \\
VW & 2810 & 5761 & 3098 & 6351 & 9767 \\
Honda & 3535 & 7382 & 3897 & 8063 & 12.43K \\
BMW & 3185 & 6479 & 3409 & 6938 & 10.59K \\
\end{array}
\]

See “Example variables” on page 188 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_of_ownership, Time) →

\[
\begin{array}{cccc}
 & 0 & 1 & 2 \\
VW & 2810 & 141 & 147 \\
Honda & 3535 & 312 & 50 \\
BMW & 3185 & 109 & 115 \\
\end{array}
\]

Uncumulate(Cost_of_ownership, Time,0) →

\[
\begin{array}{cccc}
 & 0 & 1 & 2 \\
VW & 0 & 141 & 147 \\
Honda & 0 & 312 & 50 \\
BMW & 0 & 109 & 115 \\
\end{array}
\]

See “Example variables” on page 188 for example array variables used here and below.
CumProduct(x, i, passNull, reset)

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.

For a description of the optional parameters, passNull and reset, see function “Cumulate(x, i, passNull, reset)” on page 197

Example

Cumproduct(Rate_of_inflation, Years) →

<table>
<thead>
<tr>
<th></th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td>Years</td>
<td>1</td>
<td>1.01</td>
<td>1.03</td>
<td>1.061</td>
<td>1.104</td>
</tr>
</tbody>
</table>

Rank(x, i, type, keyIndex, descending, caselnsensitive, passNaNs, passNulls)

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.

Rank(x,i,Type:Null) assigns a unique rank to every element (the numbers 1 thru N) in which tied elements may have different ranks.

A multi-key rank can be processed by indexing each key with a new index, and specifying this index for the optional keyIndex parameter. In a multi-key rank, x[@KeyIndex=1] determines the rank order, except that ties are then resolved using x[@KeyIndex=2], any ties there are resolved using x[@KeyIndex=3], and so on.

Rank(x,i,descending:true) assigns the largest value a rank 1, the second largest a rank 2, and so on. When x contains textual values, the optional boolean parameter caselnsensitive:true ignores upper-lower case differences during the comparisons. The parameters descending and caselnsensitive may also be indexed by they keyIndex when they vary by key.

By default, Rank assigns an arbitrary ranking to NaN or Null values. Alternatively, you can pass these through to the result as NaN or Null using Rank(x,i,passNaNs:true, passNulls:true).

Example

Basic example:

Rank(Years) →

<table>
<thead>
<tr>
<th></th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td>Years</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Rank(Car_prices, Car_type) →

Car_type ▼, Years ►

<table>
<thead>
<tr>
<th></th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Honda</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>BMW</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Optional Type parameter example:

Index RankType := [-1,0,1, Null]

Rank(NumRepairs,CarNum,Type:RankType) →
Chapter 13  More Array Functions

Transforming functions

**Rank_type**, CarNum

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>3.5</td>
<td>6</td>
<td>3.5</td>
<td>3.5</td>
<td>1</td>
<td>3.5</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Null</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Lowest rank for duplicates, 2 (default)

Mid rank for duplicates, 3.5

Upper rank for duplicates, 5

Unique rank for duplicates

Multi-key example:

\[
\text{Rank(NumMaintEvents, CarNum, KeyIndex:MaintType)} \rightarrow \text{CarNum}
\]

See “Example variables” on page 188 for example array variables used here and below.

**Sort(x, i, keyIndex, descending, caseInsensitive)**

Sort(x, i) returns the elements of x, reordered along index i in sorted order. The equivalent can be accomplished using the SortIndex function as:

\[x[i=\text{SortIndex}(x, i)]\]

To perform a multi-key sort, in which the first key determines the sort order unless there are ties, in which the second key breaks the ties, the third key breaks any remaining ties, etc., collect the key criteria along an index K and specify the optional keyIndex parameter, e.g.:

\[\text{Sort(Array(K, [key1, key2, key3]), i, keyIndex:K)}\]

The data is sorted in ascending order (from smallest to largest), unless you specify the optional parameter descending: true, which then reorders from largest to smallest. The optional parameter caseInsensitive: true ignores lower/upper case in textual comparisons. Either of these may also be indexed by the keyIndex when the order or case-insensitivity varies by key.

Multi-key example:

\[\text{Sort(NumMaintEvents, CarNum, KeyIndex: MaintType, descending: true)} \rightarrow \text{MaintType, CarNum}\]

**Integrate(y, x, i)**

Returns the integral of the piecewise-linear curve denoted by the points (x, y), computed by applying the trapezoidal rule of integration to the arrays of points x, y over index i. Integrate() computes the cumulative integral across i, returning a value with the same number of dimensions as y. Compare Integrate() to Area() (page 194) and Cumulate() (page 197).

When x is itself an index, then i can be omitted and y is an array indexed by x. Likewise, if y is an index, i can be omitted and x is indexed by y.

Library  Array

Example  Integrate(Cost_of_ownership, Time) \rightarrow Car_Type, Time

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
<td>0</td>
<td>2881</td>
<td>5905</td>
<td>9081</td>
<td>12.42K</td>
</tr>
<tr>
<td>Honda</td>
<td>0</td>
<td>3691</td>
<td>7563</td>
<td>11.59K</td>
<td>15.86K</td>
</tr>
<tr>
<td>BMW</td>
<td>0</td>
<td>3240</td>
<td>6591</td>
<td>10.06K</td>
<td>13.65K</td>
</tr>
</tbody>
</table>
Chapter 13  More Array Functions

**Tip** There are subtle relationships among Area(), Integrate(), Sum() and Cumulate(): When Area() operates over the entire index it returns the last value of the Integrate() series. Both of these functions are based on trapezoidal integration, meaning that they integrate the averages of each adjacent pair over the boundaries of the index. Similarly, Sum() returns the last value of Cumulate(). But in this case the operation is a simple addition, not a trapezoidal integration.

**Normalize(y, x, i)**

Returns a re-scaled version of array \( y \), such that the area under the piecewise-linear curve denoted by the points \((x_i, y_i)\) is re-scaled to be one. Normalize is equivalent to

\[
\frac{y}{\text{Area}(y,x,,,i)}
\]

The arrays \( x \) and \( y \) must both contain numeric values, and should share \( i \) as a common index. When either \( x \) or \( y \) is itself the shared index, the parameter \( i \) can be omitted, but it is a good practice to include it anyway.

**Tip** Normalize() does not force the values along index \( i \) to sum to 1, nor does it force the sum of squares to be 1. To do these operations divide \( y \) by \( \text{Sum}(y, i) \) or by \( \sqrt{\text{Sum}(y^2,i)} \). We recommend always including the third parameter, \( i \), when using Normalize(), even when the shared index is passed for \( x \), since this helps to avoid errors or confusion with these other senses of normalization.

**Library** Array

**Example**

\[
\text{Normalize(Cost\_of\_Ownership, Time, Time)} \rightarrow \text{Car\_type, Time}
\]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
<td>0.2263</td>
<td>0.2377</td>
<td>0.2495</td>
<td>0.2620</td>
<td>0.2752</td>
</tr>
<tr>
<td>Honda</td>
<td>0.2229</td>
<td>0.2426</td>
<td>0.2457</td>
<td>0.2627</td>
<td>0.2752</td>
</tr>
<tr>
<td>BMW</td>
<td>0.2333</td>
<td>0.2413</td>
<td>0.2497</td>
<td>0.2585</td>
<td>0.2678</td>
</tr>
</tbody>
</table>

See “Example variables” on page 188 for example array variables used here and below.

**Dispatch(demand, capacity, resource, active, minimum, increment)**

Allocates capacity from a set of resources to satisfy a demand, consuming capacity from the first resource in the order they appear in the resource index. Capacity is an array indexed by the resource index. The result is an array indexed by resource indicating how much of each resource’s capacity is dispatched to satisfy the total demand.

The optional active flag parameter, when specified, should be indexed by resource and set to true when the corresponding resource can be used to fill the demand, false if it cannot be applied to fill the demand.

Optional minimum and increment parameters, which may optionally be indexed by resource, specify that the indicated resource must be dispatched in fixed increments, with at least minimum capacity dispatched, in integral increments of increment. You can pass the same value to capacity and increment to indicate that each resource must be dispatched fully or not at all. A null value appearing in either minimum or increment indicates that no increment constraint applies to the corresponding resource.

**Library** Array

**Example** A set of projects have been pre-ordered from highest to lowest return on investment (ROI). Each project must receive at least half its funding or none at all. There is a budget of $600K to allocate.

\[
\text{Full\_Cost} = \]

Analytica User Guide 201
Converting between multiD and relational tables

The MDArrayToTable() function “flattens” a multi-dimensional array into a two-dimensional relational table. When a simple relational transformation is desired, the table will have one row for each array element and only one column (the right-most column) containing those element values. Alternatively MDArrayToTable() can produce a fact table in which values occupy multiple columns divided over a specified index, referred to as the Value Index. Both of these methods are described in separate sections below. The third section applies to both kinds of tables and describes partial transformations in which the MDArrayToTable() function will only operate on a subset of indexes, leaving the rest in array form.

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, before transferring multi-dimensional data between these applications and Analytica, it might be necessary to convert between multi-dimensional arrays and two-dimensional tables.

MDArrayToTable(A, I, L)
(pure relational transformation)

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. Each column along L, except the far right column, represents a coordinate index from the array. Index columns are populated with the coordinate values for the element. The far right column contains the actual value of the element. Both I and L are optional, if either or both is omitted, a local index of the appropriate length is automatically created. The local index L contains handles to the array’s indexes for the coordinate columns.

If you provide the index I with the number of elements equal to size(A), the resulting table will contain all array elements. If the number of elements in I is equal to the number of non-empty elements of A, the resulting table will contain only the non-empty elements of the array. If the number of elements in I is anything else, an error will occur. The optional parameters omitNull and omitZero control what is considered to be empty. When you omit I, the local index is sized for the non-empty elements, or you can obtain all elements by setting both omitNull and omitZero to false.
Chapter 13 More Array Functions

Converting between multiD and relational tables

Row := sequence(1, size(Array_3x3x3))
Col := ['Number', 'Letter', 'Hue', 'Value']

All but the last column elements specify indexes from the array
The last column element contains the name heading of the value column. It does not specify an index

MDArrayToTable(Array_3x3x3, Row, Col) →

<table>
<thead>
<tr>
<th>Row</th>
<th>Col</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The resulting table populates Index columns with the corresponding index labels for the array element. Set the optional parameter positional to true to return the index positions rather than index labels:

MDArrayToTable(Array_3x3x3, Row, Col, positional: true) →

<table>
<thead>
<tr>
<th>Row</th>
<th>Col</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Column headings can be customized using a one-dimensional variable to associate headings and indexes:

Index Headings := ['X', 'Y', 'Z', 'Values']
Variable Index2Heading := Table(Headings)
('Number', 'Letter', 'Hue', 'Anything')
MDArrayToTable(Array_3x3x3, Row, Index2Heading, positional: true) →

<table>
<thead>
<tr>
<th>Row</th>
<th>Col</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**MDArrayToTable(A, I, L, Valuelndex)**  
(fact table transformation)

The `MDArrayToTable()` function creates a fact table whenever the optional `Valuelndex` parameter is used. This format allows you to have multiple columns of values divided along one of the original array indexes, the designated `Valuelndex`. The headings of each value column are typically the elements of the value index. In this type of transformation, the value index is not considered to be a positional coordinate of the original array. Instead, the array is interpreted as having a reduced coordinate system spanned only by the remaining indexes, with each coordinate location containing multiple values.

**Example**  
Starting with the same array from the example above, you can specify `Hue` as the value index. The array would be considered to have only two remaining coordinate dimensions: `Number` and `Letter`. This changes the effective size of the array, and therefore the length of the row index:

```
Row := 1..Sum(1,Number,Letter)  or equivalently:  Row := 1..9
```

The `L` parameter includes index columns as before, but the value column now contains the elements of the value index:

```
Col := Concat(['Number','Letter'],Hue)  or equivalently:
Col := ['Number','Letter','Red','Green','Blue']
```

If you omit the `L` parameter, `MDArrayToTable` will create a local index for you named `L`, using the elements of `Hue` as the last there elements of the local index.

Include `Hue` as the `Valuelndex` parameter:

```
MDArrayToTable(Array_3x3x3,Row,Col,Hue)  →
```

<table>
<thead>
<tr>
<th>Number</th>
<th>Letter</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>45</td>
<td>34</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>13</td>
<td>11</td>
<td>48</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>12</td>
<td>84</td>
<td>57</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>19</td>
<td>25</td>
<td>65</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>B</td>
<td>81</td>
<td>19</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>C</td>
<td>47</td>
<td>53</td>
<td>23</td>
</tr>
</tbody>
</table>

**MDArrayToTable()**  
(partial transformation)

In both examples above, the `MDArrayToTable()` function completely "flattened" the entire array into a two-dimensional table. But every array can be regarded as a stack of smaller sub-arrays, each having one less dimension than the whole. If `MDArrayToTable()` operates on only one of these sub-arrays, the result will also be a flat two-dimensional table. If it operates on each of the sub-arrays individually, the result will be a stack of tables which can no longer be described as completely "flat". Extending this idea, it is possible to imagine a plane of tables or a cube of tables if the original array starts off with enough dimensions. This essentially describes the idea of a partial transformation. The `MDArrayToTable()` function is not obliged to operate on all available array indexes. In fact there are often advantages to leaving some dimensions intact since they can still be analyzed using Analytica's array abstraction features.

Specifying which dimensions are to be preserved in an `MDArrayToTable()` transformation involves a specialized coding technique described by some advanced users in New Jersey as, “fugetaboutit!” If you simply forget about certain indexes and remove all references to them in the `MDArrayToTable()` function and its parameters, the result will be an array of tables with dimensions along the omitted indexes. This is true whether or not the `Valuelndex` parameter is used. In
Chapter 13  More Array Functions

Converting between multiD and relational tables

any case the length of the \( I \) parameter must be equal to the number of elements (or mutually non-empty elements) in each of the flattened sub-arrays.

**MDTable(T, rows, cols, vars, conglomerationFn, defaultValue, ValueColumn)**

Returns a multi-dimensional array from a two-dimensional table of values. If \( n \) is the total number of columns and \( m \) is the number of value columns in the table \( T \), the following structure is assumed: The first \( n-m \) columns of \( T \) specify coordinates, and the right-most \( m \) columns contain data values. When there is only one data value in each row (a single value column) the table is described as a pure relational table. When there are multiple value columns it is described as a fact table.

**rows** and **cols** specify the vertical and horizontal indexes of the two-dimensional table. The length of the **rows** index is equal to the number of data records in the table. The length of **cols** is equal to the total number of columns.

The optional parameter **ValueColumn** specifies the index over which multiple value columns are divided in a fact table. If **ValueColumn** is omitted, the table is assumed in pure relational format, having a single value column.

The parameter **vars** is a list of index identifiers specifying the coordinate dimensions of the final array. It is optional if the table is in pure relational format (single value column). If **vars** is omitted, the dimensions of the final result are specified by the first \( n-1 \) elements of **cols**, where \( n \) is the number of elements in **cols**.

If a fact table (multiple value columns) is being transformed then you must use **vars** to specify the coordinate indexes of the array. The elements of **vars** must correspond to the coordinate columns of the table or an error will result. The number of elements in **vars** should be equal to \( n-m \) where \( n \) is the number of elements in **cols** and \( m \) is the number of value columns. Note that the list of coordinate indexes does not include the value index.

It is possible that two or more rows of \( T \) specify identical coordinates. In this case, a conglomeration function is used to combine the values for the given cell. The **conglomerationFn** 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 **defaultValue**, or if the **defaultValue** parameter is omitted, the cell value is set to undefined. Undefined values can be detected using the IsUndef() function.

**Library**  Array

**Example**  Starting with the fact table produced in the previous section:

```
<table>
<thead>
<tr>
<th></th>
<th>Number</th>
<th>Letter</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
<td>45</td>
<td>34</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>B</td>
<td>13</td>
<td>11</td>
<td>48</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>C</td>
<td>12</td>
<td>84</td>
<td>57</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>A</td>
<td>19</td>
<td>25</td>
<td>65</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>B</td>
<td>81</td>
<td>19</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>C</td>
<td>47</td>
<td>53</td>
<td>23</td>
</tr>
</tbody>
</table>
```

Index **rows** := 1..9
Index **cols** := ['Number','Letter','Red','Green','Blue']
Index **Hue** := ['Red','Green','Blue']
MDTable(T,rows,cols,['Number','Letter'],ValueColumn:Hue)
Example  Modifying the table by changing the (1,C) coordinate to (1,B), it now contains a missing value at (1,C) and redundant values at (1,B):

```
 Rows ▼, Cols ▼
```

<table>
<thead>
<tr>
<th>Rows</th>
<th>Number</th>
<th>Letter</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
<td>45</td>
<td>34</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>B</td>
<td>13</td>
<td>11</td>
<td>48</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>B</td>
<td>12</td>
<td>84</td>
<td>57</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>A</td>
<td>19</td>
<td>25</td>
<td>65</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>B</td>
<td>81</td>
<td>19</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>C</td>
<td>47</td>
<td>53</td>
<td>23</td>
</tr>
</tbody>
</table>

```

MDTable(T,rows,cols,["Number","Letter"],'average','N/A',Hue) →
```

<table>
<thead>
<tr>
<th>Number ▼, Letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>

Hue = “red”
```

```
<table>
<thead>
<tr>
<th>Number ▼, Letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>

Hue = “green”
```

```
<table>
<thead>
<tr>
<th>Number ▼, Letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>

Hue = “blue”
```

**MDTable() (partial transformation)**

If the input is a partially flattened array of tables, no special action is necessary. The pre-existing dimensions will automatically be rolled up into the final array without any reference being made to them. This is consistent with the general principles of array abstraction in Analytica.
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:**
  
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
</table>

- **Index_b:**
  
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
</table>

- **Array_a:**
  
<table>
<thead>
<tr>
<th>Index_a ▼, Index_b ▲</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

**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** Array

**Examples**

To see the values in Car_prices corresponding to $Years >= 2007.5$:

- **Stepinterp(Year, Car_prices, 2007.5, Years)**

<table>
<thead>
<tr>
<th>VW</th>
<th>Honda</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td>19K</td>
<td>22K</td>
<td>30K</td>
</tr>
</tbody>
</table>
Here \( v \) is a list of two values:

\[
\text{Stepinterp}(\text{Years, Car\_prices, [2007,2008], Years}) \rightarrow
\]

<table>
<thead>
<tr>
<th>VW</th>
<th>Honda</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007</td>
<td>18K</td>
<td>20K</td>
</tr>
<tr>
<td>2008</td>
<td>19K</td>
<td>22K</td>
</tr>
</tbody>
</table>

**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 \( v \), \( \text{Linearinterp}() \) 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** Array

**Example**

\[
\text{Linearinterp(Index\_b, Array\_a, 1.5, Index\_b}) \rightarrow
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-2.5</td>
<td>2.5</td>
</tr>
</tbody>
</table>

**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 \( v \), \( \text{Cubicinterp}() \) 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** Array

**Example**

\[
\text{Cubicinterp(Index\_b, Array\_a, 1.5, Index\_b}) \rightarrow
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6875</td>
<td>-2.875</td>
<td>2.219</td>
</tr>
</tbody>
</table>

**Fourier Transform**

The Fourier and inverse Fourier transforms convert a time-series into a power spectrum and vice-versa. These are well-known transformations that are employed for many applications including finding and characterizing periodicities in time-series analysis and regression; fast convolution and de-convolution; transfer function modeling in systems analysis; solving systems of differential equations; Bayesian analysis using characteristic functions; and signal filtering.

The discrete Fourier transform involves a time domain (corresponding to an index) and a frequency domain (corresponding to a frequency index). The time points are equally spaced at intervals of \( \Delta t \), and the frequency points are equally spaced at \( \Delta F \). Both index have \( n \) points. The intervals spacings are related as

\[
\Delta F = \frac{1}{n\Delta t}
\]

The quantity \( 1/(\Delta t) \) is called the sampling frequency, which should be at least twice the smallest frequency that is present in the underlying continuous signal. See FFT on the Analytica Wiki for additional details.
FFT(x,t,freq)

Computes the Discrete Fourier Transform (DFT) of the time-series \( x \), indexed by \( t \), and returns the discrete frequency spectrum as a complex array indexed by \( \text{freq} \). It performs this calculation using the Fast Fourier Transform (FFT) algorithm. The indexes \( t \) and \( \text{freq} \) must have the same length. This is a complex-valued function — even if \( x \) is real-valued, the result in general will be an array of complex numbers. You can apply the \( \text{Abs()} \) function to the result to obtain the magnitude of the frequency component if you are not interested in the phase.

The discrete Fourier transform is defined as

\[
H_k = \sum_{t=0}^{n-1} x_t e^{2j\pi tk/n}
\]

The computation is fastest when the length of the indexes is a power of 2. It is also reasonably efficient when all the prime factors of the length of the index is small. So, for example, it is fairly efficient when the indexes are a power of 10, since the length factors to powers of 2 and 5, both of which are still small. The computation time increases approximately quadratically with the square of the largest factor.

Library  Advanced math
Examples  For usage examples, please see FFT on the Analytica Wiki.

FFTInv(x,freq,t)

Computes the inverse Discrete Fourier Transform (DFT) using the Fast Fourier Transform (FFT) algorithm. The frequency spectrum, \( x \), is indexed by \( \text{freq} \), and typically consists of complex numbers encoding both the magnitude and phase of each frequency. Returns the time-series with the indicated spectrum, indexed by \( t \). The indexes \( \text{freq} \) and \( t \) must have the same length.

The inverse DFT is defined as

\[
h_t = \sum_{k=0}^{n-1} H_k e^{-2j\pi tk/n}
\]

The efficiency of FFTInv is the same as for FFT(), maximally efficient when the length of the indexes is a power or 2.

Library  Advanced math

Sets — collections of unique elements

In mathematics, a set is a collection of unique elements. A set itself may be treated as a self-contained entity, such that you might represent an array of sets. In such applications, you do not wish for the elements of the set to be treated as an array dimension, or to interact with array abstraction; instead, you want the set to be treated as a single atomic entity, to which operations such as set intersection and set union might be applied. Thus, a simple list representation would not be suitable, since the list itself would act as an implicit dimension, and hence would interact with array abstraction.

Representation of a set

A natural way to represent a set in Analytica is as a reference to a list of elements. References operations are covered in a more general context in Chapter 22 on page 378, but in the context of sets, you can simple view the reference operator, \( \L \), as an operator that takes a list of elements, \( L \), and returns a set, and the dereference operator, \( \#S \), as an operator returns a list of elements from a set.
### Literal sets

The following syntax is used to write an explicit set of literal elements:

\[
\{'a', 'b', 'c', 'd'\}
\]

You cannot omit the round parentheses, i.e., \{\{'a', 'b', 'c', 'd'\}, because square brackets following a reference operator are used to specify the indexes consumed by the reference.

### Creating sets from an array

Given a 2-D array, \(A\), indexed by \(I\) and \(J\), the expression \([I]A\) returns a 1-D array of sets indexed by \(J\). For each element of \(J\), the vector indexed by \(I\) becomes a set. At this point, duplicate elements have not been removed, so it isn’t a strict set yet; however, it is now ready to be treated as a set by the set functions described in this section.

#### Array_a:

<table>
<thead>
<tr>
<th>Index_a</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>7</td>
<td>-3</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>-4</td>
<td>-1</td>
<td>6</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td>0</td>
<td>-2</td>
</tr>
</tbody>
</table>

\([Index_a]Array_a \to \]

<table>
<thead>
<tr>
<th>Index_b</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>([-7,-4,5])</td>
<td>([-3,-1,0])</td>
<td>([-1,6,-2])</td>
<td></td>
</tr>
</tbody>
</table>

#### Display of sets

In a result table, each set displays as «ref». Double clicking on «ref» displays the elements of the set in a new result window.

#### Null values in sets

Set functions, like most other array functions, ignore null values by default. This makes it possible to pad an array with null values to hold the elements of a set, when the number of elements is less than the length of the array. However, it also means that set cannot contain the null value by default. When you want to include null as an actual element, and not just as a value to be ignored, the various set functions provide an optional boolean parameter to indicate this intent.

### SetContains(set,element)

Returns 1 (true) when \(element\) is in \(set\).

SetContains( \(\{'a','b','c','d'\}\), 5 ) \(\to\) 0
SetContains( \(\text{Sequence}(1,100,7)\), 85 ) \(\to\) 1

### SetDifference(originalSet, remove\(_1\), remove\(_2\), ..., resultIndex, keepNull)

Returns the set that results when the elements in the sets \(remove\(_1\), remove\(_2\), ...\), as well as any duplicate elements, are removed from \(originalSet\). When \(resultIndex\) is unspecified or false, the result is a reference to a list of elements. When an index is specified in the \(resultIndex\) parameter, the result is a 1-D array indexed by the \(resultIndex\). When \(keepNull\) is unspecified or null, then any null value in \(originalSet\) are ignored (and won’t be in the result). When \(keepNull\) is true, null is treated as a legitimate element.

Var \(S := \{0..10\}\);
Var \(S2 := \text{Sequence}(0,10,2)\);
Var \(S3 := \text{Sequence}(0,10,3)\);
#SetDifference(S,S2,S3) \(\to\) [1,5,7]

Index I := 1..5;
SetDifference(S,S2,S3,resultIndex:I) \(\to\)

<table>
<thead>
<tr>
<th>.I</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{null})</td>
<td>(\text{null})</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Set difference can be used to remove duplicates from a list:

```
Var L := [Null,'a','b','c','d','b','c','d'];
#SetDifference(L) → ['a','b','c','d']

#SetDifference(L,keepNull:true) → [null,'a','b','c','d']
```

### SetIntersection(sets,i,resultIndex,keepNull)

Returns the set intersection of `sets`. The `sets` parameter should be an array of sets indexed by `i`, or `i` can be omitted and `sets` specified as a list of sets. When `resultIndex` is omitted, the result is a set (a reference to a list) containing the elements common to all `sets`. Null values are ignored (and not included in the result) unless `keepNull` is specified as true, in which case null is treated like any other element.

```
Var S1 := ['a','b','c',null,'d'];
Var S2 := ['b','c',null,'e'];
SetIntersection([S1,S2]) → ['b','c']
SetIntersection([S1,S2],keepNull:true) → [null,'b','c']

#SetIntersection([('a'..'p'),('k'..'z')]) → ['k','l','m','n','o','p']
```

The following example finds all numbers under 10,000 divisible by the first 5 prime numbers:

```
Index n:= [2,3,5,7,11];
Var sets := (for j:=n do \Sequence(j,10K,j));
#SetIntersection(sets,n) → [2310,4620,6930,9240]
```

### SetsAreEqual(sets,i,ignoreNull)

Tests whether all `sets` in the parameter `sets` have the same elements. The parameter `sets` should be an array indexed by index `i`, or index `i` can be omitted and `sets` can be a list of sets. Null values are ignored unless `ignoreNull` is specified as false. The presence of duplicates does not impact equality determination.

```
Var L1 := ['a','b','c',null];
Var L2 := ['b','c','a'];
Var L3 := ['c','b','a','b'];
SetsAreEqual([L1,L2,L3]) → 1
SetsAreEqual([L1,L2,L3],ignoreNull:false) → 0
```

### SetUnion(sets,i,resultIndex,keepNull)

Returns a collection of all elements appearing in any set appearing in the parameter `sets`. The parameter `sets` is an array of sets (references to lists or to 1-D arrays) which is indexed by `i`, or it may be a list of sets when `i` is omitted. When `resultIndex` is omitted, the result is a reference to a list of elements. When an index is provided to the `resultIndex` parameter, the result is a 1-D array indexed by `resultIndex`. Null values are ignored (and not included in the result) unless `keepNull` is specified as true.

```
#SetUnion(\[['a'..'d'),\('c'..'f')\]) → ['a','b','c','d','e','f']

Index m := Sequence(1-Jan-2011,1-May-2011,dateUnit:'M');
index d := [0,14];
#SetUnion(SetUnion([d](m+d),d),m) →
```
Other array functions

Aggregate(x,map,i,targetIndex)

Converts from an array x indexed by fine-grain index i, to an array indexed by the coarser-grained index targetIndex. Map is an array indexed by i, specifying for each element if i the value of targetIndex that the i-value maps to. An optional parameter, positional:true, can be specified if map contains the target index position rather than the target index value.

Aggregate is used when many index elements in i correspond to the same targetIndex element. By default, the values mapping to the same target position are aggregated by summing them. An optional parameter, type, can be used to specify alternative aggregation methods. Common aggregation methods include: “Sum”, “Max”, “Min”, “Average”, “Median”, but in general any built-in or user-defined function able to accept two parameters, an array and index, such as with a declaration: (A : Array[I] ; I : Index ), can be named.

An optional parameter, defaultValue, specifies the value to use when no value maps to a given target position.

Library Array

Example To use Aggregate, you need to define the many-to-one map, here from Time to Period:

```
Index Period := ['Early', 'Middle', 'Late']
Variable Time2Period :=

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'Early'</td>
<td>'Middle'</td>
<td>'Middle'</td>
<td>'Late'</td>
</tr>
</tbody>
</table>
```

Aggregate(Cost_of_ownership, Time2Period, Time, Period ) → CarType ▼, Period ▼

<table>
<thead>
<tr>
<th>Early</th>
<th>Middle</th>
<th>Late</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
<td>2810</td>
<td>6049</td>
</tr>
<tr>
<td>Honda</td>
<td>3535</td>
<td>7744</td>
</tr>
<tr>
<td>BMW</td>
<td>3185</td>
<td>6703</td>
</tr>
</tbody>
</table>

Aggregate(Cost_of_ownership, Time2Period, Time, Period, type:'Average') → CarType ▼, Period ▼

<table>
<thead>
<tr>
<th>Early</th>
<th>Middle</th>
<th>Late</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
<td>2810</td>
<td>3025</td>
</tr>
<tr>
<td>Honda</td>
<td>3535</td>
<td>3872</td>
</tr>
<tr>
<td>BMW</td>
<td>3185</td>
<td>3352</td>
</tr>
</tbody>
</table>

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. When k is omitted and the result has two or more dimensions, a local index named .k will automatically be created by concatenating the elements of i and j, and the result will be indexed by this .k.

The parameter i (or j) can be omitted when a1 (or a2) is one-dimensional, if a1 (or a2) is indexed by a local index .k created by a previous call to Concat, when a1 (or a2) contains an implicit dimension, or when a1 (or a2) is atomic. The default to a local index .k makes it easy to nest calls to Concat when concatenating three or more arrays (or indexes) together. When a1 (or a2) is not array valued an i (or j) is omitted, a single element is concatenated to the front of a2 (or to the end of a1).
Examples

These examples use these variables:

IndexYears :=

2005 2006 2007 2008 2009

Index More_years:

2010 2011 2012

Index All_years := Concat(Years, More_years) →

2005 2006 2007 2008 2009 2010 2011 2012

More_prices: Car_type ▼, More_years ▶

<table>
<thead>
<tr>
<th>2010</th>
<th>2011</th>
<th>2012</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
<td>21K</td>
<td>22K</td>
</tr>
<tr>
<td>Honda</td>
<td>25K</td>
<td>28K</td>
</tr>
<tr>
<td>BMW</td>
<td>32K</td>
<td>33K</td>
</tr>
</tbody>
</table>

Concat(Car_prices, More_prices, Years, More_years, All_years) →

All_years ▼, Car_type ▶

<table>
<thead>
<tr>
<th>VW</th>
<th>Honda</th>
<th>BMW</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>16K</td>
<td>18K</td>
</tr>
<tr>
<td>2006</td>
<td>17K</td>
<td>19K</td>
</tr>
<tr>
<td>2007</td>
<td>18K</td>
<td>20K</td>
</tr>
<tr>
<td>2008</td>
<td>19K</td>
<td>22K</td>
</tr>
<tr>
<td>2009</td>
<td>20K</td>
<td>24K</td>
</tr>
<tr>
<td>2010</td>
<td>21K</td>
<td>25K</td>
</tr>
<tr>
<td>2011</td>
<td>22K</td>
<td>28K</td>
</tr>
<tr>
<td>2012</td>
<td>24K</td>
<td>29K</td>
</tr>
</tbody>
</table>

Example of nested usage and local .k index:

Concat(Car_prices, Concat(' ', Miles, , Years), Years) →

Car_type ▼, .k ▶

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
<td>16K</td>
<td>17K</td>
<td>18K</td>
<td>19K</td>
<td>20K</td>
<td>8000</td>
<td>7000</td>
<td>10K</td>
<td>6000</td>
</tr>
<tr>
<td>Honda</td>
<td>18K</td>
<td>19K</td>
<td>20K</td>
<td>22K</td>
<td>24K</td>
<td>10K</td>
<td>12K</td>
<td>11K</td>
<td>14K</td>
</tr>
<tr>
<td>BMW</td>
<td>25K</td>
<td>26K</td>
<td>28K</td>
<td>30K</td>
<td>32K</td>
<td>5000</td>
<td>8000</td>
<td>8000</td>
<td>7000</td>
</tr>
</tbody>
</table>

ConcatRows(a, rowIndex, colIndex, concatIndex, concatIndex)

Takes an array, a indexed by rowIndex and colIndex, and concatenates each row, flattening the array by eliminating the row dimension. The result is indexed by the concatIndex, which must be an index with size(rowIndex) * size(colIndex) elements. If concatIndex is omitted, a local index named .concatIndex is introduced defined as 1..N, where N is size(rowIndex) * size(colIndex).

Library Array

Example

ConcatRows(Car_prices, Car_type, Years) →

.CarIndex ▶

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>16K</td>
<td>17K</td>
<td>18K</td>
<td>19K</td>
<td>20K</td>
<td>...</td>
<td>28K</td>
<td>30K</td>
<td>32K</td>
</tr>
</tbody>
</table>

See “Example variables” on page 188 for example array variables used here and below.
IndexNames(a)

Returns a list of the identifiers of the indexes of the array \( a \) as text values.

**Library** Array

**Example**

\[
\text{IndexNames(Car_prices) } \rightarrow \left[ \text{'Car_type'}, \text{'Years'} \right]
\]

IndexesOf(a)

Returns a list of the indexes of the array \( a \) as handles (see “Handles to objects” on page 382). 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**

\[
\text{IndexesOf(Car_prices) } \rightarrow \left[ \text{Car_type, Years} \right]
\]

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**

\[
\text{Index L := [i, j, k, "value"]}
\text{Index rows := 1..Size(A)}
\text{Variable Flat_A := MdArrayToTable(A, rows, IndexValue(L))}
\]

Size(u, listLen)

Returns the number of atoms (elementary cells) in array \( u \). The size of an atom (including the special value null) is 1. The size of an empty list is 0.

Size(I) can be used to find the number of elements in an index \( I \), provided that \( I \) is a pure index and not a self-indexed array. When \( X \) is a self-indexed array, Size(\( X \)) returns the number of cells in the array, while Size(IndexValue(\( X \))) returns the number of elements of \( X \)'s self-index.

If an array \( A \) contains an implicit dimension, it is not possible to use Size(IndexValue(...)) to obtain the length of the implicit dimension, since the implicit dimension has no name to refer to it by. Setting the optional parameter listLen:true returns the length of the implicit index, or null if there it has no implicit index.

**Library** Array Functions

**Examples**

\[
\text{Size(Years) } \rightarrow 4
\text{Size(Car_prices) } \rightarrow 12
\text{Size(10) } \rightarrow 1
\text{Size([]) } \rightarrow 0
\text{Size([Years, Car_prices]) } \rightarrow 24
\text{Size([Years, Car_prices], listLen: true) } \rightarrow 2
\]

Subset(d, position, i, resultIndex )

Subset(d) returns the subset of d’s index values that correspond to true values in d. This basic usage, explained in “Subset(d)” on page 174, cannot be employed when d has more than one
dimension, that is, it does not array abstract. The optional parameters i and resultIndex provide an array-abstractable form that can be applied to multi-dimensional arrays, where the parameter i specifies the index to take the subset over, and resultIndex specifies the index for the final result. When the number of true elements exceeds N (the number of elements in resultIndex), the index values corresponding to the first N true values are returned. For the cases where there are fewer than N true values, the result is padded with null values. Setting the optional position parameter to true returns the index positions, rather than the index elements, of the true values.

Library Array

Example

Index MultiEventCars := 1..4

Subset(Miles > 8000,i:Car_type,resultindex:Car_type) → Car_type, Years

<table>
<thead>
<tr>
<th></th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW</td>
<td>Honda</td>
<td>Honda</td>
<td>VW</td>
<td>Honda</td>
<td>VW</td>
</tr>
<tr>
<td>Honda</td>
<td>&lt;&lt;null&gt;&gt;</td>
<td>&lt;&lt;null&gt;&gt;</td>
<td>Honda</td>
<td>&lt;&lt;null&gt;&gt;</td>
<td>Honda</td>
</tr>
<tr>
<td>BMW</td>
<td>&lt;&lt;null&gt;&gt;</td>
<td>&lt;&lt;null&gt;&gt;</td>
<td>&lt;&lt;null&gt;&gt;</td>
<td>&lt;&lt;null&gt;&gt;</td>
<td>BMW</td>
</tr>
</tbody>
</table>

In the above example, Miles values for Honda exceeded 8,000 in 2008 and 2009.
See “Example variables” on page 188 for example array variables used here and below.

**DetermTable: Deterministic tables**

A DetermTable provides an input view like that of an edit table (see page 178), 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 123) with the index for the `Choice()` function set to `Self`. You must then set the domain attribute to either `Explicit Values`, `Copy from index`, or `Expression`. The `Explicit Values` option allows you to exist all possible values directly. A `Copy from index` domain pulls the list of possible values from a separate index object that already contains the list of possible values. `Expression` is used when you want to compute the possible values based on other variables in your model. When using `Expression` for a `DetermTable`, the result must be a list or a call to the `Discrete()` function with the 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`...
3. Select **Array** from the **Library** popup menu and select **Deterministic Table** 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:

\[
\text{DeterministicTable}(i_1, i_2, \ldots, i_n)(r_1, r_2, r_3, \ldots, r_m)
\]

This describes an \(n\)-dimensional conditional deterministic table, indexed by the indexes and discrete conditioning variables \(i_1, i_2, \ldots, i_n\). The last index, \(i_n\), is the innermost index, varying the most rapidly. \(r_1, r_2, \ldots, r_m\) are the outcomes in the array.

**Converting a Table to a Deterministic Table**

To convert an existing table to a determTable, view either the Object Window or Attribute pane for the variable and use the pull-down to change the definition type to **Other...**. Answer **Yes** when asked to replace the current definition, and the **Object Finder** dialog (page 110) appears. From the **Array library** select **Deterministic Table** 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 **Deterministic Table**.

**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 trade-off. 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 determTables.

### 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.

`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:

\[ \text{Sum}(\text{MatrixA} \times \text{MatrixB}, i) \]

where \( i \) is the common index.

Example

Variable MatrixA:
\[
\begin{array}{ccc}
1 & 2 & 3 \\
a & 4 & 1 & 2 \\
b & 2 & 5 & 3 \\
c & 3 & 2 & 7 \\
\end{array}
\]

Variable MatrixB:
\[
\begin{array}{ccc}
1 & 2 & 3 \\
l & 3 & 2 & 1 \\
m & 2 & 5 & 3 \\
n & 4 & 1 & 2 \\
\end{array}
\]

\[ \text{Sum}(\text{MatrixA} \times \text{MatrixB}, i) \rightarrow \]
\[
\begin{array}{ccc}
a & b & c \\
l & 16 & 19 & 20 \\
m & 19 & 38 & 37 \\
n & 21 & 19 & 28 \\
\end{array}
\]

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. aCol and bRow must have the same length or it flags an error. If aRow and bCol are the same index, it returns only the diagonal of the result.

Example

Matrices
\[
\begin{array}{ccc}
1 & 2 & 3 \\
c & 1 & 2 & 1 \\
d & 1 & 2 & 0 \\
\end{array}
\]

\[
\begin{array}{ccc}
a & b & c \\
1 & 3 & 0 & 1 \\
2 & 0 & 1 & 1 \\
\end{array}
\]

\[
\text{MatrixMultiply}(C, \text{index1}, \text{index2}, D, \text{index2}, \text{index3}) \rightarrow \\
\text{index1}, \text{index3} \\
\]

\[
\begin{array}{ccc}
a & b & c \\
1 & 3 & 2 & 3 \\
2 & 3 & 0 & 1 \\
\end{array}
\]

When the inner index is shared by C and D, the expression \( \text{Sum}(C \times D, \text{index2}) \) is equivalent to their dot product (page 219).
**Tip**

The way to multiply a matrix by its transpose is:

```
```

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 ►
```

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

The conjugate transpose of a complex matrix is obtained using

```
Transpose(Conj(c),i,j)
```

---

**Identity matrix**

Given two indexes of equal length, the identity matrix is obtained by the expression `(i=j)`.

**Example**

```
@i=@j →
```

```
j ▼, i ►
```

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

---

**Unit vector**

A unit vector on index i is obtained by the expression `Array(i,1)`. There is no need to differentiate between a row vector and column vector, since it is the index that determines its orientation.

**Example**

```
Array(i,1) →
```

```
i ►
```

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

---

**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 ►
```

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.326</td>
<td>-0.034</td>
<td>-0.079</td>
</tr>
<tr>
<td>b</td>
<td>-0.056</td>
<td>0.247</td>
<td>-0.090</td>
</tr>
<tr>
<td>c</td>
<td>-0.124</td>
<td>-0.056</td>
<td>0.202</td>
</tr>
</tbody>
</table>
Determinant(c, i, j)

Returns the determinant of matrix c along dimensions i and j.

Example: MatrixA:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

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 \( v \cdot C \cdot v > 0 \), for all vectors \( v \).)

Cholesky decomposition computes a lower diagonal matrix \( L \) such that \( L \cdot L' = C \), where \( L' \) is the transpose of \( L \).

Example: Matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

Decompose(Matrix, L, M) →

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.4495</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.8165</td>
<td>1.8257</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2.4495</td>
<td>0.5477</td>
<td>1.6432</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1.2247</td>
<td>0</td>
<td>0</td>
<td>2.5495</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0.4082</td>
<td>1.4606</td>
<td>1.3389</td>
<td>1.3728</td>
<td>1.0113</td>
</tr>
</tbody>
</table>

EigenDecomp(a: Numeric[i, j]; i, j: Index)

Computes the Eigenvalues and Eigenvectors of a square, symmetric matrix a indexed by i and j. EigenDecomp() returns a result indexed by j and .item (where .item is a temporary index with the two elements {'value', 'vector'}). Each column of the result contains one Eigenvalue/Eigenvector pair. The Eigenvalue is a number, the Eigenvector is a reference to a rows-indexed Eigenvector. If result is the result of evaluating EigenDecomp(), then the Eigenvalues are given by result[.item='value'], and the Eigenvectors are given by #result[.item='vector']. Each Eigenvector is indexed by i.

Given a square matrix a, a non-zero number (\( \lambda \)) is called an Eigenvalue of a, and a non-zero vector \( x \) the corresponding Eigenvector of a when:

\[ a \cdot x = \lambda \cdot x \]
An $N \times N$ matrix does have $N$ (not-necessarily unique) Eigenvalue-Eigenvector pairs. When $A$ is a symmetric matrix, the Eigenvalues and Eigenvectors are real-valued. Eigen-analysis is widely used in Engineering and statistics.

**Tip**
The matrix $a$ must be square and symmetric. Mathematically, Eigen decompositions do exist for square non-symmetric matrices, but the algorithm used here is limited only to symmetric matrices, since symmetric decompositions are guaranteed to be real-valued, while, in general, Eigen decompositions can be complex.

**Library**
Matrix

**Example**

Covariance Matrix
stock1 ▼, stock2 ▼

<table>
<thead>
<tr>
<th></th>
<th>INTC</th>
<th>MOT</th>
<th>AMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTC</td>
<td>30.47</td>
<td>13.26</td>
<td>18.9</td>
</tr>
<tr>
<td>MOT</td>
<td>13.26</td>
<td>16.58</td>
<td>14.67</td>
</tr>
<tr>
<td>AMD</td>
<td>18.9</td>
<td>14.67</td>
<td>17.11</td>
</tr>
</tbody>
</table>

EigenDecomp(Covariance, Stock1, Stock2) ➔
.item ▼, stock2 ▼

<table>
<thead>
<tr>
<th></th>
<th>INTC</th>
<th>MOT</th>
<th>AMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1.025</td>
<td>9.232</td>
<td>53.9</td>
</tr>
<tr>
<td>vector</td>
<td>«ref_1»</td>
<td>«ref_2»</td>
<td>«ref_3»</td>
</tr>
</tbody>
</table>

SingularValueDecomp(a, i, j, j2)

**SingularValueDecomp()** (singular value decomposition) is often used with sets of equations or matrices that are singular or ill-conditioned (that is, very close to singular). It factors a matrix $a$, indexed by $i$ and $j$, with $\text{Size}(i) \geq \text{Size}(j)$, into three matrices, $U$, $W$, and $V$, such that:

$$ a = U \cdot W \cdot V $$

(7)

where $U$ and $V$ are orthogonal matrices and $W$ is a diagonal matrix. $U$ is dimensioned by $i$ and $j$, $W$ by $j$ and $j2$, and $V$ by $j$ and $j2$. In Analytica notation:

```
Variable A := Sum(Sum(U*W, J) * Transpose(V, J, J2), J2)
```

The index $j2$ must be the same size as $j$ and is used to index the resulting $W$ and $V$ arrays.

**SingularValueDecomp()** returns an array of three elements indexed by a special system index named $\text{SvdIndex}$ with each element, $U$, $W$, and $V$, being a reference to the corresponding array. Use the # (dereference) operator to obtain the matrix value from each reference, as in:

```
Index J2 := CopyIndex(J)
Variable SvdResult := SingularValueDecomp(A, I, J, J2)
Variable U := #SvdResult[SvdIndex='U']
Variable W := #SvdResult[SvdIndex='W']
Variable V := #SvdResult[SvdIndex='V']
```
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 224)
- Date functions for working with date numbers (see page 228)
- Advanced math functions (see page 231)
- Financial functions (see page 233)
- A library of extra financial functions, including functions for valuing options (see page 238)
- Advanced probability functions (see page 241)
Text functions

These functions work with text values (page 139) (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 mth through the nth character of text t (where the first character is m=1). If n is omitted it returns characters from the mth through the end of t.

- SelectText('One or two', 1, 3) → 'One'
- SelectText('One or two', 8) → 'two'

**FindInText(substr, text, start, case Insensitive, re, return, subpattern, repeat, repeatSubpattern, repeatIndex)**

Returns the position of the first occurrence of the text substr within 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

Setting the optional parameter, re, to True causes substr to be interpreted as a Perl-compatible regular expression. The optional return parameter alters what is returned by FindInText, according to the possible values:

- 'P' (or 'Position'): The position of the matching text or subpattern (default)
- 'L' (or 'Length'): The length of the matching text or subpattern.
• ‘S’ (or ‘Subpattern’): The text matched the regular expression or subpattern.

• ‘#’ (or ‘#Subpatterns’): The number of subpatterns in the regular expression.

Parentheses within a regular expression denote subpatterns, numbered in a depth first fashion. Subpatterns can also be named using the regular expression format “(?<name>...)”. You can the match information for any subpattern by specifying the subpattern number or subpattern name in the optional subpattern parameter.

```plaintext
FindInText('d.*T', 'FindInText', re:1) → 4
FindInText('d.*T', 'FindInText', re:1, return:['L','S']) → [4,'dInT']
FindInText('d\d\d\d-\d\d\d\d\d\d', '650-212-1212', re:1, return:'S', subpattern:[0,1,2]) → ['212-1212','212','1212']
FindInText('a*(?<bcd>b+(c+)(d*))','zyabdaabbcccdf', re:1, subpattern:['bcd',0,1,2,3]) → [8,6,8,10,13]
```

Normally FindInText returns information on the first match, but by using any of the three optional repeat parameters can be used to find all matches in text. When multiple matches are returned, the result will be an array and an index is required to index the matches found. When repeat:true is specified, the function creates a local index named .Repeat with elements 1..n to index the result for the n matches found. Alternatively, you can specify your own pre-existing index in the repeatIndex parameter. If your index contains n elements, then only the first n matches are returned. Finally, you can specify a regular expression subpattern, either numbered or named, in the repeatSubpattern parameter. In that case, a local index is created using the matching text for that subpattern as the index labels. When a named (textual) subpattern is specified, the subpattern name is used as a local index name.

The following examples parses XML text, returning an array of ages with a local index named .Name, where the labels of the local index are the names of each person:

```plaintext
FindInText('〈person.*?name="(?<name>(.*?))".*?〉.*?〈age>(?<age>.*?)〈/age〉.*?〉</person>', xmlText, re:true, return:'S', repeatSubpattern:'name', subpattern:'age')
```

**Tip** Consult the Analytica Wiki for more detailed information on using regular expressions. The Wiki contains additional information on regular expression syntax, and far more detail on the more advanced regular expression matching options. In the Analytica Wiki, see pages on Regular Expressions and FindInText.

### 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.

```plaintext
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(text, pattern, subst, all, caselnsensitive, re)

If all is omitted or False, it returns text with the first occurrence of pattern replaced by subst. If all is True, it returns text with all occurrences of text pattern replaced by subst. Pattern is matched in a case-sensitive fashion unless caselnsensitive is True.

```plaintext
TextReplace('StringReplace, StringLength', 'String', 'Text') → 'TextReplace, StringLength'
TextReplace('StringReplace, StringLength', 'String', 'Text', True)
```
TextReplace, TextLength

When the optional **re** parameter is True, **pattern** is treated as a Perl-compatible regular expression. In this mode, the character sequence \0 in **subst** is replaced by the matching text, and \1, \2, ..., \9 are replaced by the subtext matched by the corresponding numbered subpattern in the regular expression. The character sequence <name> in **subst** is replaced by the subtext matched to the indicated named subpattern.

```
TextReplace('Hello world', '\w+', '<\0>', all:True, re:True)
→ '«Hello» «world»'
TextReplace('Hello world', '.{1,7}.*', '\1..', re:True)
→ 'Hello w..'
TextReplace(text: 'swap first and last',
  pattern: '(?<first>\w+)(?<mid>.*)(?<last>\b\w+)',
  subst: '<last><mid><first>', re:True)
→ 'last first and swap'
TextReplace('swap first and last', '(\w)(\w*)(\w)', '\3\2\1',
  re:1, all:1 ) → 'pwas tirsf dna tasl'
```

### 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, default, textForNull)

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'])
B:= Array(I, ['VW', Null, '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, ';', ';') → 'VW; Honda; and BMW'
```

Null values in **a** are ignored unless the optional parameter **textForNull** is specified.

```
JoinText(B, I, '', textForNull:='') → 'one,two'
JoinText(B, I, '', textForNull:'') → 'one,,two'
JoinText(B, I, '', textForNull:'NULL') → 'one,NULL,two'
```

The optional **default** parameter is returned when all values are ignored, or **a** has a zero length.

```
JoinText([Null,Null,Null], default:Null) → «null»
```
SplitText(text, separator, caselnSensitivity, re)

Returns a list of text values formed by splitting the elements of text value text 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.'

When separator contains letters, setting caselnSensitivity to True matches in a lower/upper case-insensitive manner. When the re parameter is True, separator is interpreted as a Perl-compatible regular expression.

Variable s := 'Yes, Virginia. There is a Santa Claus!'  
SplitText(s, '\[s,\s+]\+', re:1) → ['Yes', 'Virginia', 'There', 'is', 'a', 'Santa', 'Claus', ' ']
SplitText(TextTrim(s, trimChars: ',.!', re:1), '\[s,\s+]\+', re:1) → ['Yes', 'Virginia', 'There', 'is', 'a', 'Santa', 'Claus']

Tip  With SplitText(), text 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 375 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'

NumberToText(x, format)

Converts a number to text using the format specified in the second parameter. The possible formats are: 'Suffix', 'Exponential', 'Fixed Point', 'Integer', 'Percent', 'Date', 'Boolean', or 'Hexadecimal'. For conciseness, the first letter of each may be used:

NumberToText(3.45M, ['S', 'E', 'F', 'I', 'H']) → ['3.45M', '3.45.e+006', '3450000', '3450000', '0x34a490']
NumberToText(0.0012, ['S', 'E', 'F', 'P']) →
Several optional parameters control aspects of each format. The parameter digits specifies the precision for Suffix and Exponential formats, and the digits to the right of the decimal for Fixed Point and Percent formats.

```plaintext
NumberToText(Pi, 'Suffix', digits:5) → '3.1416'
NumberToText(Pi, 'Fixed Point', digits:5) → '3.14159'
NumberToText(Pi, 'Percent', digits:5) → '314.15927%
```

You can specify showZeros as true to force the inclusion of trailing zeros.

```plaintext
NumberToText(1/4, 'Percent', digits:2, showZeros:True) → '25.00'
```

To include thousands separators, pass True to the thousandsSeparators parameter.

```plaintext
NumberToText(7^12, 'I', thousandsSeparators:True) → '13,841,287,201'
```

The currency parameter specifies a template that includes the currency symbol and its placement relative to the number and the minus sign.

```plaintext
```

When converting date-time numbers, the dateFormat parameter provides the date format template.

```plaintext
NumberToText(Today(), dateFormat:'yyyy MMMM dd (wwww)') → '2013 July 01 (Monday)'
```

There are places, such as when exporting data, where you want to make sure that the numbers are printed at full precision. When the fullPrecision parameter is true, extra digits are included when necessary to ensure that the full precision of the number is output.

```plaintext
NumberToText(Pi, fullPrecision:True) → '3.141592653589793'
```

### ParseNumber(text, badVal)

Parses a text value into a number. Dates are not parsed by this function (use ParseDate for dates). The result is independent of the number format setting. Values that are already numeric are returned. The optional badVal parameter specifies the value returned when text is unparsable, which defaults to Null. The usage ParseNumber(x, x) can be used when x is an array and you want to pass unparsable entries through.

```plaintext
ParseNumber('12.43K') → 12.43K
ParseNumber('hello') → «null»
ParseNumber(14.3) → 14.3
Var x:=['3,214', 14, 'foo'] Do ParseNumber(x, x) → [ 3214, 14, 'foo']
```

### 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 82. 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.
Date functions

Examples

MakeDate(2007, 5, 15) → 15-May-2007
MakeDate(2000) → 1-Jan-2000

MakeTime(h, m, s)

Gives the fraction of a day elapsed since midnight for the given hour, minute and second. The hour, h, is typically between 0 and 23 inclusive (but can be greater than 23 when encoding a duration of more than one day). Minutes and seconds must be between 0 and 59 inclusive.

Examples

MakeTime(12, 0, 0) → 0.5
MakeTime(15, 30, 0) → 0.6458 { 3:30:00 pm }

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.
- Q gives the quarter as a number between 1 and 4.
- 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 clock (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', wwws → '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 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 week 1 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-')
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 \( Y, Q, M, D, WD, h, m, \) 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:

\[
\text{DateAdd}(\text{MakeDate}(2006, 1, 1), 0..12, "M") \\
\rightarrow \{ "1 Jan 2006", "1 Feb 2006", "1 Mar 2006", ... "1 Jan 2007" \}
\]

If an offset would appear to go past the end of a month, it returns the last day of the month:

\[
\text{DateAdd}( \text{MakeDate}(2004, 2, 29), 1, 'Y' ) \rightarrow 2005-Feb-28 \\
\text{DateAdd}( \text{MakeDate}(2006, 10, 31), 1, 'M' ) \rightarrow 2006-Nov-30
\]

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

ParseDate(date,badVal)

Parses a textual date or time into a numeric value representing the number of days elapsed since the date origin. The parsing occurs independent of the number format setting for the variable being evaluated. The second optional parameter, `badVal`, specifies the return value when `date` is not textual or cannot be parsed as a date. When omitted, `badVal` default to `Null`.

\[
\begin{align*}
\text{ParseDate}("July 22, 2009") & \rightarrow 2009-Jul-22 \{ 38554 \} \\
\text{ParseDate}("38554") & \rightarrow \text{null} \\
\text{ParseDate}("3:00 pm") & \rightarrow 0.625 \\
\text{ParseDate}("7/22/2009 15:00:00") & \rightarrow 2009-Jul-22 3:00pm \{ 38554.625 \} \\
\end{align*}
\]

\[
\text{Var } x := ["hello","7-22-2009"] \text{ Do ParseDate}(x,x) \rightarrow ["hello",38554]
\]

**Note:** The results in this example assume the default date origin of 1-Jan-1904 and that Windows is set to United States regional settings.

**Sequence(start,end,dateTime:...)**

The `Sequence` function, described on page 173, can be used to create a sequence of dates or date-times. A date or time sequence is created when `start` and `end` are date-time numbers, or when the `dateUnit` parameter is specified. The `dateUnit` parameter can be any of \( Y, Q, M, D, WD, h, m, \) or \( s \). Note that options \( Y, Q, M, \) and \( WD \) result in increments that have date-aware spacings, but in which the increment measured in days actually varies, due to variations in the lengths of months, the positions of weekends, and the presence of leap years.
Sequence(9-Aug-2012, 15-Aug-2012, dateUnit:"WD") →

Sequence(1-Jan-2012, 1-Jan-2013, dateUnit:"Q") →

Sequence(1-Jan-2012, 1-May-2012, dateUnit:"M") →

The step parameter can also be specified, for example, to step in increments of 2 months. When specified with a dateUnit increment, step must be an integer value. If it is not an integer, it is rounded down.

Ceil(x,dateUnit:...), Floor(x,dateUnit:...), Round(x,dateUnit:...)

These rounding functions (described in "Math functions" on page 143) can be used to round dates or times to the nearest date unit (i.e., nearest year, month, day, weekday, hour, minute or second) by specifying the optional dateUnit parameter. The date unit can be one of the following values: "Y", "Q", "M", "D", "WD", "h", "m", or "s". To specify dateUnit, you should use a named-parameter syntax.

Ceil(11-Aug-2012,dateUnit:['"Y","Q","WD","Q"])) →

Floor(11-Aug-2012,dateUnit:['"Y","Q","WD","Q"])) →

Round(11-Aug-2012,dateUnit:['"Y","Q","WD","Q"])) →

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 143. There are additional advanced math functions covered in “Importance weighting” on page 285.

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 cylindri-
Cosh(x), Sinh(x), Tanh(x)

The hyperbolic cosine, sine, and tangent of \( x \), \( x \) assumed to be in degrees.

\[
\begin{align*}
\text{Cosh}(0) & \rightarrow 1 \\
\text{Sinh}(0) & \rightarrow 0 \\
\text{Tanh}(\text{INF}) & \rightarrow 1
\end{align*}
\]

Lgamma(x)

Returns the Log Gamma function of \( x \). Without numeric overflow, this function is equivalent to \( \ln(\text{GammaFn}(X)) \). Because the gamma function grows so rapidly, it is often much more convenient to use \( \text{LGamma()} \) to avoid numeric overflow.

\[
\text{LGamma}(10) \rightarrow 12.8
\]

Complex number functions

As a general rule, you can pass a complex number to nearly any of the built-in mathematical functions described elsewhere in this manual for which a complex variation of the function is meaningful. For example, you can evaluate \( \text{Sqrt}(3+2j) \), \( \text{Exp}(4-1j) \), \( \text{Ln}(-1-3j) \), \( \text{ArcCos}(1-2j) \), \((3+2j)^{(1-3j)}\), and so on.

EnableComplexNumbers

There are several standard mathematical functions that produce a complex number when passed a real-valued parameter -- for example, \( \text{Sqrt}(-1) \) or \( \text{ArcCos}(2) \). These will produce a warning and return \( \text{NaN} \) unless you have enabled complex numbers. To enable complex numbers, make sure you are in edit mode with no object selected, then select Definition → System Variables → EnableComplexNumbers. Change the definition to 1. After it is changed, these functions return a complex result without a warning.

Abs(x)

The magnitude of a complex number.

\[
\text{Abs}(3 - 4j) \rightarrow 5
\]

ComplexDegrees(x)

The phase of a complex number in degrees, between -180 and 180.

\[
\text{ComplexDegrees}(-1+1j) \rightarrow 135
\]

ComplexRadians(x)

The phase of a complex number in radians, between \( -\pi \) and \( \pi \).

\[
\text{ComplexRadians}(-1j) \rightarrow -1.571
\]
Conj(x)

The complex conjugate on a number.

\[ \text{Conj}(4+8j) \rightarrow 4-8j \]

ImPart(x)

The imaginary part of a complex number.

\[ \text{ImPart}(3+4j) \rightarrow 4 \]

RealPart(x)

The real part of a complex number.

\[ \text{RealPart}(3+4j) \rightarrow 3 \]

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.

- **pv**: The period (between 1 and nPer) being computed.

- **pv**: The present value of the annuity. For example, for a loan this is the loan amount (positive if you receive the loan, negative if you are the lender).

- **fv**: The future value of the annuity. This is the remaining value of the annuity after the final payment. In the case of a loan, for example, this is the balloon payment at the end (positive if you are the lender, negative if you pay the balloon amount). This parameter is usually optional with a default value of zero.

- **pmt**: The total payment per period (interest + principal). If you receive payments, this is positive. If you make payments, this is negative.

- **type**: Indicates whether payments are due at the beginning or end of each period.

- **True**: Payments are due at the beginning of each period, with the first payment due immediately.

- **False** (default): Payments are due at the end of each period.

Cumipmt(rate, nPer, pv, startPeriod, endPeriod, type)

Returns the cumulative interest paid on an annuity between, and including, startPeriod (shown as \(sp\) in equation below) and endPeriod (shown as \(ep\) in equation below). The annuity is assumed to have a constant interest rate and periodic payments. This is equal to:

\[ \sum_{n = sp}^{ep} \text{Ipmt}(rate,n,nPer,pv,0,\text{Type}) \]

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

<table>
<thead>
<tr>
<th>2009</th>
<th>2010</th>
<th>2011</th>
<th>2012</th>
<th>2013</th>
<th>2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1M</td>
<td>-500K</td>
<td>-100K</td>
<td>100K</td>
<td>1M</td>
<td>2M</td>
</tr>
</tbody>
</table>

Irr(Earnings, Time) → 17.15%

**MIrr(values, i, financeRate, reinvestRate)**

Computes the modified internal rate of return for a series of periodic cash flows, given in values over the index i. The MIrr is the rate of return of an investment when capital invested must be borrowed at financeRate, and intermediate returns are re-invested at reinvestRate. Because the result of MIrr is expressed as a rate-of-return, it shares the intuitive appeal of Irr as a measure of the quality of a cash flow, while avoiding the many pitfalls and distortions associated with Irr. The MIrr is defined by the following formula

\[
MIrr(x, i, f, r) = \left( \frac{Npv(r, x \cdot (x > 0), i) \cdot (1 + r)^{n + 1}}{Npv(f, -x \cdot (x < 0), i) \cdot (1 + f)^{n + 1}} \right)^{1/n} - 1
\]

Example

<table>
<thead>
<tr>
<th>2009</th>
<th>2010</th>
<th>2011</th>
<th>2012</th>
<th>2013</th>
<th>2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1M</td>
<td>-500K</td>
<td>-100K</td>
<td>100K</td>
<td>1M</td>
<td>2M</td>
</tr>
</tbody>
</table>

MIrr(Earnings, Time, 8%, 4%) → 15.24%

**Tip**

To compute MIrr for a non-periodic cash flow, use XMirr().

**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, offset)**

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 optional offset parameter specifies the offset of the first value relative to the current time period. By default, offset is 1, indicating that the first value is discounted as if it is one step in the future. Npv(.,.,offset: 0) applies no discount to the first value, which should be used when the cash flow starts in the current time period. The NPV is given by:

\[
\sum_{j=offset}^{n+1-offset} \frac{Values[i=j]}{(1+DiscountRate)^j}
\]

The first value is discounted as if it is one step in the future. To treat the first value as occurring in the first time period, set the optional offset parameter to zero.
Tip
To compute the NPV for a non-periodic cash flow, use Xnpv().

**Example**

**Earnings: Time**

<table>
<thead>
<tr>
<th>Year</th>
<th>Cash Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999</td>
<td>-1M</td>
</tr>
<tr>
<td>2000</td>
<td>-500K</td>
</tr>
<tr>
<td>2001</td>
<td>-100K</td>
</tr>
<tr>
<td>2002</td>
<td>100K</td>
</tr>
<tr>
<td>2003</td>
<td>1M</td>
</tr>
<tr>
<td>2004</td>
<td>2M</td>
</tr>
</tbody>
</table>

At a discount rate of 5%, the net present value of this cash flow is:

\( \text{Npv}(5\%, \text{Earnings}, \text{Time}) \rightarrow \$865,947.76 \)

**Pmt(rate, nPer, pv, fv, type)**

Returns the total payment per period (interest + principal) for an annuity with constant periodic payments and fixed interest rate.

**Example**

You obtain a 30-year fixed mortgage at 8%/12 per month for $100K. Your monthly payment is (note that the result of \( Pmt() \) is negative since this is a payment for you):

\(-Pmt(8%/12, 30*12, 100K) \rightarrow \$733.76\)

**Ppmt(rate, per, nPer, pv, fv, type)**

Returns the principal portion of a payment on an annuity with constant period payments and fixed interest rate.

**Example**

You have a 30-year fixed $100K loan at a rate of 8%/12 monthly. On your 24th payment, the amount of your payment that goes towards principal is (note that the result of \( PPmt() \) is negative since this is a payment for you):

\(-PPmt(8%/12, 24, 12*30, 100K) \rightarrow \$78.18\)

**Pv(rate, nPer, pmt, fv, type)**

Returns the present value of an annuity. The annuity is assumed to have constant periodic payments to you of \( pmt \) per period for \( nPer \) periods, with a return of \( rate \) per period.

**Example**

To receive $100 per month from an annuity that returns 6%/12 per month for the next 10 years, you would need to invest (note that the result from \( Pv() \) is negative since you are paying to make the investment):

\(-Pv(6%/12, 10*12, 100) \rightarrow \$9,007.35\)

**Rate(nPer, pmt, pv, fv, type, guess)**

Returns the interest rate (per period) for an annuity. The value returned is the interest rate that results in equal payments of \( pmt \) per period over the \( nPer \) periods of the annuity.

In general, \( Rate() \) 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:

\( Rate(30, Pmt(7\%, 30, 100K), 100K-3500) \rightarrow 7.36\%\)

**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 \texttt{Xirr()} returns \texttt{NaN}. If the cash flow changes sign more than once, \texttt{Xirr()} can have multiple solutions, but returns only the first solution found. The optional parameter, \texttt{guess}, can be provided as a starting point for the iterative search, and \texttt{Xirr()} generally finds the solution closest to \texttt{guess}. If not provided, \texttt{guess} defaults to 10%. If no solution is found within 30 iterations, \texttt{Xirr()} returns \texttt{NaN}.

To compute the IRR for a series of period payments, use \texttt{Irr()}.

\textbf{Example}

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{EarningAmt}</td>
<td>-400K</td>
<td>-200K</td>
<td>100K</td>
<td>600K</td>
</tr>
</tbody>
</table>

\texttt{EarningDate: J}  

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{EarningDate}</td>
<td>July 5, 2009</td>
<td>Dec 1, 2009</td>
<td>Jan 21, 2010</td>
<td>Aug 10, 2011</td>
</tr>
</tbody>
</table>

\texttt{Xirr(EarningAmt, EarningDate, J) → 9.32%}

\textbf{Tip}  

\texttt{EarningDate} can be entered by selecting \texttt{Number Format} from the \texttt{Result} menu while editing the table for \texttt{EarningDate}. From the \texttt{Number format} dialog, select a date format, then enter the dates.

\texttt{XMIrr(values, dates, i, financeRate, reinvestRate)}

Computes the modified internal rate of return for a series of non-periodic cash flows occurring on arbitrary dates. The parameters \texttt{values} and \texttt{dates} should share the index \texttt{i}, with \texttt{values} containing the cash flow amounts and \texttt{dates} containing the corresponding date on which that cash flow occurs. Each date is specified as a number indicating the number of days elapsed since the date origin. The \texttt{MIRR} is the rate of return of an investment when capital invested must be borrowed at \texttt{financeRate}, and intermediate returns are re-invested at \texttt{reinvestRate}.

\textbf{Example}

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{EarningAmt}</td>
<td>-400K</td>
<td>-200K</td>
<td>100K</td>
<td>600K</td>
</tr>
</tbody>
</table>

\texttt{EarningDate: J}  

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{EarningDate}</td>
<td>July 5, 2009</td>
<td>Dec 1, 2009</td>
<td>Jan 21, 2010</td>
<td>Aug 10, 2011</td>
</tr>
</tbody>
</table>

\texttt{XMIrr(EarningAmt, EarningDate, J, 8%, 4%) → 8.62%}

\texttt{Xnpv(rate, values, dates, i)}

Returns the net present value (NPV) of a non-periodic cash flow with a constant discount rate. \texttt{rate} is the annual discount rate for a 365 day year. Both \texttt{values}, the cash-flow amounts, and \texttt{dates}, the date of each payment (negative value) or inflow (positive value), must be indexed by \texttt{i}.

See also \texttt{Npv()}.  

\textbf{Example}

Using the cash flow shown in the example for \texttt{Xirr()} above, the net present value at a 5% discount rate is:

\texttt{Xnpv(5%, EarningAmt, EarningDate, J) → $42,838.71}
YearFrac(startDate, endDate, basis)

Returns the fraction of the year represented by the number of whole days between two dates. The startDate and endDate are numeric, denoting the number of days elapsed since the date origin. The result is always positive, with the result being the span between the lesser and greater of the two dates. Use this function when you wish to compute the proportion of a whole year as applicable to certain financial instruments. The optional basis parameter selects the accounting method to be used, as follows:

0 (or omitted) = US (NASD) 30/360
1 = actual/actual
2 = Actual/360
3 = Actual/365
4 = European 30/360

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,

**Parameters**
- $R_f$ = risk free rate
- $R_m$ = 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 covariation 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**
- $r_{Opp}$ = opportunity cost of capital
- $r_D$ = expected return on debt
- $T_c$ = 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**
- $r_{AllEq}$ = cost of capital under all-equity financing
- $T_c$ = 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
Chapter 14 Other Functions

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)
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:

\[ \text{BetaFn}(a, b) = \int_0^1 x^{a-1}(1-x)^{b-1} \, dx \]

Betal(x, a, b) The incomplete beta function, defined as:

\[ \text{Betal}(x, a, b) = \frac{1}{\text{Beta}(a, b)} \int_0^x x^{a-1}(1-x)^{b-1} \, dx \]

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:

\[ P = \text{Betal}(p, k, n-k+1) \]

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:

\[ \text{Student}(x \mid n) = 1 - \text{Betal}(n/(n+x^2), n/2, 1/2) \]

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 Betal as:

\[ F(x, n_1, n_2) = \text{Betal}(n_2/(n_1x + n_2)) \]

BetalInv(p, x, b) The inverse of the incomplete beta function with respect to the second parameter, \( a \). Returns the value \( a \) such that \( \text{Betal}(x, a, b)=p \).

BetalInv(p, a, b) The inverse of the incomplete beta function with respect to the first parameter, \( x \). Returns the value \( x \) such that \( \text{Betal}(x, a, b)=p \).

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).

\[ \text{Combinations}(2, 4) \rightarrow 6 \]

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.

\[ \text{Permutations}(2, 4) \rightarrow 12 \]

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:

\[ p = \text{Pr}[x \leq X] \]
for a normal distribution with a given mean and standard deviation. **mean** and **stddev** are optional and default to **mean** = 0, **stddev** = 1.

\[ \text{CumNormal}(1) - \text{CumNormal}(-1) \rightarrow .683 \]

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**. This is also called the **Probit function**. Returns the value **X** where:

\[ p = Pr[x \leq X] \]

**mean** and **stddev** are optional and default to **mean** = 0, **stddev** = 1.

**Erf(x)** The error function, defined as:

\[ \text{Erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \]

**ErfInv(y)** The inverse error function. Returns the value **X** such that \( \text{Erf}(X) = y \).

**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, \( \text{GammaFn}(n+1) = n! \). For this reason, it is often preferable to use the **LGamma()** function.

**GammaI(x, a, b)** Returns the incomplete gamma function, defined as:

\[ \text{GammaI}(x, a, b) = \frac{1}{\Gamma(a)} \int_0^{x/b} e^{-t} t^{b-1} dt \]

**a** is the shape parameter, **b** is an optional scale factor (default **b**=1). Some textbooks use \( \lambda = 1/a \) as the scale factor. The incomplete gamma function is defined for \( x \geq 0 \).

The incomplete gamma function returns the cumulative area from zero to **x** under the gamma distribution.

The incomplete gamma function is useful in a number of mathematical and statistical contexts.

The cumulative Poisson distribution function, which encodes the probability that the number of Poisson random events (**x**) occurring will be less than **k** (where **k** is an integer) where the expected mean number is **a**, is given by (recall that parameter **b** is optional).

\[ P(x < k) = \text{GammaI}(k, a) \]

**GammaInv(y, a, b)** The inverse of the incomplete gamma function. Returns the value **x** such that \( \text{GammaI}(x, a, b) = y \). **b** is optional and defaults to 1.

**Logit(p)** The **logit** function, or **log-odds function**, is defined for \( 0 < p < 1 \) as

\[ \text{Logit}(p) = \ln\left(\frac{p}{1-p}\right) \]

The inverse of the **Logit** function is the **Sigmoid** function.

**Sigmoid(x)** The sigmoid function approaches 0 as **x** approaches \( -\infty \) and approaches 1 as **x** approaches \( \infty \), crossing 0.5 at **x**=0. It is also sometimes called the **expit function** or the **inverse logit function**. It is defined as
The inverse of the Sigmoid function is the Logit function.

\[
Sigmoid(x) = \frac{1}{1 + \exp(-x)}
\]
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
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?

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.
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 20) of the diagram with Definition (page 104) selected from the Attribute popup menu.

To define the distribution:

1. Click the expr menu above the definition field and select Distribution.
Defining a variable as a distribution

The Object Finder opens, showing the Distribution library.

2. Select the distribution you wish to use.

3. Enter the values for the parameters. You can use an expression or refer to other variables by name in the parameter fields.

4. Click OK to accept the distribution.

If the parameters of the distribution are single numbers, a button appears with the name of the distribution, indicating that the variable is defined as a distribution. To edit the parameters, click this button.

If the parameters of the distribution are complex expressions, the distribution displays as an expression. For example:

\[ \text{Normal}((\text{Price}/\text{Mpy}) \times \text{Mpg}, \text{Mpg}/10) \]

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:

\[ \text{Normal}(0.105, 0.015) \]
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 29) — 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

\[
A := \text{Normal}(10, 2) \rightarrow \\
\text{Iteration (Run)}
\]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10.74</td>
<td>13.2</td>
<td>9.092</td>
<td>11.44</td>
<td>9.519</td>
<td>13.03</td>
</tr>
</tbody>
</table>

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 251).

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

\[ B := \text{Normal}(5, 1) \rightarrow \]

<table>
<thead>
<tr>
<th>Iteration (Run)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5.09</td>
<td>4.94</td>
<td>4.65</td>
<td>6.60</td>
<td>5.24</td>
<td>6.96</td>
</tr>
</tbody>
</table>

\[ C := A + B \rightarrow \]

<table>
<thead>
<tr>
<th>Iteration (Run)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15.83</td>
<td>18.13</td>
<td>13.75</td>
<td>18.04</td>
<td>14.76</td>
<td>19.99</td>
</tr>
</tbody>
</table>

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 251).

Tip

If you try to apply an array-reducing function (page 192) to a probability distribution across Run, Analytica returns the distribution’s mid value.

Example:

\[ X := \text{Beta}(2, 3) \]

\[ \text{Mid}(X) \rightarrow 0.3857 \text{ and } \text{Max}(X, \text{Run}) \rightarrow 0.3857 \]

To evaluate the input parameters probabilistically and reduce across Run, use Sample() (page 295).

Example:

\[ \text{Max(Sample}(X), \text{Run}) \rightarrow 0.8892 \]

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.

<table>
<thead>
<tr>
<th>Analysis option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncertainty Sample</td>
</tr>
<tr>
<td>Statistics</td>
</tr>
<tr>
<td>Probability Bands</td>
</tr>
<tr>
<td>Probability Density</td>
</tr>
<tr>
<td>Cumulative Probability</td>
</tr>
</tbody>
</table>

Uncertainty sample

To change the sample size or sampling method for the model, select the Uncertainty Sample option from the Analysis option popup menu.
The default dialog 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 416 for guidelines on selecting a sample size. The sample size must be between 2 and 32,000. You can access this number in expressions in your models as the system variable Samplesize.

Sampling method  The sampling method is used to determine how to generate a random sample of the specified sample size, \( m \), for each uncertain quantity, \( X \). Analytica provides three options:

- **Simple Monte Carlo**

  The simplest sampling method is known as Monte Carlo, named after the randomness prevalent in games of chance, such as at the famous casino in Monte Carlo. In this method, each of the \( m \) sample points for each uncertainty quantity, \( X \), is generated at random from \( X \) with probability proportional to the probability density (or probability mass for discrete quantities) for \( X \). Analytica uses the inverse cumulative method; it generates \( m \) uniform random values, \( u_i \) for \( i = 1, 2, ..., m \), between 0 and 1, using the specified random number method (see below). It then uses the inverse of the cumulative probability distribution to generate the corresponding values of \( X \),

  \[
  X_i \text{ where } P(x \leq X_i) = u_i \text{ for } i = 1, 2, ..., m.
  \]

  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 416.

- **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

  \[ Xi \text{ where } P(x \leq Xi) = (i-0.5)/m, \text{ for } i=1,2,...,m. \]

  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:

\[ X := \text{Uniform}(1, \text{Samplesize}) \]
\[ Y := \sin(2\pi X) \]

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.
Chapter 15  Expressing Uncertainty

Uncertainty Setup dialog

**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 option

To change how probability density is estimated and drawn, select **Probability Density** from the **Analysis option** popup menu.

Analytica estimates the probability density function, like other uncertainty views, from the underlying array of sample values for each uncertain quantity. The probability density is highly susceptible to random sampling variation and noise. Both histogramming and kernel density smoothing techniques are available for estimating the probability density from the sample, but to ultimately reduce noise and variability it may be necessary to increase sample size (for details on selecting the sample size, see “Appendix A: Selecting the Sample Size” on page 416). The following example graphs compare the two methods on the same uncertain result:

**Histogram**

The histogram estimation methods partition the space of possible continuous values into bins, and then tally how many samples land in each bin. The probability density is then equal to the fraction of the Monte Carlo sample landing in a given bin divided by the bin’s width. The average number of points landing in each bin determines both the smoothness of the resulting function and the resolution of the resulting plot. With more bins, a finer resolution is obtained, but since fewer points land in each bin, the amount of random fluctuation increases resulting in a noisier plot. The **Samples per PDF step interval** setting sizes the bin width to match the average number of points per bin.

With larger sample sizes, you can increase the **Samples per PDF step interval** to achieve smoother plots, since more samples will land in each bin. A number approximately equal to the square root of sample size tends to work fairly well.

You can also control how the partitioning of the space of values is performed. When **Equal X axis steps** is used, the range of values from the smallest to largest sample point is partitioned into equal sized bins. With this method, all bins have the same width, but the number of points falling in each bin varies. When **Equal weighted probability steps** is used, the bins are sized so that
each bin contains approximately the same fraction of the total probability. With this method, the fraction of the sample in each bin is nearly constant, but the width of each bin varies. When Equal sample probability steps is used, the bins are partitioned so that the number of sample points in each bin is constant, with the width of each bin again varying. Equal weighted probability steps and Equal sample probability steps are exactly equivalent when standard equally-weighted Monte Carlo or Latin Hypercube sampling is being used. They differ when the SampleWeighting system variable assigns different weights to each sample along the Run index, as is sometimes employed with importance sampling, logic sampling for posterior analysis, and rare-event modeling. See “Importance weighting” on page 285.

Probability density plots using the histogram method default to the Step chart type, which emphasizes the histogram and reveals the bin placements. When desired, this can be changed to the standard line style from the Graph Setup, see “Chart Type tab” on page 86.

Smoothing

The smoothing method estimates probability density using a technique known as Kernel Density Estimation (KDE) or Kernel Density Smoothing. This technique replaces each Monte Carlo sample with a Gaussian curve, called a Kernel, and then sums the curve to obtain the final continuous curve. Unlike a histogram, the degree of smoothness and the resolution of the plot are independent. The Smoothing factor controls the smoothness or amount of detail in the estimated PDF. The more info button ( ) next to the Smoothing radio control jumps to a page on the Analytica Wiki that elaborates in more detail on how kernel density smoothing works.

Due to the randomness of Monte Carlo sampling, estimations of probability density are often quite noisy. The Smoothing method can often provide smoother and more intuitively appealing plots than Histogram methods, but the averaging effects inherent in smoothing can also introduce some minor artifacts. In particular, Smoothing tends to increase the apparent variance in your result slightly, with a greater increase when the Smoothing factor is greater. This increase in variance is also seen as a decrease in the height of peaks. Sharp cutoffs (such as is the case with a Uniform distribution, for example) become rounded with a decaying tail past the cutoff point. And when positive-only distributions begin with a very sharp rise, the density estimate may get smoothed into a plot with a tail extending into the negative value territory.

Cumulative probability option

To change how the cumulative probability values are drawn or to change their resolution, select the respective option from the Analysis option popup menu.

Analytica estimates the 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. Cumulative probability estimates are less susceptible to noise than, for example, probability density estimates.

The Samples per CDF plot point setting controls the average number of sample values used to estimate each point on the cumulative distribution function (CDF) curve, which ultimately controls the number of points plotted on your result.
The Equal X axis steps, Equal weighted probability steps and Equal sample probability steps control which points are used in plot of the cumulative probability. **Equal X axis steps** spaces points equally along the X axis. **Equal weighted probability steps** uses the sample to estimate a set of \( m+1 \) fractiles (quantiles), \( x_p \), at equal probability intervals, where \( p=0, \frac{1}{m}, \frac{2}{m}, ... 1, \) and \( q = 1/m \). The cumulative probability is plotted at each of the points \( x_p \), increasing in equal steps along the vertical axis. Points are plotted closer together along the horizontal axis in the regions where the density is the greatest. **Equal sample probability steps** plots one point each at each nth sample point, where n is the sample per CDF plot point, ignoring the weight on each sample point when they are weighted differently. The cumulative probability up to the nth point is estimated and plotted. **Equal weighted probability steps** and **Equal sample probability steps** are exactly equivalent unless unequal sample weighting is employed (see “Importance weighting” on page 285).
Chapter 16  

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 246 and “Glossary” on page 439 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 267), 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 281). You may set parameters to array values, specify an index to the optional Over parameter, or use functions from the Multivariate library.

<table>
<thead>
<tr>
<th>Parametric Discrete</th>
<th>Parametric Continuous</th>
<th>Multivariate</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Bernouli() page 261</td>
<td>• Uniform() page 270</td>
<td>• Normal_correl() page 282</td>
</tr>
<tr>
<td>• Binomial() page 261</td>
<td>• Triangular() page 270</td>
<td>• Correlate_with() page 282</td>
</tr>
<tr>
<td>• Poisson() page 261</td>
<td>• Normal() page 271</td>
<td>• Dist_reshape() page 283</td>
</tr>
<tr>
<td>• Geometric() page 262</td>
<td>• Lognormal() page 271</td>
<td>• Correlate_dists() page 283</td>
</tr>
<tr>
<td>• Hypergeometric() page 262</td>
<td>• Beta() page 272</td>
<td>• Gaussian() page 283</td>
</tr>
<tr>
<td>• NegativeBinomial() page 262</td>
<td>• Exponential() page 273</td>
<td>• Multinormal() page 283</td>
</tr>
<tr>
<td>• Uniform() page 262</td>
<td>• Gamma() page 274</td>
<td>• BiNormal() page 283</td>
</tr>
<tr>
<td>• Wilcoxon() page 263</td>
<td>• Logistic() page 274</td>
<td>• Dirichlet() page 283</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Custom Discrete</th>
<th>Custom Continuous</th>
<th>Special Probabilistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Probtable() page 266</td>
<td>• Cumdist() page 277</td>
<td>• Certain() page 279</td>
</tr>
<tr>
<td>• Determtable() page 268</td>
<td>• Probdist() page 278</td>
<td>• Shuffle() page 279</td>
</tr>
<tr>
<td>• Chancedist() page 269</td>
<td></td>
<td>• Truncate() page 279</td>
</tr>
<tr>
<td></td>
<td>• Normal_additive_gro() page 285</td>
<td>• Random() page 280</td>
</tr>
</tbody>
</table>

There are many other distributions, which can be found in the library and can be used to create more complex distributions.
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:

\[
\text{If Uniform}(0, 1) < P \text{ Then 1 Else 0}
\]

Library: Distribution

Example: The domain, List of numbers, is \([0, 1]\).

\[
\text{Bernoulli} \_\_\_\_ := \text{Bernoulli} \ (0.3) \to
\]

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:

\[
\text{Function Binomial2}(n, p) \\
\text{Parameters: (n: Atom; p)} \\
\text{Definition: Index i := 1..n; } \\
\text{Sum(FOR J := I DO Bernoulli(p), i)}
\]

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.
Chapter 16 Probability Distributions

Parametric discrete distributions

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.

NegativeBinomial(r, p)

The negative binomial distribution describes the number of successes that occur before $r$ failures occur, where each independent Bernoulli trial has a probability $p$ of being a success. The domain is the set of non-negative integers. The Geometric distribution is a special case when $r=1$. When $r>1$ the distribution tends to be bell shaped.

There are three common bell-shaped discrete distribution types on the integers 0, 1, 2, ..., namely the binomial, Poisson and the negative binomial. A binomial has a variance that is less than the mean, a Poisson has a variance equal to the mean, and a negative binomial has a variance greater than the mean.

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.

$$\text{Uniform}(5, 14, \text{Integer: True})$$
Wilcoxon(m,n,exact)

The Wilcoxon distribution characterizes the distribution of the Wilcoxon rank-sum statistic, known as the U-statistic, that is obtained when comparing two populations with unpaired samples having sizes \(m\) and \(n\). The rank-sum test is a non-parametric alternative to the Student-t test, which unlike the Student-t test, does not assume that the distribution is normally distributed. The distribution is non-negative and discrete, and approaches a normal distribution for large \(m+n\).

Given two samples, \(x\) of size \(m\) and \(y\) of size \(n\), the U-statistic is the number of pairs \((x_i, y_j)\) in which \(x_i < y_j\).

Like other distribution functions in Analytica, the \(\text{Wilcoxon}(m,n)\) function generates a random sample, which in this case would be the distribution of U-statistic values when two samples of size \(m\) and \(n\) and equal distributions are compared. Also available as built-in functions are \(\text{ProbWilcoxon}(u,m,n)\), \(\text{CumWilcoxon}(u,m,n)\) and \(\text{CumWilcoxonInv}(p,m,n)\), the analytic probability distribution, cumulative probability and inverse cumulative probability functions respectively. If you want to perform a rank-signed test, the \(\text{CumWilcoxon}(u,m,n)\) function is the one to use, since it returns the p-value. Although they are built-in functions, the analytic functions aren’t listed on the Definition menu libraries until you add the Distribution Densities library to your model.

The \texttt{exact} parameter controls whether an exact algorithm is used or a Normal approximation. By default, a Normal approximation is used when \(m+n>100\), which has been observed empirically to be within 0.1% accuracy. Time and memory usage can be excessive when \(m+n\) gets large. When accuracy requirements are not demanding, efficiency can be gained by using \(\text{Wilcoxon}(m,n,\texttt{exact:m+n<50})\), or \(\text{CumWilcoxon}(m,n,\texttt{exact:m+n<50})\), etc.

Probability density and mass graphs

When you select the Probability density as the uncertainty view (page 29) 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.

![Probability Mass of Poisson(5)](image)

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.

![Cumulative Probability of Poisson(5)](image)

**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:

\[
\text{Chance Indiscrete} := \text{Poisson}(4) \times 0.5
\]

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 266), 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: Each cell of the result is a real number. Lower and upper bounds may be optionally specified. There is also a semi-continuous variation in which the value must be between the indicated bounds or it may be zero. The semi-continuous variation must be entered as an Expression, e.g., Continuous(100K,1M,orZero:True). When defined as continuous, uncertain variables are treated as continuous, whereas the other types are discrete.

Integer: Each cell of the result is an integer, with lower and upper bounds optionally specified. Uncertain results are graphed as discrete numeric. The Optimizer searches for an integer solution.

Grouped Integer: (Optimizer edition only) All scalar decision variables belonging to the same group must have a value between 1 and n, where n is the number of cells belonging to the same group, and all the values of each must be different.

Boolean: Each cell of the result is 0 (False) or 1 (True) valued.

Discrete: Discrete but its values are unspecified. Influences how uncertain quantities are treated and in some cases, how graph axes are scaled by default.

Explicit Values: You specify a list of possible values. Within this option, you can also select whether these possible values are to be limited to be textual or numeric.

Copy From Index: You enter the name of an index variable, to use its values as the domain, or another variable to copy its domain values.

Expression: An arbitrary expression that computes the domain type, ranges and possible values. The expression should return with a call to one of the functions Continuous(), Integer(), GroupedInteger(), or Discrete(), or it can return a list of values. The expression can contain arbitrary logic and can use values from other variables in the model.
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 343).

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.

Protable(): 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 protable) (see “Arrays and Indexes” on page 149).

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 20) 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 ProTable button appears in the definition. Click it to open the Edit Table window (see “Defining a variable as an edit table” on page 176).
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.

![Indexes dialog](image)

**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.

![Edit Table](image)

**Expression view of probability table** The Weather probability table when viewed as an expression, looks like this.

\[ \text{Probtable}(\text{Self})\{P, (1-P)\} \]

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:

\[ \text{Probtable}(i1, i2, \ldots, in)\{p1, p2, p3, \ldots, pm\} \]

This example is an n-dimensional conditional probability table, indexed by the indexes i1, i2, \ldots, in. One index must be Self. p1, p2, p3, \ldots, pm are the probabilities in the array. m is the product of the sizes of the indexes i1, i2, \ldots, 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 267), 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 13, “More Array Functions,” on page 215, 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 216.

Example In “Create a probability table” on page 266, 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.
**Chapter 16  Probability Distributions**

**Custom discrete probabilities**

**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.
- You want to use other variables or expressions to define the outcomes or probability arrays.

**Library** Distribution

**Example**

Index_b:

<table>
<thead>
<tr>
<th></th>
<th>Red</th>
<th>White</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Array_q:

Index_b

<table>
<thead>
<tr>
<th></th>
<th>Red</th>
<th>White</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.3</td>
<td>0.2</td>
<td>0.5</td>
</tr>
</tbody>
</table>

The domain of the variable is a list of labels: ['Red', 'White', 'Blue'].

Chancedist(Array_q, Index_b, Index_b) →
Parametric continuous distributions

**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 262.

**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) →

![Uniform distribution example graph](image)

**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) →

![Triangular distribution example graph](image)
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 $[mean-stdev, mean+stdev]$ 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 geo-
Parametric continuous distributions

metric standard deviation, gsdev, must be 1 or greater. It is sometimes also known as the uncertainty factor or error factor. The range \([\text{median}/\text{gsdev}, \text{median} \times \text{gsdev}]\) encloses about 68% of the probability — just like the range \([\text{mean} - \text{stddev}, \text{mean} + \text{stddev}]\) 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 \(\ln(\text{median})\) and standard deviation \(\ln(\text{gsdev})\).

**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)

\[
\text{Beta}(x, y, \text{min}, \text{max})
\]

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:

\[
\frac{x}{x + y} \times (\text{max} - \text{min}) + \text{min}
\]

**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:

\[
\text{Uniform}(0, 1)
\]

Or equivalently:

\[
\text{Beta}(1, 1)
\]

If you observe \(r\) heads in \(n\) tosses of the coin, your new (posterior) opinion about \(h\), should be:

\[
\text{Beta}(1 + r, 1 + n - r)
\]
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 $\text{Beta}(5, 10) \rightarrow$

$\text{Beta}(5, 10, 2, 4) \rightarrow$

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:

$$\frac{\pi^2}{3} \times \frac{s^2}{3}$$

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:

$$\text{Logit}(p(x)) = \ln(p(x) / (1-p(x))) = \frac{x}{s} - \frac{m}{s}$$

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 $m>0$:

$$\text{GetFract(}\text{StudentT}(n-1), 0.95)$$

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 $\text{StudentT}(8)$

![Graph of StudentT distribution]

Weibull($n$, $s$)

The Weibull distribution has a cumulative density given by:

$$f(x) = \frac{(x/n)^{n-1}}{s^n} e^{-x/n}$$

for $x \geq 0$. It is similar in shape to the gamma distribution, but tends to be less skewed and tail-heavy. It is often used to represent failure time in reliability models. In such models, $f(x)$ 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 $\text{Weibull}(10, 4)$ →

![Graph of Weibull distribution]
ChiSquared(d)

The ChiSquared() distribution with d degrees of freedom describes the distribution of a Chi-Squared metric defined as:

\[ \chi^2 = \sum_{i=1}^{n} y_i^2 \]

where each \( y_i \) is independently sampled from a standard normal distribution and \( d = n - 1 \). The distribution is defined over non-negative values.

The Chi-squared distribution is commonly used for analyses of second moments, such as analyses of variance and contingency table analyses. It can also be used to generate the F distribution. Suppose:

\[
\begin{align*}
\text{Variable } V & := \text{ChiSquared}(k) \\
\text{Variable } W & := \text{ChiSquared}(m) \\
\text{Variable } S & := \frac{V}{k} \cdot \frac{W}{m}
\end{align*}
\]

\( S \) 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 non-decreasing and should start with 0 and end with 1. The values of \( r \) must also be non-decreasing 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.

<table>
<thead>
<tr>
<th>Library</th>
<th>Distribution</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array_b</td>
<td>Index_a</td>
<td></td>
</tr>
<tr>
<td>1 2 3</td>
<td>0 0.6 1.0</td>
<td></td>
</tr>
<tr>
<td>Array_x</td>
<td>Index_a</td>
<td></td>
</tr>
<tr>
<td>1 2 3</td>
<td>10 20 30</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{CumDist(Array_b, Array_x)} \]
### 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 $2r_1 - r_i$ (or $2r_n - r_{n-1}$).

Either $r$ must be an index of $p$, or $p$ and $r$ must have an index in common. If $p$ or $r$ have more than one index, you must specify the index $i$ to link $p$ and $r$.

It produces the density function using linear interpolation between the points on the density function (quadratic on CDF).

#### Library

**Distribution**

<table>
<thead>
<tr>
<th>p</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0.4 0.2 0.5 0.2 0</td>
<td></td>
</tr>
</tbody>
</table>

| 10 15 20 25 30 35 |

```plaintext
Probdist(p, r, i)
```
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 122) 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:

\[
\begin{align*}
\text{Chance } X & := \text{Normal}(10, 2) \\
\text{Chance } A & := \text{Truncate}(X, 7) \\
\text{Chance } B & := \text{Truncate}(X, 10) \\
\text{Chance } C & := \text{Truncate}(X, 8, 12) \\
\text{Variable } \text{Compare}_{\text{truncated }} x & := [A, B, C]
\end{align*}
\]

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:

\[
\text{Random(Uniform(-100, 100))}
\]

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:

\[
\begin{align*}
\text{Random(Uniform(-100, 100))} & \rightarrow 74.4213148 \\
\text{Random()} & \rightarrow 0.265569265
\end{align*}
\]

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:

\[
\text{Random(Over: I)}
\]
returns an array of independent uniform random numbers between 0 and 1 indexed by i. It is equivalent to:

\[
\text{Random(Uniform(0, 1, Over: i))}
\]

**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, NegativeBinomial, 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, 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:

\[
\text{singleSampleMethod: Optional Atom Number}
\]

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:

\[
\text{Normal(10, 2, Over: K)}
\]

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:

\[
\text{Uniform(0, 10, Over: I, J, K)}
\]

**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:

\[
\begin{align*}
\text{Index K := ['A', 'B', 'C']} \\
\text{Variable Xmean := Table(K)(10, 11, 12)} \\
\text{Variable X := Normal(Xmean, 2)}
\end{align*}
\]
Chapter 16  Probability Distributions

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 346.

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:

\[
\text{Correlate_with(LogNormal}(2, 3), Z, 0.8)
\]

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:

\[ \text{Rank} \left( \text{Sample}(x), \text{Run} \right) = \text{Rank} \left( \text{Sample}(\text{Dist}_\text{reshape}(x, y)), \text{Run} \right) \]

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:

\[ \text{RankCorrel}(x, z) = \text{RankCorrel}(\text{Dist}_\text{reshape}(x, y), z) \]

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:

\[ k \cdot \text{Product}(x^{\alpha-1}, i) \]

where k is a normalization factor equal to:

\[ \frac{\text{GammaFn}(\text{Sum}(\alpha, i))}{\text{Sum}(\text{GammaFn}(\alpha), i)} \]
The \textbf{alpha} parameters can be interpreted as observation counts. The mean is given by the relative values of \textbf{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 \textbf{alpha} = 0, and you have \textbf{n} observations.

\textbf{Multinomial(n, theta, i)}

Returns the multinomial distribution, a generalization of the binomial distribution to \textbf{n} possible outcomes. For example, if you were to roll a fair die \textbf{n} times, the outcome would be the number of times each of the six numbers appears. \textbf{theta} would be the probability of each outcome, where \textbf{Sum(theta, i)}=1, and index \textbf{i} is the list of possible outcomes. If \textbf{theta} doesn’t sum to 1, it is normalized.

Each sample is a vector indexed by \textbf{i} indicating the number of times the corresponding outcome (die number) occurred during that sample point. Each sample has the property

\[
\text{Sum(result, I)} = \text{n}
\]

\textbf{UniformSpherical(i, r)}

Generates points uniformly on a sphere (or circle or hypersphere). Each sample generated is indexed by \textbf{i}, so if \textbf{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.

\textbf{MultiUniform(cm, i, j, lb, ub)}

The multi-variate uniform distribution.

Generates vector samples (indexed by \textbf{i}) such that each component has a uniform marginal distribution, and each component has the pair-wise correlation matrix \textbf{cm}, indexed by \textbf{i} and \textbf{j}, which must have the same number of elements. \textbf{cm} needs to be symmetric and must obey a certain semidefinite condition, namely that the transformed matrix \([2\sin(30^\circ)\text{cov}]\) is positive semidefinite. (In most cases, this roughly the same as \textbf{cm} being positive semidefinite.) \textbf{lb} and \textbf{ub} can be used to specify upper and lower bounds, either for all components, or individually if these bounds are indexed by \textbf{i}. If \textbf{lb} and \textbf{ub} are omitted, each component has marginal \textbf{Uniform(0, 1)}.

\textit{Note: cm is the true sample correlation, not rank correlation.}

The transformation is based on:


\textbf{Arrays with serial correlation}

These six functions each generate an array of distributions over an index \textbf{t} such that each distribution has a specified serial correlation with the preceding element over \textbf{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. \textbf{Normal_serial_correl()} and \textbf{Dist_serial_correl()} generate arrays of serially correlated distributions that are normal and arbitrary, respectively. \textbf{Normal_additive_gro()} and \textbf{Dist_additive_growth()} produce arrays with uncertain additive growth with serial correlation. \textbf{Normal-compound_gro()} and \textbf{Dist-compound_growth()} produce arrays with uncertain compound growth with serial correlation.
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:

\[
f(x) / ((p f(x) + (1 - p) cr(x)) \tag{3}
\]

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:

\[
\text{Chance Demand} := \text{Lognormal}(100, 1.5) \\
\text{Decision Capacity} := 240 \\
\text{Probability(Demand) } \rightarrow 0.015
\]

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:

\[
\text{Variable Price} := 100 \\
\text{Variable Penalty} := 1000 \\
\text{Variable Revenue} := \text{IF Demand <= Capacity THEN Price*Demand ELSE Price*Capacity - (Demand - Capacity)*Penalty} \\
\text{Mean (Revenue) } \rightarrow \$2309
\]
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 296.
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.
- Functions to fit generalized linear models.
- Functions to encode Stochastic Information Packets (SIPs) for integration with a Probability Management infrastructure.
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 29) available in the Result window use these functions.

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:

```plaintext
Chance X := Normal(0, 1)
Variable X90 := Getfract(X, .9)
X90 \rightarrow 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:

```plaintext
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 416 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 291)
- $s$ Standard deviation (see “Sdeviation(x)” on page 291)
- $m$ Sample size (see “Appendix A: Selecting the Sample Size” on page 416)

**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:

```plaintext
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)
```
Chapter 17  Statistics, Sensitivity, and Uncertainty Analysis

Mean(x)

Returns an estimate of the mean of x if x is probabilistic. Otherwise, returns x.

\[ \frac{1}{m} \sum_{i=1}^{m} x_i = \bar{x} \]

**Library**  Statistical

**Examples**  
\[ \text{Mean(Fuel_price)} \rightarrow 1.19 \]
\[ \text{Mean(Skfuel_price)} \rightarrow 1.33 \]

Median(x)

Returns an estimate of the median of x from its sample if x is probabilistic. When x is non-probabilistic, returns x. Equivalent to \text{GetFract(x, 0.5)}.

**Library**  Statistical

**Examples**  
\[ \text{Median(Fuel_price)} \rightarrow 1.19 \]

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.

\[ \sqrt{\frac{1}{m-1} \sum_{i=1}^{m} (x_i - \bar{x})^2} = \sigma \]

**Library**  Statistical

**Example**  
\[ \text{Sdeviation(Fuel_price)} \rightarrow 0.10 \]

Variance(x)

Returns an estimate of the variance of x if x is probabilistic. If x is non-probabilistic, returns 0.

\[ \frac{1}{m-1} \sum_{i=1}^{m} (x_i - \bar{x})^2 = \sigma^2 \]

**Library**  Statistical

**Example**  
\[ \text{Variance(Fuel_price)} \rightarrow 0.01 \]

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.

\[ \frac{1}{m} \sum_{i=1}^{m} \left( \frac{x_i - \bar{x}}{\sigma} \right)^3 \]
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.

\[
\text{Kurtosis}(x) = \left( \frac{1}{m} \sum_{i=1}^{m} \left( \frac{x_i - \overline{x}}{\overline{s}} \right)^4 \right) - 3
\]

Probability(b)

Returns an estimate of the probability or array of probabilities that the Boolean value b is True.

GetFract(x, p, I)

Returns an estimate of the pth fractile (also known as quantile or percentile) of x over index I. The index I is optional. If it is omitted, the function operates over the Run index and returns probability fractiles. This is the value of x such that x has a probability p of being less than that value. If x is constant over index I—for example, a non-probabilistic variable using Run as the fractile index→, all fractiles are equal to x.

The value of p must be a number or array of numbers between 0 and 1, inclusive.

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 251), Probability Bands option.
## 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** $j := CopyIndex(I)$
- **Library** Statistical
- **Function** 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 $\text{Variance}(x) = \text{Covariance}(x, x)$. 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:

$$
\frac{\sum_{i} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i} (x_i - \bar{x})^2} \times \sqrt{\sum_{i} (y_i - \bar{y})^2}}
$$

### Library Statistical

**Example** With `sampleSize` set to 100 and number format set to two decimal digits:

- Correlation($\text{Alt\_fuel\_price} + \text{Fuel\_price}$, $\text{Fuel\_price}$) → 0.71

Correlation of two independent, uncorrelated distributions approaches 0 as the sample size approaches infinity.

**Example** With `sampleSize = 20`:

- Correlation($\text{Normal}(1.19, 0.1)$, $\text{Normal}(1.19, 0.1)$) → -.28

With `sampleSize = 1000`:

- Correlation($\text{Normal}(1.19, 0.1)$, $\text{Normal}(1.19, 0.1)$) → 0.03
Chapter 17  Statistics, Sensitivity, and Uncertainty Analysis

**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 297).

**Library**  Statistical

**Example**  With `sampleSize = 100`:

\[
\text{Rankcorrel(Fuel\_price, Alt\_fuel\_price)} \rightarrow .02
\]

**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 251). If you want the relative frequency rather than the count of each value, divide the result by `sampleSize`.

**Library**  Statistical

**Example**  (continuous)

\[
\text{Index Index\_a := \{1.2,1.25\}}
\]

\[
\text{Frequency(Fuel\_price, Index\_a)} \rightarrow
\]

\[
\begin{array}{|c|c|}
\hline
1.2 & 1.25 \\
\hline
54 & 19 \\
\hline
\end{array}
\]

**Example**  (discrete)

`Bern\_out: [0,1]`

(Possible outcomes of the Bernoulli Distribution.)

With `Samplesize = 100`:

\[
\text{Frequency(Bernoulli (0.3), Bern\_out)} \rightarrow
\]

\[
\begin{array}{|c|c|}
\hline
0 & 1 \\
\hline
70 & 30 \\
\hline
\end{array}
\]

With `Samplesize = 25`:

\[
\text{Frequency(Bernoulli (0.3), Bern\_out)} \rightarrow
\]

\[
\begin{array}{|c|c|}
\hline
0 & 1 \\
\hline
18 & 7 \\
\hline
\end{array}
\]

(Compare to the Bernoulli example on page 261.)
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 251).

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) ▶

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.191</td>
<td>1.32</td>
<td>1.19</td>
<td>1.164</td>
<td>1.191</td>
<td>0.962</td>
</tr>
</tbody>
</table>

Statistics(x)

Returns an array of statistics of x. Select the statistics to display in the Uncertainty Setup dialog (page 251), Statistics option.

Library Statistical

Example

Statistics(Fuel_price) →

<table>
<thead>
<tr>
<th>Min</th>
<th>Median</th>
<th>Mean</th>
<th>Max</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.93</td>
<td>1.19</td>
<td>1.19</td>
<td>1.45</td>
<td>0.10</td>
</tr>
</tbody>
</table>

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 265). 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.

spacingMethod  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.

smoothingMethod  Selects the method for estimating the continuous density, Pdf(...) only.

- 0 : Histogramming.
- 1 : Kernel Density Smoothing.
- 2 : Kernel Density Smoothing with given bandwidth.

smoothingFactor  Pdf(...) function only. When smoothingMethod is 1 or 2, this determines the degree of smoothing. For smoothingMethod:1, smoothingFactor should be a value between -1 and 1 indicating the desired smoothing relative to what Pdf(...) determines to be the optimal bandwidth. A value of 0 indicates that the optimal bandwidth should be used, negative values for more detail, positive values for more smoothing.

When smoothingMethod:2 is used, Pdf(...) performs a Fast Gaussian Transform (FGT) using the positive value to smoothingFactor as the Gaussian bandwidth.

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:

```
X
```
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:

\[
\text{Mean}(Y, I, W: Y>0)
\]

If you set the system variable SampleWeighting to something other than 1 (see “Importance weighting” on page 285, 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 \(U_{\text{Inputs}}\), a list of the uncertain inputs, and a general variable, \(U_{\text{Importance}}\), 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.
**Chapter 17: Statistics, Sensitivity, and Uncertainty Analysis**

### Importance Analysis

**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.

**Fuel cost per year Importance** is defined as:

\[
\text{Abs} \left( \text{Rankcorrel}(\text{Fuel\_cost\_per\_year\_inputs}, \text{Fuel\_cost\_per\_year}) \right)
\]

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, **Fuel cost 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.

![Probability Density Plot of fuelCost](image)
**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:

\[
\text{Dydx(fuelCost, mpg) } \rightarrow -19.7
\]

The reverse is not true, because mpg is not dependent on fuelCost. That is, fuelCost does not cause any change in mpg:

\[
\text{Dydx(Mpg, Fuelcost) } \rightarrow 0
\]

In this model of fuelCost, a small change in gasPrice has by far the largest effect of all its inputs:

\[
\text{Dydx(fuelCost, gasPrice) } \rightarrow 428.6
\]

\[
\text{Dydx(fuelCost, mpy) } \rightarrow 0.04643
\]

**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:

\[
Y := \text{DyDx(Kurtosis(Normal(0, X)), X)}
\]

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:

\[
Y := \text{DyDx(Kurtosis(Normal(0, Mid(X))), X)}
\]

**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:

\[
\text{Elasticity}(y, x) = \lim_{u \to 0} \frac{1}{u} \frac{y(x(1+u)) - y(x)}{y(x)} \tag{3}
\]

When x is a positive scalar, but not when x is array-valued, Elasticity() is related to Dydx() in the following manner:

\[
\text{Elasticity}(y, x) = \text{Dydx}(y, x) \times (x/y)
\]

**Library** Special

**Examples** Elasticity(fuelCost, mpg) \(\rightarrow -1\)

Elasticity(fuelCost, gasprice) \(\rightarrow 1\)

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 vNew 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%.
Tornado charts

3. Create a node, Tornado_Analysis. Assume that the output variable is Net_value. Define Tornado_Analysis as:
   
   \[ \text{WhatIfAll}(X, Vars, \text{Level} * Vars) \]

4. Create a node, Input_Vars, defined as:
   
   \[ \text{sortIndex}(-\text{abs}(\text{Tornado_Analysis[Level='high']} - \text{Tornado_Analysis[Level='low']})) \]

5. Create a node, Net_value_range, to hold the final graph, defined as:
   
   \[ \text{Tornado_Analysis[Vars=Input_Vars]} \]

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 \text{WhatIfAll()} function typically provides the easiest method for implementing a tornado analysis in Analytica. Note that the third parameter to \text{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 \text{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 \text{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 \textit{Net\_Value}. Click the \textit{XY} button to open the \textit{XY Comparison Sources} dialog, check \textit{Use another variable}, press \textit{Add...}, and in the \textit{Object Finder} select the variable \textit{Net\_Value}. Press OK twice.

6. In the Bar Origin pulldown, select \textit{Net\_value}.

\section*{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 \textit{Mid}, \textit{Mean}, \textit{Statistics}, \textit{Probability Bands}, and \textit{Sample} view modes.

To graph one variable against another:

1. Open a \textbf{Result} window for the $y$- (vertical axis) variable.
2. Click the \textit{XY} button located in the top-right corner of the window to open the \textit{XY Comparison Sources} dialog.
3. Check the \textit{Use another variable} checkbox, press \textit{Add...}, and in the \textit{Object Finder}, select the $x$- (horizontal axis) variable.

The two variables in an \textit{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 \textbf{Common Index} — only indexes of both $x$ and $y$ might be selected.

\begin{verbatim}
Variable Angle := Sequence(0, 360, 10)
Variable Radius := 1..3
Variable SinX := Radius * Sin(Angle)
Variable Cosine := Radius * Cos(Angle)
\end{verbatim}
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.

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 297), 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\( (y, b, i, k) \)

Generalized linear regression. Finds the best-fit (least squared error) curve to a set of data points. Regression() finds the parameters \( a_k \) in an equation of the form:

\[
y = \sum_k a_k b_k(\hat{x})
\]

The data points are contained in \( y \) (the dependent variable) and \( b \) (the independent variables), both of which must be indexed by \( i \). \( b \) is the basis set and is indexed by \( i \) and \( k \). The function returns the set of parameters \( a_k \) indexed by \( k \). Any datapoint having \( y=\text{Null} \) 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:

\[
\text{Index } K := [\text{"m", "b"}]
\]

Next, define \( b \) as a table indexed by \( k \):

```
Variable b := k

<table>
<thead>
<tr>
<th>m</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Chapter 17 Statistics, Sensitivity, and Uncertainty Analysis

Uncertainty in regression results

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 b 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:

\[ y = \text{Sum}(c*b, k) + \text{Normal}(0, s) \]

s is the measurement noise. You have the data b[i, k] and y[i]. You might or might not know the measurement noise s. So you perform a linear regression to obtain an estimate of c. Because your estimate is obtained from a finite amount of data, your estimate of c is itself uncertain. This function returns the coefficients c as a distribution (i.e., in sample mode, it returns a sampling of coefficients indexed by Run and k), reflecting the uncertainty in the estimation of these parameters.

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:

$$Y = \sum(c*b, k) + \text{Normal}(0, s)$$

If this result is very close to zero, it probably indicates that the assumption of linearity is bad. If it is very close to one, then it validates the assumption of linearity.

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:

$$y = \sum(c*b, i) + \text{Normal}(0, s)$$

This function computes an estimate for $s$ by assuming that the sample noise is the same for each point in the data set.

When using in conjunction with RegressionDist, it is most efficient to provide the optional parameter $c$ to both routines, where $c$ is the expected value of the regression coefficients, obtained from calling Regression($y, b, i, k$). Doing so avoids an unnecessary call to the built-in Regression function.

Logistic regression functions

You can use the functions in this section to estimate the probability (or probability distribution) of a binary or categorical dependent (output) variable as a function of known values for independent (input) variables. This is similar to linear regression, which predicts the value of a dependent variable as a function of known values for independent variables. Logistic regression is the best known example generalized regression, so even though the term logistic regression technically refers to one specific form of generalized regression (with probit and poisson regression being other instances), it is also not uncommon to hear the term logistic regression functions used synonymously with generalized linear regression, as we have done with the title of this section.

The functions LogisticRegression and ProbitRegression predict the probability of a Bernoulli (i.e., 0,1-valued) random variable from a set of continuous independent variables. Both functions apply to the same scenarios and accept identical parameters; the final models differ slightly in their functional form. The function PoissonRegression predicts the probability distribution for the number of events that occur, where the dependent (output) variable is a non-negative integer.
All three functions accept the same parameters as the **Regression** function, described on page 307. As with those functions, you construct a basis for your dependent variables, and will usually want to include the constant term (a 1 in the basis). In addition to those parameters, these functions also have two parameters, **priorType** and **priorDev**, which allow you to specify a Bayesian prior.

**Bayesian priors**

The regression methods in this section are highly susceptible to overfitting. The problem is particularly bad when there are a small number of data points or a large number of basis terms. When your model has been overfit, it will produce probability estimates that are too close to zero or one; in other words, its predictions are overconfident. To avoid overfitting, you will usually want to employ a Bayesian prior, which you do by specifying the **priorType** parameter, which recognizes these options:

- 0 = Maximum likelihood (default)
- 1 = Exponential L1 prior
- 2 = Normal L2 prior

Maximum likelihood corresponds to having no prior. The L1 and L2 priors penalize larger coefficient weights. The joint prior probability of each coefficient is statistically independent, having the shape of a decaying exponential function in the case of an L1 prior or of a half-normal distribution in the case of the L2 prior.

You can also optionally specify the strength of the prior using the **priorDev** parameter, which specifies the standard deviation of each marginal prior distribution on each coefficient. Cross validation techniques vary this parameter to find the optimal prior strength for a given problem, which is demonstrated in the "Logistic Regression prior selection.ana" example model included with Analytica in the Data Analysis example models folder. If you omit the **priorDev** parameter, the function makes a reasonable guess, which will usually be superior to maximum likelihood.

**LogisticRegression(y,b,i,k,priorType,priorDev)**

**Logistic regression** is a technique for predicting a Bernoulli (i.e., 0,1-valued) random variable from a set of continuous dependent variables. See the Wikipedia article on logistic regression (http://en.wikipedia.org/wiki/Logistic_regression) for a simple description. The **LogisticRegression** function finds the parameters $c_k$ that fit a model of the form

$$\text{Logit}(p(x)) = \sum_k c_k b_k(x)$$  \hspace{1cm} (7)$$

where $p(y)$ is the probability of outcome $y$, and $b_k(x)$ is the basis vector for a data which is indexed by $k$. To understand how to put together a basis from your independent variables, you should read the section on the **Regression** function on page 307. It is exactly the same here. Notice that the righthand side of Equation (7) is the same as for standard **Regression** (Equation (6)), but the lefthand side involves the **Logit** function (see page 242). The inverse of the **Logit** function is the **Sigmoid** function, so that once you've obtained the result from **LogisticRegression**, you can use it to predict the probability for a new data point using

$$\text{Sigmoid}(\text{Sum}(c*B(x), k))$$

where B(x) is a user-defined function that returns the basis vector for the data point.

**Example**

Suppose you have Height, Weight and Gender information for a set of people, with these three variables indexed by Person. A logistic regression model might estimate the probability that a given person is male based on height and weight, encoded as follows:

```plaintext
Index K := ['b','height','weight']
Function PersonBasis(height,weight) :=
   Array(K,[1,height,weight])
Variable coef :=
   LogisticRegression(Gender='M', PersonBasis(Height,Weight),
```

```plaintext
```
Person, K, priorType:2)

With these coefficients, the probability that a 85kg, 170cm tall person is male is

\[
\text{Sigmoid}\left( \sum \text{coef*PersonBasis}(170,85), k \right)
\]

**ProbitRegression**\((y,b,i,k,\text{priorType},\text{priorDev})\)

A probit model relates a continuous vector of dependent measurements to the probability of a Bernoulli (i.e., 0,1-valued) outcome. In econometrics, this model is sometimes called the Harvard model. The ProbitRegression function finds the parameters \(c_k\) that fit a model of the form

\[
\text{CumNormal}(p(x)) = \sum c_k b_k(x)
\]

where \(p(y)\) is the probability of outcome \(y\), and \(b_k(x)\) is the basis vector for a data which is indexed by \(k\). To understand how to put together a basis from your independent variables, you should read the section on the Regression function on page 307, it is exactly the same here. Notice that the righthand side of Equation (7) is the same as for standard Regression (Equation (6)), but the lefthand side involves the CumNormal function (see page 241). Once you’ve obtained the result from LogisticRegression, you can use it to predict the probability for a new data point using

\[
\text{CumNormalInv}\left( \sum (c*B(x), k) \right)
\]

where \(B(x)\) is a user-defined function that returns the basis vector for the data point.

**Example**

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

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

\[
\text{Variable c := ProbRegression( Treatment_effective, Test_results, Patient_ID, Lab_test, priorType:2)}
\]

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

\[
\text{Variable Prob_Effective := CumNormalInv( Sum( c*Test_results, Lab_test ))}
\]

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

\[
\text{CumNormalInv( Sum( c*New_patient_tests, Lab_test))}
\]

**PoissonRegression**\((y,b,i,k,\text{priorType},\text{priorDev})\)

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

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

The random component in the prediction is assumed to be Poisson-distributed, so that given a new data point \( b \), the distribution for that point is

\[
\text{Poisson}(\exp(\sum(c \cdot b, K)))
\]

**Example**

You have data collected from surveys on how many times TV viewers were exposed to your ads in a given week, and on how many times you ran ads in each time slot on those weeks. You want to fit a model to this data so that you can predict the distribution of exposures that you can expect in the future for a given allocation of ads to each time slot.

Each data point used for training is one survey response (from one person) taken at the end of one particular week (\( \text{Training\_exposures} \) indexed by \( \text{Survey\_response} \)). The basis includes a constant term plus the number of times ads were run in each time slot that week (\( \text{Training\_basis} \) indexed by \( \text{Time\_slot\_K} \) and \( \text{Survey\_response} \)).

Index \( \text{Time\_Slot\_K} := [1, 'Prime time', 'Late night', 'Day time'] \)

Variable \( \text{exposure\_coeffs} := \text{PoissonRegression}( \text{Training\_exposures}, \text{Training\_basis}, \text{Survey\_response}, \text{Time\_slot\_K}) \)

To estimate the distribution for how many times a viewer will be exposed to your ads next week if you run 30 ads in prime time, 20 in late night and 50 during the day, use

Decision \( \text{Ad\_Allocation} := \text{Table}(\text{Time\_slot\_K})(1, 30, 20, 50) \)

Chance \( \text{Viewers\_Exposed} := \text{Poisson}(\exp(\sum(\text{Exposure\_coeffs} \cdot \text{Ad\_Allocation}, \text{Time\_slot\_K}))) \)

This example can be found in the Example Models / Data Analysis folder in the model file "Poisson regression ad exposures.ana".

---

**Stochastic Information Packets (SIPs)**

Any arbitrary probability distribution can be approximately represented by its Monte Carlo sample. When a consistent ordering of the Monte Carlo samples is maintained along the common run index, the representation also captures complex correlation relationships relative to other variables. ProbabilityManagement.org advocates a corporate framework in which organizations maintain libraries of official distributions for key uncertain quantities, which can be exchanged between model builders so that all models rely on common assumptions. Each univariate Monte Carlo sample is collected into a unit called a Stochastic Information Packet (SIP), which is a textual XML snippet containing a compressed representation of the Monte Carlo sample. The DIST\textsuperscript{TM} standard specifies a unified representation for SIPs which is used by several products in addition to Analytica. Analytica’s \text{SipEncode()} and \text{SipDecode()} function convert Monte Carlo samples to the DIST\textsuperscript{TM} standard representation for a SIP. When additional indexes are present, the result is an array of SIPs, which is termed a Stochastic Library Unit with Relationships Preserved (SLURP).


**SipEncode(x, i, name, type, origin, version)**

Encodes a Stochastic Information Packet (SIP) in accordance with the DIST\textsuperscript{TM} standard as a textual XML snippet. \( x \) is an uncertain quantity with a Monte Carlo sample indexed by \( i \). The index \( i \) is typically not specified and defaults to run. The optional \text{name} parameter is a textual identifier that is embedded in the XML to identify what the quantity represents. The \text{type} parameter specifies how the distribution should be compressed, with these possible values:
• ‘Single’: A lossy compression that uses about 1 1/3 character per number. Stores the values using 256 distinct bins.
• ‘Double’: A slightly lossy compression that uses about 2 2/3 characters per bin, with 65536 distinct bins. The precision is substantially better than with ‘Single’.
• ‘Binary’: Encodes a Bernoulli (true/false, 0/1) quantity, losslessly, encoding 6 binary values per character.

Use the optional origin parameter to include citation information to indicate the source of the distribution, when available. The optional version parameter can be set to 1 or 1.1 to specify the DIST™ 1.0 or DIST™ 1.1 standard. The 1.1 standard is the default.

\[
\text{SipEncode( Normal(0,1), type:'Single') } \rightarrow \text{SipDecode(xml,resultIndex,localIndexName)}
\]

Decodes a textual Stochastic Information Packet (SIP), encoded according to the DIST™ standard, into a Monte Carlo sample indexed by the system index Run when neither resultIndex nor localIndexName is specified. When an index is specified for resultIndex, the index is used to index the result. When localIndexName is specified (as text), then a local index is created with the same length as the original sample and used to index the result. It is an error to specify both resultIndex and localIndexName.

\[
\text{SipDecode(SipEncode(Normal(0,1),type:'Single')) } \rightarrow
\]

**Note:** This plot was produced by selecting the Probability Density result view and using a standard line for the graph line style. Although the compression and decoding is lossy, no distortion is perceivable to the naked eye.
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 — with a recurrence or dependence on previous time periods. The system function Dynamic() and system variable Time enable you to model changes over time.

**Tip**
Read Chapter 12, “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.

![Object - Time](image)

**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().

**Tip**
A variation, Dynamic[T]() can be used to represent recurrences over indexes other than Time, but placing the index name in square brackets. This provides a way to express secondary recurrences if you’ve already used your Time index for sometime else. The dynamic concepts are introduced thoroughly in this chapter using Time, but if you have a loop using a different index, just substitute your other index for Time in what follows.

### 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. There are only two functions in Analytica that allow a cycle to be formed, in which a variable can refer to its own value or to other variables that depend on it. Those two functions are Dynamic() and Iterate(). Only dynamic variables can compute their value based on the values 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).

---

**Object - Gasoline price**

- **Variable**: Gasprice
  - **Units**: $/gallon

- **Title**: Gasoline price

- **Description**: The price of a gallon of gasoline, over time

- **Definition**: Dynamic(1.2, Gasprice[Time-1] * 1.05)

- **Inputs**: Time, Gasprice

- **Outputs**: Gasprice
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 \([\text{Time-1}]\) by 1.05 (the 5 percent inflation rate).

\[ x \ [\text{Time-k}] \]

Given a variable \(x\) and brackets enclosing \(\text{Time}\) 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**

\(\text{Time-k}\) in the expression \(\text{var[Time-k]}\) refers to the position of the elements in the **Time** index, not values of **Time**.

For example, if \(\text{Time}\) 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 \((\text{Time-3})\) 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 170 and “Defining a variable as an edit table” on page 176). 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**

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 317 but allows for changes in time period increments.

```plaintext
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

```
Gasprice := Dynamic(1.2, Gasprice[Time-1] * 1.05)
```

![Graph showing the time series for Gasprice with labels at equal intervals.](image)
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 51).

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. It is typically the case that at least 1 initial value is supplied.

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 316), 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 74).
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:

$$\text{Dynamic}(0, \text{self}[Time-1] + \text{Max}(A, Time))$$

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 $Time$, $\text{Max}(A, Time)$ simply returns $A$, so that the above expression is equivalent to:

$$\text{Dynamic}(0, \text{self}[Time-1] + A)$$

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 $\text{Max}(A, Time)$, 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:

```plaintext
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`:

```plaintext
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:

```plaintext
Birthrate := Dynamic(Normal(1.2, .3), Normal(1.2, .3))
Pop_by_year := Dynamic(Population, Self[Time-1] + Birthrate)
```

**Dynamic on non-Time Indexes**

The `Dynamic` function can be used to express a recurrence along any index, where the value at one value depends on previous values, the same way it is used to express a recurrence along the `Time` index. For non-`Time` indexes, you must specify the index explicitly in square brackets following the function name.
In the following example, we are to acquire as much of item 1 as we can afford, but can only purchase an integer number of units. With whatever funds remain, we purchase as many integer units of item 2 as possible, as so on.

\[
\begin{align*}
\text{Index Item} & := [1,2,3,4] \\
\text{Unit_price} & := \text{Table(Item)}(5100,1600,800,250) \\
\text{Budget} & := 20K \\
\text{Num_acquired} & := \text{Floor}(\text{Funds_remaining}/\text{Unit_price}) \\
\text{Spent} & := \text{Num_acquired} \times \text{Unit_price} \\
\text{Funds_remaining} & := \text{Dynamic}[\text{Item}](\text{budget}, \text{Self[Item-1]}-\text{Spent[Item-1]})
\end{align*}
\]

The three variables, \text{Num_acquired}, \text{Spent}, and \text{Funds_remaining}, form a recurrence over the \text{Item} index. Evaluation proceeds starting with the dynamic context \text{Item}=1. When \text{Num_acquired} is evaluated in this context, the \text{Unit_price} appearing in its definition evaluates to \text{Unit_price[Item=1]}, since the use of a variable in a definition refers to its value sliced by the current dynamic context. Evaluation proceeds by evaluating each variable in the loop as \text{Item} is incremented and retaining the results for each value of \text{Item}:

Analytica's dynamic evaluation is tuned maximum efficiency using the \text{Time} index; thus, it is best to use the system \text{Time} index as your model's primary time index, utilizing non-\text{Time} indexes only when a secondary recurrent dimension is required.

Multi-dimensional recurrences are possible when dynamic recurrences over different indexes intersect. Here any unspent funds in the previous example are folded over into the following time-period's budget, forming a recurrence over both \text{Time} and \text{Item}:

\[
\begin{align*}
\text{Funds_remaining} & := \\
& \quad \text{Dynamic}[\text{Item}](\text{avail_funds}, \text{Self[Item-1]}-\text{Spent[Item-1]}) \\

\text{Unspent_funds} & := \\
& \quad \text{Funds_remaining[Item=max(Item)]} - \text{Spent[Item=Max(item)]} \\
\text{Avail_funds} & := \text{Dynamic}(\text{budget}, \text{budget}+\text{Unspent_funds[Time-1]})
\end{align*}
\]

Variables participating in dynamic loops containing two (or more) dynamic indexes will have all the dynamic indexes in their results. For some of these variables, such as \text{Unspent_funds} above, the result does not vary along that index, but the extra index appears as a consequence of it participating in a dynamic loop over that index.
Chapter 18 Dynamic Simulation

Dynamic on non-Time Indexes
Chapter 19

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 138) 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 336).

### 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 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.
Using OLE to link results to other applications

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 Clothflow 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.
Using OLE links, 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.
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 into an edit table or list
To import a definition from a text file into an edit table or list:

1. Select the definition field of the variable in either the Object window or Attribute panel definition view. If variable is a table, open the edit table.
2. If you are importing a simple tab-delimited file, then select the cell(s) in which to import. If you are importing a data file in the format described in “Edit table data import/export format” on page 336, then the cell selection is not used.
3. Select 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 all the slices), a special text format is required (see “Edit table data import/export format” on page 336). 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.
- **View** is the name of the view from which the data originated. If your data was exported from an edit table, this is **EditTable**. If it was exported from a result table, it will be **Value**, **Mean**, **Statistics**, **Probbands**, **PDF**, **CDF** or **Sample**.
- **Variable identifier** is the identifier of the variable node where the data originated.
- **Index identifier** is the identifier of the index for this variable. An index with this name must already exist in the model, with the same length as in the file. When the index is a domain index, the **<Index Identifier>** is of the form **Domain of <identifier>**.
- Each index value and array value pair must be separated by tab characters.
  - Numeric values appear directly, in any of Analytica's standard number or date/time formats.
  - Text values are enclosed in either double (") or single (') quotes. Tab and newline characters cannot appear between the quotes. If the text between the quotes includes a tab or newline character, it must be escaped using one of the sequences: \n, \r, \t. If a backslash or quote character appears in the text, it too must be escaped using \, \' or \". (When the text is enclosed in double quotes, single quotes don’t have to be escaped, and vice versa).
  - Expressions (from edit table cells), which include distribution functions that might appear in edit table cells, appear without quotes as long as they single-line expressions (i.e., do not contain any newline characters), do not start and end with the same quote character (such as the expression "a" & "b"), and do not contain any tab character. Otherwise, the entire expression is enclosed between tildes, and the escape sequences \n, \r, \t, \ are used for newline, tab, backslash or tilde characters. For example, the following multi-line expression:

    ```
    var s := "--";
    JoinText(A,I,s) & '\'
    ```

    would appear as:

    ```
    ~var s:= "\--";
    \JoinText(A,I,s) & '\'
    ```

  - The value null may appear as «null».

**One-dimensional array**

The format for a one-dimensional array is:

```
TextTable <View> <Variable identifier> <line break>
[Index identifier]<line break>
<Array value><line break>
```

**Example**

```
Keyword   View   Variable identifier
TextTable Mean House_cost_inputs
House_inputs
'PropTax' 3400
'Tax rate' 0.44
'Maintenance' 4000
'Interest' 0.105
'Appreciation' 0.08
```

Index values

Array values
Two-dimensional array

The format for a two-dimensional array is:

```
TextTable <View><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

<table>
<thead>
<tr>
<th>Keyword</th>
<th>View</th>
<th>Variable identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Down_payment</td>
<td>Buying_price</td>
<td>20000 45000 60000</td>
</tr>
<tr>
<td>200000</td>
<td>180000</td>
<td>155000 140000</td>
</tr>
<tr>
<td>400000</td>
<td>380000</td>
<td>355000 340000</td>
</tr>
<tr>
<td>600000</td>
<td>580000</td>
<td>555000 540000</td>
</tr>
</tbody>
</table>

Three-dimensional array

The format for a three-dimensional array is:

```
TextTable <View> <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><tab><Index2 values separated by tabs><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

<table>
<thead>
<tr>
<th>Keyword</th>
<th>View</th>
<th>Variable identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextTable</td>
<td>EditTable</td>
<td>Net_diff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index1</th>
<th>Index1 Value1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buying_price</td>
<td>200000</td>
</tr>
<tr>
<td>Index2 values</td>
<td></td>
</tr>
<tr>
<td>Years_owned</td>
<td>5 10 15</td>
</tr>
<tr>
<td>Index3 values</td>
<td></td>
</tr>
<tr>
<td>Down_payment</td>
<td></td>
</tr>
<tr>
<td>20000</td>
<td>10112 12160 13525</td>
</tr>
<tr>
<td>45000</td>
<td>10093 12158 13540</td>
</tr>
<tr>
<td>60000</td>
<td>10073 12157 13555</td>
</tr>
</tbody>
</table>

| Array values |

<table>
<thead>
<tr>
<th>Index1 Value2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buying_price</td>
</tr>
<tr>
<td>Years_owned</td>
</tr>
<tr>
<td>Down_payment</td>
</tr>
<tr>
<td>20000</td>
</tr>
<tr>
<td>45000</td>
</tr>
<tr>
<td>65000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index1 Value3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buying_price</td>
</tr>
<tr>
<td>Years Owned</td>
</tr>
<tr>
<td>Down_payment</td>
</tr>
<tr>
<td>20000</td>
</tr>
<tr>
<td>45000</td>
</tr>
<tr>
<td>60000</td>
</tr>
</tbody>
</table>

Number format

Numerical data can be imported in any format recognized by Analytica (see “Number formats” on page 78).

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”.
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 the module path down to the current module:

You can jump to any parent or ancestor module by clicking on its name in the strip. When you click on an arrow, a tree menu displays other modules at that level and enables you to quickly navigate directly to any other module in the model:
Chapter 20: Working with Large Models

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 jump to a variable and open the **Diagram** containing it, double-click its entry in the outline.

To display a variable’s **Object** window, select it and press the **Object Window** button in the toolbar.

**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 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 15).

To view the attributes of a listed node:
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.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Function</th>
<th>Module</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author</td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Check</td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Class</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Created</td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Definition</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Description</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Domain</td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>File Info</td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Help</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Identifier</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Inputs</td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Last Saved</td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>MetaOnly</td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Outputs</td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Probvalue</td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Recursive</td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Title</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Units</td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Value</td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>User-created (up to 5)</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Key:
- plain = modifiable by user
- * = always displayed
- italic = set by Analytica
- + = optionally displayed

For descriptions of the attributes, see “Glossary” on page 439.

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.
Managing attributes

- **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 21, “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 53) 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.
Adding a module or library

To add a module or library, select Add Module (Control+l) 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.

4. Check the Read-only checkbox.
5. Close the Properties window.

Adding a module to a model
To add a field module to the active model, use the Add Module dialog (page 346). You can either embed a copy of the module or link to the original of the fielded module.

Adding library to a model
To add a fielded library to the active model, use the Add Module dialog (page 346). You can either embed a copy of the library or link to the original of the fielded library.

Removing a module or library from a model
To remove a fielded 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 fielded module or library, the Save command saves every fielded 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, fielded module or fielded 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.
The standard **Open Model** dialog appears. Select the desired module in that dialog. The following dialog then appears.

![Add a Module or Library dialog](image)

**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.

- **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 346) 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 345.

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 346), 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 334) 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 334) 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 347.

Created attributes When you combine models with created attributes, the maximum number of defined attributes is five (see “Managing attributes” on page 343).

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.
Large Memory Usage

The evaluation of some models may require large amounts of memory, especially when they contain arrays with high dimensionality. When the memory needed by your model approaches the available memory on your computer, you may encounter an error informing you that there is insufficient memory to carry out the computation, or you may experience a general slowdown as the process starts utilizing virtual memory. To address the challenges presented by memory bottlenecks, you may need to rethink various aspects of how you are attacking your problem, or you may have to employ various advanced modeling techniques to obtain the desired computations.

Memory limits

When you evaluate a model that requires more memory than your computer can provide, your experience is likely to be different depending on whether you are running the 32-bit or 64-bit edition of Analytica. Analytica 32-bit, like any 32-bit process, is limited to a maximum memory usage of 2GB, 3GB or 4GB, depending on the operating system. When this limit is reached, Analytica will report an insufficient memory error. With the 64-bit edition, which does not have this same limit, you are more likely to encounter a slowdown when the model’s memory requirements substantially exceed the amount of RAM on your computer. The model continues to evaluate, but the computer is slowed as it swaps pages of memory between a page file on your hard drive and RAM.

Configuring available memory

Various settings in the Windows operating system determine the maximum amount of memory that your process can use before running out of memory.

In 32-bit editions of Windows, processes are limited to 2GB of space unless you explicitly configure a flag in the c:\boot.ini file. With the /3GB flag in boot.ini, processes like Analytica may use up to 3GB (see How to access more memory on the Analytica Wiki).

Analytica 32-bit may be used from a 64-bit edition of Windows, in which case it can utilize 4GB with no special configuration required.

If you need to utilize more memory than the amount of RAM on your computer, you may need to adjust your virtual memory settings. On Windows, configure this from My Computer → Properties → Advanced system settings → Performance → Settings → Advanced → Virtual Memory (Change). You can substantially improve performance by installing a solid state hard drive (SSDI) and locating your page file on that drive. You should configure a custom size with an initial size that is large enough to accommodate your large computations. We find that Windows freezes for several minutes when it needs to increase

---

1. A 32-bit process runs in a process space where each byte is identified by its 32-bit address. With 32-bits, it is possible to have 2^{32} distinct locations, limiting the process to 4 billion bytes

2. The 2^{64} ceiling for a 64-bit process, which is about 18 million terabytes, will not be the limiting factor.

Your virtual memory settings on Windows determine the maximum memory available. Various editions of Windows also impose varying limits on how much memory can be accessed. For example, Windows 7 Professional limits processes to 192GB.
the size of its page file, which you avoid by using a suitably large initial size.

**Viewing memory usage**

You can monitor the total amount of memory in use by your Analytica process by using the Memory Usage dialog. See “Memory usage” on page 428.

When you find your model consuming excessive amounts of memory, the first step is to determine where it is using that memory, and track it to a particular variable. For that, the “Performance Profiler library” on page 411 is especially useful. This is a feature available in Analytica Enterprise, and with it, you can determine how much memory is being consumed by each variable in your model. It is not uncommon to find that a small handful of variables consume the bulk of memory, and knowing that, you can focus your efforts on those variables.

**Reduce dimensionality**

Excessive memory usage is most often the result of excessive dimensionality. Analytica makes it very easy to add indexes to your model, but the effect is multiplicative in space and time when those indexes appear in the same array. Finding the right trade-off between dimensionality, level of detail and resource usage (time and computation) is an inherent aspect of model building.

When reducing dimensionality, focus on individual arrays that consume a lot of space as a result of high dimensionality. Can you reformulate your problem using a lower-dimensional array?

**Selective parametric analysis**

Parametric analysis refers to the practice of inserting an extra index into a model input in order to explore how outputs vary across a range of values for the given input. Analytica makes it easy to do this simultaneously for multiple inputs, but in the process, the dimensionality of computed results increases.

Selective parametric analysis is the technique of performing parametric analysis on only a few inputs at a time. All other inputs are left at a single point; hence, only a few parametric indexes are combined in any one evaluation. The user of the model then repeats the process, selecting a new subset of inputs for parametric analysis, and re-evaluating the model.

You can facilitate selective parametric analysis by utilizing Choice menu input controls for potential parametric inputs. See “Creating a choice menu” on page 123.

**Configuring caching**

Whenever Analytica computes the result of a variable, it stores the result in memory. If the value is requested later by another variable, or if the user views the result, Analytica simply returns the previously computed result without having to recompute it again. This is referred to as caching.

It may be unnecessary to cache many of the intermediate variables within your model, or to keep the cached value once all the children of that variable have been computed. You can configure how cached results are retained by setting the CachingMethod attribute for each variable. You can configure a variable to always cache its results, never cache, never cache array values, or release cached results after all children are fully computed. To control the CachingMethod, you must first make the attribute visible as described in “Managing attributes” on page 343. Note that you should always cache results for any variable containing a random or distribution function.

There are some other limitations and interactions to be aware of when managing caching policies, as described further in Controlling when results are cached on the Analytica Wiki.

**Looping over the model**

Analytica’s array abstraction computes the entire model in one pass. Because all intermediate variables are normally cached, this means that the results for all scenarios, across all indexes, are stored in memory for all variables.

---

3. Use of CachingMethod requires Analytica Enterprise.
An alternative is to loop over key dimensions, computing the entire model for a single case at a time, and collecting only the results for final output variable in each case. This consumes only the amount of memory required to compute the entire model for a single scenario, plus the memory for the final result.

The basic technique is illustrated by the following example. Suppose your model has an input \( X \) indexed by \( I \), and an output \( Y \). We also assume that no intermediate steps in the model operate over the index \( I \). Then \( Y \) by looping using:

\[
\text{For } x_i[ ] := X \text{ do WhatIf}(Y,X,x_i)
\]

This simple example is illustrative. As you apply this to complex models involving multiple inputs, outputs and looping dimensions, a greater level of sophistication becomes necessary. The article Looping over a model on the Analytica Wiki covers this topic in greater detail.

**Large sample library**

Monte Carlo simulations with a very large sample size may also lead to insufficient memory. The Large sample library runs a Monte Carlo simulation in small batches, collecting the entire Monte Carlo sample for a selected subset of output variables. The Large Sample Library User Guide is found on the Analytica Wiki, where the library itself is also available for download.

**CompressMemoryUsedBy(A)**

In some cases, the CompressMemoryUsedBy() function is able to reduce the amount of memory consumed by array \( A \). The logical contents of \( A \) remains unchanged, so you will not see a difference, other than possibly a drop in memory usage.

Analytica’s internal representation of arrays is able to accommodate certain forms of sparseness, and when such patterns of sparseness occur within \( A \), CompressMemoryUsedBy(A) condenses the internal representation to leverage the sparseness. There are two forms of sparseness that may occur: Shared subarrays and constant subvectors. For more details on these forms of sparseness, see CompressMemoryUsedBy on the Analytica Wiki.
Chapter 21

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:

\[ \text{Capm}(5\%, 8\%, \text{StockBeta}) \]

This evaluates function \( \text{Capm}(R_f, R_m, \text{Beta}) \) with \( R_f \) set to 5\%, \( R_m \) set to 8\%, and \( \text{Beta} \) set to \( \text{Stockbeta} \).

Name-based calling Analytica also supports a more flexible name-based calling syntax, identifying the parameters by name:

\[ \text{Capm}(	ext{beta: StockBeta, rf: 5\%, rm: 8\%}) \]

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 359).

You can mix positional and named parameters, provided the positional parameters come first:

\[ \text{Ful}(1, 2, D: 4, C:3) \]

You cannot give a positional parameter after a named parameter. For example, the following entry displays an error message:

\[ \text{Ful}(1, D: 4, 2, 3) \text{ Invalid} \]

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 181.

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:

\[(x, y, z: \text{Number})\]

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 369.

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:

\[(a: \text{Number Array}[i, j]; i, j: \text{Index}; c; d: \text{Atom Text Optional ="NA"})\]

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 \(\text{Atom Text}\), 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:

```plaintext
Function Fn1(x)
Parameters: (x: Context)
Mean(Fn1(x))
```

\(\text{Mean}(\cdot)\) is a statistical function that always evaluates its parameter probabilistically. Hence, the evaluation context for \(x\) is probabilistic, and so \(\text{Fn1}\) 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.
Parameter qualifiers

**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:

\[
\text{Mean}(x: \text{ContextSample}[i]; \ i: \text{Index} = \text{Run})
\]

Thus, \(\text{Mean}(x, \text{Run})\) evaluates \(x\) in prob mode. So does \(\text{Mean}(x)\), because the index \(i\) defaults to Run. But, \(\text{Mean}(x, j)\) evaluates \(x\) in mid mode, because \(j\) is not Run.

When the parameter declaration contains more than one dimension, prob mode is used if any of the indexes is Run.

**Mid**  Evaluates the parameter deterministically, or in mid mode, using the mid (usually median) of any explicit probability distribution.

**Prob**  Evaluates the parameter probabilistically, i.e., in prob mode, if it can. If you declare the dimension of the parameter, include the dimension Run in the declaration if you want the variable to hold the full sample, or omit Run from the list if you want the variable to hold individual samples. For example:

\[
(A: \text{Prob} [\ \text{In1}, \text{Run}])
\]

**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:

\[
(A: \text{Sample}[\ \text{In1}, \text{Run}])
\]

**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 \(\text{Sum}(x, i)\) within the function.

**Var**  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 374.

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:

\[
a: \text{Coerce Text} [I]
\]

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:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>Text</td>
<td>&quot;Null&quot;</td>
</tr>
<tr>
<td>Number</td>
<td>Text</td>
<td>Number as text, using the number format of the variable or function calling the function.</td>
</tr>
<tr>
<td>Text</td>
<td>Number or Positive</td>
<td>If possible, interprets it as a date or number, using the number format.</td>
</tr>
<tr>
<td>Null</td>
<td>Reference</td>
<td>\Null</td>
</tr>
<tr>
<td>Number</td>
<td>Reference</td>
<td>\X</td>
</tr>
<tr>
<td>Text</td>
<td>Reference</td>
<td>\Text</td>
</tr>
</tbody>
</table>

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:

\[
A: \text{Ascending} [I]
\]

Optional parameters

You can specify a parameter as optional using the qualifier Optional, for example:

\[
\text{Function } F(a: \text{Number; b: Optional Number})
\]

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:

\[
\text{Function } F(a: \text{Number; b: Number Optional = 0})
\]

It uses the default value if the actual parameter is omitted. Given an equal sign and default value, the Optional qualifier is itself optional (!):

\[
\text{Function } F(a: \text{Number; b: Number = 0})
\]

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:

\[
\text{Function } G(A: \text{Optional; B: C: Optional; D; E: Optional})
\]
You can call \texttt{G} in any of these ways:

\begin{itemize}
\item \texttt{G(1, 2, 3, 4, 5)}
\item \texttt{G(1, 2, , 4)}
\item \texttt{G(, 2, , 4)}
\item \texttt{G(, 2, 3, 4, 5)}
\end{itemize}

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:

\begin{itemize}
\item \texttt{G(B: 2, D: 4)}
\end{itemize}

\texttt{IsNotSpecified(v)}

If you omit a parameter that is not given a default value, you can test this inside the function definition using function \texttt{IsNotSpecified(v)}. For example, the first line of the body of the function might read:

\begin{verbatim}
    If IsNotSpecified(a) then a := 0;
\end{verbatim}

But it is usually simpler to specify the default value in the parameter list as:

\begin{verbatim}
    Function H(x; a := 0)
\end{verbatim}

\section*{Repeated parameters (\ldots\texttt{})}

Three dots, \ldots\texttt{”}, qualifies a parameter as repeatable, meaning that the function accepts one or more actual parameters for the formal parameter. For example:

\begin{verbatim}
    Function ValMax(x: \ldots \texttt{Number}) := \texttt{Max(x)}
    ValMax(3, 6, -2, 4) \to 6
\end{verbatim}

\texttt{ValMax()} returns the maximum value of the actual parameters given for its repeated parameter, \texttt{x}. Unlike the built-in \texttt{Max()} function, it doesn't need square brackets around its parameters.

During evaluation of \texttt{ValMax()}, the value of the repeated parameter, \texttt{x}, is a list of the values of the actual parameters, with implicit (\texttt{Null}) index:

\[ [3, 6, -2, 4] \]

\texttt{ValMax()} can also take array parameters, for example:

\begin{verbatim}
    Variable Z := [0.2, 0.5, 1, 2, 4]
    ValMax(Sqrt(Z), Z^2, 0)
\end{verbatim}

By itself, the qualifier \ldots\texttt{”} means that the qualified parameter expects one or more parameters. If you combine \ldots\texttt{”} with \texttt{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:

\begin{verbatim}
    Function F2(a; b: \ldots)
    F2(1, 2, 3, 4)
\end{verbatim}

Within the function, \texttt{F2}, the value of \texttt{a} is 1, and the value of \texttt{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:

\begin{verbatim}
    Function Fxy(X: \ldots \texttt{scalar}; Y: \ldots \texttt{Optional Scalar})
\end{verbatim}

You have several options for syntax to call the function. Use name-based calling:

\begin{verbatim}
    Fxy(x: 10, 20, 40, y: 2, 3, 4)
\end{verbatim}

Or use position for the first repeated parameter group and name only the second parameter \texttt{y}:

\begin{verbatim}
    Fxy(10, 20, 40, y: 2, 3, 4)
\end{verbatim}

Or enclose each set of repeated parameters in square brackets:

\begin{verbatim}
    Fxy([10, 20, 40], [2, 3, 4])
\end{verbatim}
Depreciated synonyms for parameter qualifiers

Most parameter qualifiers have several synonyms. For example, Atomic, AtomType, and Atom-icType 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 53):
- 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 53) 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 346).

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 361) to the way it is displayed in the Object Finder.
This chapter shows you how to use the procedural features of the Analytica modeling language, including:

- Begin-End, (), and ";" for grouping expressions (page 366)
- Declaring local variables and assigning to them (page 366)
- For and While loops and recursion (page 369)
- Local indexes (page 373)
- References and data structures (page 378)
- Handles to objects (page 382)
- Dialog functions (page 385)
- Miscellaneous functions (page 388)
A **procedural program** is a 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 contains 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 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 407.

### An example of procedural programming

The following function, **Factors(x)**, computes the prime factors of an integer x. It illustrates many of the key constructs of procedural programming.

```plaintext
Numbers identify features below

Function Factors(x)

Definition:
1. VAR result := [1];
2. VAR n := 2;
```

See below for an explanation of each of these constructs, and cross-reference to where they are.
Summary of programming constructs

<table>
<thead>
<tr>
<th>Construct</th>
<th>Meaning</th>
<th>For more see</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1; e_2; \ldots e_i$</td>
<td>Semicolons join a group of expressions to be evaluated in sequence.</td>
<td>page 366</td>
</tr>
<tr>
<td>BEGIN $e_1; e_2; \ldots e_i$ END</td>
<td>A group of expressions to be evaluated in sequence.</td>
<td>page 366</td>
</tr>
<tr>
<td>$(e_1; e_2; \ldots e_i)$</td>
<td>Another way to group expressions.</td>
<td>page 366</td>
</tr>
<tr>
<td>$m \ldots n$</td>
<td>Generates a list of successive integers from $m$ to $n$.</td>
<td>page 375</td>
</tr>
<tr>
<td>Var $x := e$</td>
<td>Define local variable $x$ and assign initial value $e$.</td>
<td>page 366</td>
</tr>
<tr>
<td>Index $i := e$</td>
<td>Define local index $i$ and assign initial value $e$.</td>
<td>page 373</td>
</tr>
<tr>
<td>$x := e$</td>
<td>Assigns value from evaluating $e$ to local variable $x$. Returns value $e$.</td>
<td>page 367</td>
</tr>
<tr>
<td>While Test Do Body</td>
<td>While Test is True, evaluate Body and repeat. Returns last value of Body.</td>
<td>page 371</td>
</tr>
</tbody>
</table>
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 ( and ).

The overall value of the group of statements is the value from evaluating the last expression. For example:

\[
(VAR x := 10; x := x/2; x - 2) \rightarrow 3
\]

Analytica also tolerates a semicolon (;) after the last expression in a group. It still returns the value of the last expression. For example:

\[
(VAR x := 10; x := x/2; x/2;) \rightarrow 2.5
\]

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:

\[
\text{Sum} (\text{Array}_a*\text{Array}_b, N)/(1+\text{Sum} (\text{Array}_a*\text{Array}_b, N))
\]

Define it as:

\[
\text{VAR } t := \text{Sum} (\text{Array}_a*\text{Array}_b, N); t/(1+t)
\]
To compute a correlation between \texttt{Xdata} and \texttt{Ydata}, instead of:

\[
\text{Sum}(\text{Sum}(\text{Xdata}, \text{Data_index})/\text{Nopts})-(\text{Ydata-}
\text{Sum}(\text{Ydata}, \text{Data_index})/\text{Nopts}), \text{Data_index})/
\text{Sqrt}(\text{Sum}(\text{Sum}(\text{Xdata}, \text{Data_index})/
\text{Nopts})^2, \text{Data_index}) * \text{Sum}(\text{Ydata -}
\text{Sum}(\text{Ydata, Data_index})/\text{Nopts})^2, \text{Data_index})
\]

Define the correlation as:

\[
\text{VAR } \texttt{mx} := \text{Sum}(\texttt{Xdata}, \text{Data_index})/\texttt{Nopts};
\]

\[
\text{VAR } \texttt{my} := \text{Sum}(\texttt{Ydata}, \text{Data_index})/\texttt{Nopts};
\]

\[
\text{VAR } \texttt{dx} := \texttt{Xdata} - \texttt{mx};
\]

\[
\text{VAR } \texttt{dy} := \texttt{Ydata} - \texttt{my};
\]

\[
\text{Sum}(\texttt{dx*dy}, \text{Data_index})/\text{Sqrt}(\text{Sum}(\texttt{dx}^2, \text{Data_index})*\text{Sum}(\texttt{dy}^2, \\
\text{Data_index}))
\]

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 \texttt{Correlation()} function (page 293) when data is dimensioned by an index other than the system index \texttt{Run}.

### Assigning to a local variable: \texttt{v := e}

The \texttt{:=} (assignment operator) sets the local variable \texttt{v} to the value of expression \texttt{e}.

The assignment expression also returns the value of \texttt{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, \textit{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 \texttt{Var} or \texttt{Index}, in the \textit{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 \textit{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 407 for details. You can also assign to a global variable \texttt{V} from the definition of \texttt{X} when \texttt{V} is defined as \texttt{ComputedBy(X)}.

### ComputedBy(x)

This function indicates that the value of a variable is computed as a side-effect of another variable, \texttt{x}. Suppose \texttt{v} is defined as \texttt{ComputedBy(x)}, and the value of \texttt{v} needs to be computed, then Analytica will evaluate \texttt{x}. During the evaluation of \texttt{x}, \texttt{x} must set the value of \texttt{v} using an assignment operator.

Even though \texttt{v} is a \textit{side-effect} of \texttt{x}, its definition is still \textit{referentially transparent}, which means that its definition completely describes its computed value.

\texttt{ComputedBy} is useful when multiple items are computed simultaneously within an expression. It is particularly useful from within an \texttt{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), \( n \) evaluates to be a value or values 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 \( x \). The result returned from the assignment operator is the value \( e \), not
the full value of \( x \), which can be a source of confusion but is that way so you can chain assign-
ments, e.g.:

\[ x[i=3] := x[i=5] := 7 \]

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 \( e \). 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 \( i \).

To set a cell in a multi-dimensional array, include multiple subscript coordinates, e.g.:

\[ x[i=2,j=5,k=3] := 7 \]

Slice assignment array abstracts when \( x \), \( n \), or \( e \) have extra dimensions, and the abstraction is
coordinated when an index is shared by \( x \), \( n \), or \( e \). Using abstraction, it is possible to assign to
many cells in a single assignment operation.
For and While loops and recursion

Tip

Analytica’s Intelligent Array features mean 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 150 and page 166).

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.

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. To avoid the error, the expression can be rewritten as:

For j:=i do
If j<2 then x[i=1] else x[i=j-1]

Out-of-range errors can also be avoided without using For by placing the conditional inside an argument. For example, the two examples above can be written without For as follows:

Sqrt(if x<0 then 0 else x)

x[i=(if i<2 then 1 else i-1)]
**Dimensionality reduction**

*For* can be used to apply a function that requires an atom, one- or two-dimensional input to a multi-dimensional result. This usage is rare in Analytica since array abstraction normally does this automatically; however, the need occasionally arises in some circumstances.

Suppose you have an array `A` indexed by `I`, and you wish to apply a function `f(x)` to each element of `A` along `I`. In a conventional programming language, this would require a loop over the elements of `A`; however, in almost all cases, Analytica’s array abstraction does this automatically — the expression is simply `f(A)`, 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 356). 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
```

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
A   B   C
D   E   F
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 219), for an expression of the form `Sum(a*b, i)`, it performs a similar transformation internally.
While(Condition) Do Body

While evaluates Body repeatedly as long as Condition <> 0. For While ... to terminate, Body must produce a side-effect on a local variable that is used by Condition, causing Condition eventually to equal 0. If Condition never becomes False, While continues to loop indefinitely. If you suspect that might be happening, type Control+. (Control+period) to interrupt execution.

Condition must evaluate to an atomic (non-array) value; therefore, it is a good idea to force any local variable used in Condition 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 376 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 Condition 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 376 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:

```pascal
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 \( y \):

```plaintext
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:

```plaintext
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., $A \times A'$, 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 376), which requires its termination condition to be an atom.
Ensuring array abstraction

Chapter 22 Procedural Programming

- If \( b \) Then \( c \) Else \( d \) (page 376), 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 377), such as \( \text{Sum}(x) \), \( \text{Product} \), \( \text{Max} \), \( \text{Min} \), \( \text{Average} \), \( \text{Argmax} \), \( \text{SubIndex} \), \( \text{ChanceDist} \), \( \text{CumDist} \), and \( \text{ProbDist} \).

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 \( B := 1..N \)

\( B \rightarrow \) Evaluation error:

One or both parameters to \( \text{Sequence}(m, n) \) or \( m .. n \) are not scalars.

The expression \( 1..N \), or equivalently, \( \text{Sequence}(1, \ N) \), 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 \( \text{Sequence} \) to be atoms (single elements).

Most functions and expressions that, like \( \text{Sequence} \), 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 \( \text{Sequence} \), \( \text{Subset} \), \( \text{SplitText} \), \( \text{SortIndex} \) (if the second parameter is omitted), \( \text{Concat} \), \( \text{CopyIndex} \), and \( \text{Unique} \).

Why would you want array abstraction using such a function? Consider this approach to writing a function to compute a factorial:

Function \( \text{Factorial2} \)

Parameters: (\( n \))

Definition: \( \text{Product}(1..n) \)

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 \( \text{For} \) loop works fine:

Function \( \text{Factorial3} \)

Parameters: (\( n \))

Definition: \( \text{FOR} \ m := n \ \text{DO} \ \text{Product}(1..m) \)

The \( \text{For} \) loop repeats with the loop variable \( m \) set to each atom of \( n \), and evaluates the body \( \text{Product}(1..m) \) for each value. Because \( m \) is guaranteed to be an atom, this works fine. The \( \text{For} \) loop reassembles the result of each evaluation of \( \text{Product}(1..m) \) 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 \( \text{Atom} \) 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 \( \text{Factorial3} \)

Parameters: (\( n: \text{Atom} \))

Definition: \( \text{Product}(1..n) \)
Index \( K := 1 \ldots 6 \)
\( \text{Factorial3}(K) \rightarrow \)

Notice that \texttt{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 \texttt{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 \texttt{For} loop above. This scheme works fine even if you qualify several parameters of the function as \texttt{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 357 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.

The \texttt{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, \texttt{Atom} is useful to ensure that a function using a \texttt{While} loop array abstracts, as it was for the \texttt{Sequence} function. Here’s a way to write a Factorial function using a \texttt{While} loop:

\[
\text{Function Factorial4} \\
\text{Parameters: (n: Atom)} \\
\text{Definition:} \\
\text{VAR fact := 1; VAR a := 1;} \\
\text{WHILE a < n DO (a := a + 1; fact := fact * a)}
\]

In this example, the \texttt{Atom} qualifier assures that \( n \) and hence the \texttt{While} termination condition \( a < n \) is an atom during each evaluation of \texttt{Factorial4}.

Consider this example:

\[
\text{Variable X := -2..2} \\
\text{Sqrt(X) → [NAN, NAN, 0, 1, 1.414]} \\
\text{The square root of negative numbers -2 and -1 returns NAN (not a number) after issuing a warning. Now consider the definition of Y:} \\
\text{Variable Y := (IF X>0 THEN Sqrt(X) ELSE 0)} \\
Y → [0, 0, 0, 1 1.414]
\]

For the construct \texttt{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 \texttt{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:

\[
\text{Variable Z := [1000, '10,000', '100,000']} \\
\text{This kind of array containing true numbers, e.g., 1000, and numbers with commas turned into text values, often arises when copying arrays of numbers from spreadsheets. The following function would seem helpful to remove the commas and convert the text values into numbers:}
\]

\[
\text{Function RemoveCommas(t)} \\
\text{Parameters: (t)} \\
\text{Definition: Evaluate(TextReplace(t, ',', ''))}
\]
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 RemoveCommas, t, and hence IsText(t), is atomic, either True or False. When False, the IF construct evaluates the Else part but not the Then part, and so calls TextReplace when t is truly a text value. After calling TextReplace separately for each element, it reassembles the results into the array shown above with the same index as Z.

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.
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.

\[
\text{Var Temp}[i1, i2, ...] := X;
\]

**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:

\[
\begin{align*}
\text{Variable M} \\
\text{Definition: 100} \\
\text{Variable Ref_to_M} \\
\text{Definition: \ M}
\end{align*}
\]

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:

```plaintext
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:

```
\ [K] Ksquare →
```

It creates a similar result to \ Ksquare, since K is the only index of Ksquare.
To summarize:

<table>
<thead>
<tr>
<th>( e )</th>
<th>Creates a reference to the value of expression ( e ), whether it is an atom or an array.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>( [] e )</th>
<th>Creates an array indexed by all indexes of ( e ) containing references to all atoms from ( e ).</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>( [i] e )</th>
<th>Creates an array indexed by any indexes of ( e ) other than ( i ) of references to subarrays of ( e ) each indexed by ( i ).</th>
</tr>
</thead>
</table>

| \([i, j \ldots] e \) | Creates an array indexed by any indexes of \( e \) other than \( i, j \ldots \) of references to subarrays of \( e \) each indexed by \( i, j \ldots \). |

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;

Analytica User Guide 381
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 173 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(Val) → Val

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") → Val

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.

ListOfHandles(identifiers...)

Returns a list of handles to the specified Analytica objects, given their identifiers.

ListOfHandles(Val, Va2, Va3) → [Val, Va2, Va3]

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
```

<table>
<thead>
<tr>
<th>General index</th>
<th>Meta-Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index I0 := [E,X,Pi,True] MetaOnly of I0 := 0 {or not set}</td>
<td>Index I1 := [E,X,Pi,True] MetaOnly of I1 := 1</td>
</tr>
<tr>
<td>Variable A0 := Table(I0)(1,2,3,4)</td>
<td>Variable A1 := Table(I1)(1,2,3,4)</td>
</tr>
<tr>
<td>IndexValue(I0) \to [E,X,Pi,True]</td>
<td>IndexValue(I1) \to [E,X,Pi,True]</td>
</tr>
<tr>
<td>Mid(I0) \to 2.718,-1,3.142,1</td>
<td>Mid(I1) \to [E,X,Pi,True]</td>
</tr>
<tr>
<td>A0[I0=Handle(E)] \to 1</td>
<td>A0[I1=Handle(E)] \to 1</td>
</tr>
<tr>
<td>A0[I0=Handle(True)] \to 4</td>
<td>A0[I1=Handle(True)] \to 4</td>
</tr>
<tr>
<td>A0[I0=E] \to 1</td>
<td>A0[I1=E] \to Error:-2.718 not in I1</td>
</tr>
<tr>
<td>A0[I0=-1] \to 2</td>
<td>A0[I1=-1] \to Error:-1 not in I1</td>
</tr>
<tr>
<td>A0[I0=True] \to 4</td>
<td>A0[I1=True] \to Error:1 not in I1</td>
</tr>
<tr>
<td>A0[I0=Handle(E)] \to 1</td>
<td>A0[I1=Handle(E)] \to 1</td>
</tr>
<tr>
<td>A0[I0=Handle(True)] \to 4</td>
<td>A0[I1=Handle(True)] \to 4</td>
</tr>
<tr>
<td>A0[I0=E] \to 1</td>
<td>A0[I1=E] \to Error:-2.718 not in I1</td>
</tr>
<tr>
<td>A0[I0=-1] \to 2</td>
<td>A0[I1=-1] \to Error:-1 not in I1</td>
</tr>
<tr>
<td>A0[I0=True] \to 4</td>
<td>A0[I1=True] \to Error:1 not in I1</td>
</tr>
</tbody>
</table>

**MetaIndex..Do**

The construct, `MetaIndex i := indexExpr`, declares a local meta-index (see “Local indexes” on page 373). 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).

**Local Variables and Handles**

When you declare and use local variables that might hold handles, you should declare these local variables using `MetaVar..Do` or `LocalAlias..Do`, rather than `Var..Do` or `For..Do`. When local variables hold handles, there are two semantic interpretations — the local variable could either be a storage location for a handle to an object, or it could act as an alias to the object. The `MetaVar..Do` or `LocalAlias..Do` declarations makes this distinction clear.

**MetaVar..Do**

The construct, `MetaVar temp := expr`, declares a local variable named `temp` that holds an atomic value or an array of values, some of which might be handles. When the value is a handle, the handle is treated as just another entity, whose datatype happens to be a handle. If you assign another handle or value to `temp`, you simply change the contents of the local variable, and do not alter the object that the handle points to. Use this construct to manage collections of handles.

```
MetaVar hRevModule := Handle(Revenue_module);
MetaVar hExpModule := Handle(Expense_module);
MetaIndex m := [hRevModule,hExpModule];
...
```

For the next example, suppose a global variable named `Hy` exists with a definition of `Handle(Y)`.

```
MetaVar a := Handle(X);
a := Hy
```
Following the assignment, the local variable \( a \) contains a handle to \( Y \). The variable \( X \) is unaltered. You should compare this to the same example using `LocalAlias..Do` in the following section.

When you need to iterate over a collection of handles, you can use, e.g.,

```
MetaVar h[ ] := Contains of Revenue_module;
```

You can assign non-handle values, such as numbers or text, to a local variable declared with `MetaVar`. When you do so, there is no difference between `MetaVar..Do` and `Var..Do`.

### LocalAlias..Do

The construct, `LocalAlias temp := expr`, evaluates `expr` and declares a local variable named `temp` that holds a single atomic value. When `expr` evaluates to a handle, the local variable identifier acts as an exact alias to the object pointed to by the handle.

```
LocalAlias r := Handle(Rate_input) Do r := r * 1.1
```

The preceding expression is functionally identical to\(^1\)

```
Rate_input := Rate_input * 1.1
```

For the next example, suppose a global variable named \( H_y \) exists with a definition of `Handle(Y)`.

```
LocalAlias a := Handle(X)
a := H_y
```

This expression is functionally identical to

```
X := H_y
```

The expression, if evaluated in a context where side-effects are permitted, would alter the definition of \( X \). Following the assignment to \( a \), the local variable \( a \) continues to be an alias to \( X \). You should compare this to the same example for `MetaVar..Do` in the preceding section.

When `expr` is an array of handles, then the body expression is iterated for each alias. The following expression increases the definition of every global variable within an indicated module to a value 1.1 times its initial value.

```
LocalAlias x := Contains of Input_module;
x := x * 1.1
```

The rule for `LocalAlias` is that the result for the expressions using `temp` are identical to what you would get if you replaced every occurrence of `temp` within those expressions with the identifier for the variable pointed to by the handle, and then evaluated the expressions\(^2\).

An easy mistake to make when using `LocalAlias` is to forget that `expr` is evaluated before the assignment. This feature allows you to compute the handle to the object that will be aliased, but it also means that you need to remember to surround an identifier with a call to `Handle()` when you just want a simple alias. If you write

```
LocalAlias x := y
```

then this assigns to \( x \) the value of \( y \), not a handle to the object \( y \). If \( y \) happens to hold a handle to \( z \), then \( x \) becomes an alias to \( z \), not an alias to \( y \). To obtain an alias for \( y \), you must use

```
LocalAlias x := Handle(y);
```

You can assign non-handle values, such as numbers and text, to a `LocalAlias` variable. When you do so, the local variable identifier serves as an alias for a single value, and is the same as

```
Var temp[ ] := expr;
```

---

\(^1\) This expression contains a side-effect to a global variable. Side-effects to global variables are only allowed in expressions or user-defined functions that are run directly from button scripts. This would cause an error if attempted from a variable’s definition.

\(^2\) Not every object in Analytica lives in the global namespace (e.g., local indexes), so this rule doesn’t literally apply for handles to objects not in the global namespace. For these, we cannot replace the local name with an identifier since no global identifier exists for the object pointed to by the handle.
Conversions between MetaVar and LocalAlias

When writing expressions that manipulate handles in local variables, you may sometimes need to convert between MetaVar and LocalAlias semantics. When a MetaVar holds a handle, its value is an entity with the data type of handle. Since subtraction is not defined for a handle, the following expression is an error

\[
\text{MetaVar } hr := \text{Handle(Revenue)} \text{ Do } hr - \text{Expenses} \quad \{ \text{Error} \}
\]

The expression is attempting to subtract the value of expenses (a number) from a handle. In this example, we need the LocalAlias semantics instead, so given \( hr \) we can convert using

\[
\text{MetaVar } hr := \text{Handle(Revenue)}; \\
\text{LocalAlias } r := hr; \\
r - \text{Expenses}
\]

A MetaVar can also be dereferenced by using the \texttt{Evaluate()} function

\[
\text{MetaVar } hr := \text{Handle(Revenue)} \text{ Do } \text{Evaluate(hr)} - \text{Expenses}
\]

In the other direction, when a LocalAlias, \( r \), is an alias for another object and you need the handle to the object, use \texttt{Handle(r)}. It is legal to use the \texttt{Handle()} function on a local variable declared using LocalAlias, but it is an error to apply the \texttt{Handle()} function to a local variable defined using MetaVar. This is because a local variable is not an object (it is just a name for a value), and since handles can only point to objects, there is no such thing as a handle to a local variable\(^3\).

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. (\texttt{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.

\texttt{MsgBox(message, buttons, title)}

Displays a dialog with the text \texttt{message}, a set of \texttt{buttons} and an icon (according to numerical codes below), with \texttt{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 \texttt{buttons} parameter is a number that controls which buttons to display, as follows:

\begin{itemize}
  \item 0 = OK only
  \item 1 = OK and Cancel (the default if \texttt{buttons} is omitted)
  \item 2 = Abort, Retry, and Ignore
  \item 3 = Yes, No, and Cancel
  \item 4 = Yes and No
  \item 5 = Retry and Cancel
\end{itemize}

---

\(^3\) A local index is an object, so you can have a handle to a local index. The \texttt{Index..Do} and \texttt{MetaIndex..Do} constructs create an index object, along with a local variable name that has LocalAlias semantics for the index object.
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') →
```
Dialog functions

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 117), it shows the error message when the check fails instead of the default check error message, letting you tailor the message.

AskMsgChoice(question, title, optionList, default, showAll, comboBox)
Opens a dialog displaying question text with a choice drop down control where the user can select an answer from a list. The optionList parameter is a list or 1-D array of selections, and default is initial selection. The selected item is returned as the result.

When showAll is true, then an «All» option is also included, and if selected then the entire optionList is returned. When comboBox is specified as true, then a combo box control is displayed rather than a choice list. In a combo box, the user is free to enter his own text, but the optionList provides suggestions.

Example
```
AskMsgChoice("Please rate","Survey",['Loved','Liked', 'Neutral', 'Disliked', 'Hated'])
```

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.

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)
```

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.

![Progress Dialog](image)

**Declaration**  
`ShowProgressBar(title, text: Text atomic; p: number atomic)`  

**Example**  
In this example:

```analytica
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:

\[
\text{(VAR } x := 10; \text{ Evaluate}(x \& \"+\& x)) \rightarrow 20
\]

If \( e \) is a number or expression that is not a text value, it just returns its value:

\[
\text{Evaluate}(10M /10) \rightarrow 1M
\]

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.

\text{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 \text{Evaluate(e)}. For example, this would give an evaluation error:

\[
\text{Variable } A := (\text{VAR } r := 99; \text{ Evaluate('r^2') })
\]

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.

\text{Evaluate and dependencies}

Analytica’s dependency mechanism does not work with variables or functions whose identifiers appear inside the text parameter of \text{Evaluate}. For example, consider:

\[
\text{Variable } B := \text{Evaluate("F(A)")}
\]
\[
\text{Variable } C := F(A)
\]

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 \text{Evaluate} is appropriate; otherwise, use it only with care.

\text{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 \text{ShowPdfFile}, \text{ReadTextFile}, or \text{RunConsoleProcess}). The locations of those files could be stored in the registry by your installer, so that your model knows where to look.

\text{Example} \ GetRegistryValue("HKEY_CURRENT_USER", "Software/MyCompany/MyProduct", "FileLocation")

\text{IgnoreWarnings(expr)}

Evaluates its parameter \( expr \), and returns its value, while suppressing most warnings (page 432) 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 \text{Show Result Warnings} in the Preferences dialog (page 54), because you do want to see warnings that might appear in other parts of the model.

\text{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 \text{Sample(IsResultComputed(x))}, or to test the mid value use \text{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 23

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
- Reading and writing data in Excel worksheets.
- Reading data from the internet
- 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
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.

![Diagram of result table with record and label nodes]

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.

Suppose as an example that we have a database containing the addresses individuals. To ensure the titles are meaningful, we name the indexes `Individuals` and `Address_fields`. The query is then encoded in the indexes and variable as follows.

```plaintext
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.

**Tip**
In 64-bit Windows, ODBC drivers may be either 64-bit drivers or 32-bit drivers. When you are using Analytica 64-bit, you can only make use of 64-bit ODBC drivers. If you do not have a 64-bit driver for the database you are using installed on your computer, you will not be able to query that database from Analytica 64-bit. Likewise, the 32-bit editions of Analytica can only make use of 32-bit ODBC drivers, so the appropriate 32-bit driver must be installed on your computer.

Note that 64-bit versions of the Microsoft JET drivers do not exist. These drivers are installed with Microsoft Access and include the Access ODBC driver, the Excel ODBC driver, and the flat file ODBC driver. Microsoft apparently has no plans to release 64-bit versions of these drivers, and has indicated it wants to phase out the use of these drivers entirely. These drivers therefore cannot be utilized from Analytica 64-bit. Most other major database drivers are available in both 32- and 64-bit.

**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.

**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:

'\texttt{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:

'\texttt{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:

'\texttt{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 \texttt{DBTable()}. Thus, you should take care to ensure that duplicate column labels do not result. This can be accomplished, for example, using the \texttt{AS} keyword, for example:

'\texttt{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:

'\texttt{SELECT * FROM myQuery}'

To retrieve a result table from a data source, you need:

1. The data source connection text.
2. The SQL query. These are discussed in the previous two sections. For illustrative purposes, suppose the connection text is 'DSN=Automobile Data', and the SQL statement is '\texttt{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) (pure relational transformation) is described in “MDArrayToTable(A, I, L) (pure relational transformation)” on page 202. 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.

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.

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:

\[
\text{Variable Write\_A\_to\_DB := DBWrite(DB, WriteTableSql(A, RowIndex, LabelIndex, 'TableA'))}
\]

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:

\[
\text{Write\_A\_to\_DB := DBWrite(DB, WriteTableSql(A, RowIndex, LabelIndex, 'TableA')); 'Done'}
\]

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(connection, sql, key)

Used to define an index variable. The definition of the index should contain only one DBQuery() statement. connection specifies a data source (e.g., 'DSN=MyDatabase') and sql defines an SQL query. The optional key parameter may specify the name of a column in the database that will be used to determine the row index values returned by DBQuery. It is best to only use key columns that contain unique values in each row.

When placed as the definition of an index variable, DBQuery() is evaluated as soon as the definition is complete, or immediately when the model is loaded. If you place it in a variable node, it will be evaluated when the result is requested, rather than when the model is loaded, provided there are no index nodes downstream that depend on it. 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().

When the optional key parameter is not specified, DBQuery() returns a sequence 1..n, where n is the number of records (rows) in the result table. When key is specified, DBQuery returns a list consisting of the values from that column in the database source.

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, \textit{dbIndex}, must be the name of a variable (normally an index) in your Analytica model that is defined with a \texttt{DBQuery()} statement. If the second argument, \textit{column}, is a text value, it identifies the name of a column label in the result table, in which case \texttt{DBTable()} returns a 1D array (indexed by \textit{dbIndex}) with the data for that column. If the second argument is a list of text values (the \textit{columnList} form), then \texttt{DBTable()} returns a 2D table with records indexed by \textit{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 \textit{columnIndex}, and rows indexed by \textit{dbIndex}. With this last form, \textit{columnIndex} can be defined as \texttt{DBLabels(dbIndex)}.

\textbf{DbTableNames(connection, catalogs, schemas, tables, tableTypes)}

Connects to an ODBC data source and returns catalog data for the data source. \textit{connection} specifies a data source (e.g., '\texttt{DSN=MyDatabase}'). \texttt{Catalogs, schemas, tables, and tableTypes} may be names or patterns (as long as the driver manager on your computer 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 '\\\'. \texttt{tableTypes} 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.

\textbf{Examples} To get all catalog entries (tables, views, schemas, etc) in \textit{My db}:
\begin{verbatim}
   DBTableNames('DSN=My db')
\end{verbatim}
To get just information on tables in \textit{My db}:
\begin{verbatim}
   DBTableNames('DSN=My db',tables:'%')
\end{verbatim}
To get all valid views:
\begin{verbatim}
   DBTableNames('DSN=My db',tableTypes:'VIEW')
\end{verbatim}
The precise value you give to \texttt{tableTypes} depends on which database server you are querying. If you are not sure which type names are used by your database, evaluate \texttt{DBTableNames} first with none of the parameters and you should see a column that provides the type names for existing catalog entries.

\textbf{DBWrite(connection, sql)}

This function is identical to \texttt{DBQuery()} except that the query is processed in read-write mode, making it possible to store data in the data source from within Analytica.

\textbf{MdxQuery(connection, mdx)}

\texttt{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. \texttt{MdxQuery()} works with Microsoft SQL Server Analysis Services.

\textit{connection} is the standard text used to identify and connect with the database, similar to that used in other database functions, such as \texttt{DBQuery()}. \textit{mdx} is text containing the query in the MDX language.

\texttt{MdxQuery()} creates a local index for each dimension. The local indexes are named \texttt{.Axis1}, \texttt{.Axis2}, \texttt{.Axis3}, etc., and contain the cube member captions as elements. Some cube axes returned from MDX queries are hierarchical, and for these, \texttt{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.

Reading and writing from Excel spreadsheets

Several functions on the Database menu allow direct reading and writing from Excel workbooks. These functions launch Excel (without the GUI if it isn’t already running) to access and compute the requested cells, so they require Microsoft Excel to be installed on your computer in order to use these functions.
SpreadsheetOpen( filename, showDialog )

Opens the indicated Excel workbook file. The optional showDialog parameter can be specified as true or false to force or prevent the display of the file selector dialog. Returns a special «workbook» object which is passed as a parameter to the other Excel functions.

SpreadsheetSave(workbook, filename)

Saves an Excel workbook to a file. The workbook parameter must be a workbook object returned from a previous call to SpreadsheetOpen(). The workbook is saved to the same file it was originally loaded from when the filename parameter is omitted. This function is used after changes have been made to the workbook using SpreadsheetSetCell() or SpreadsheetSetRange().

SpreadsheetCell( workbook, sheet, column, row, what )

Reads the value of one cell in an Excel worksheet given its coordinates. The function fully array abstracts so that it can also be used to read an range of cells by specifying the column or row parameters as arrays. Workbook is the result of a call to SpreadsheetOpen. The sheet and column parameters may be either the textual names or the ordinal number identifying the sheet in the workbook or column in the worksheet.

The following demonstrates equivalent methods for obtaining the value from cell C7 of the first worksheet, "Sheet1":

SpreadsheetCell(workbook,'Sheet1','C',7)
SpreadsheetCell(workbook,1,3,7)

The following reads the 2-D array of cells C3:J10 in Sheet1:

Index Year := 2008..2015;
Index Asset := 1..16;
SpreadsheetCell(workbook,'Sheet1',@Year+2,@Asset+3)

You can obtain the contents of a particular cell coordinate from all sheets by specifying sheet at '***'. The result is indexed by a local index named .Sheet containing the sheet names. This is particularly useful for obtaining a list of worksheets.

Index Sheet := CopyIndex(SpreadsheetCell(workbook,'***',1,1).Sheet)

When the optional what parameter is not specified, the value of the cell is retrieved. Use what to read other types of cell information: 'Value', 'Formula', 'RelativeFormula', 'NumberFormat', 'TextColor', 'BackColor', 'FontName', 'FontSize', 'FontStyle', or 'Border[Left|Right|Up|Down][Color|Style|Weight]' (e.g., 'BorderRightColor').

SpreadsheetSetCell(workbook, sheet, col, row, value)

Writes value (either a number or text) to the cell identified by sheet, col, and row. Can be used to write multiple values via array abstraction when col, row and value are arrays that share one or more indexes. The first parameter, workbook, must be an object obtained from a previous call to SpreadsheetOpen().

To write the value 3.5 to cell Sheet1!C5:

SpreadsheetSetCell(workbook,'Sheet1','C',5,3.5)

SpreadsheetRange(workbook, range, colIndex, rowIndex, howToIndex, sheet, what )

Reads a range of cells from an Excel workbook. The range can be the name of a named range in the Excel workbook, range coordinates such as "Sheet1!C7" or "Sheet3!C7:F12", or all populated cells in a sheet as "Sheet1". Alternatively, you can specify a range coordinate as "C7:F12" and specify the sheet either by name or number in the optional sheet parameter. If you omit range but specify sheet, then the smallest rectangular range of cells containing all populated
cells is returned. When specifying a named range, it is not necessary to specify the worksheet name, but when using range coordinates the sheet must be identified.

Depending on the dimensions of the cells requested, the result may be a scalar (single cell), a column vector, a row vector, or a 2-D array. When the range contains multiple cells, the result is an array. When the optional `colIndex` or `rowIndex` parameters are not provided, local indexes are created for those dimensions when the range for the result contains more than one column or more than one row. If you already have indexes for those dimensions, you should provide them explicitly in the `colIndex` and `rolIndex` parameters.

The optional `howToIndex` parameter provides additional control over how the result is indexed and the range interpreted. It can be any additive combination of the following values:

- `1 = Force column index when range has only one column`
- `2 = Force row index when range has only one row`
- `4 = First row of range contains column labels`
- `8 = First column of range contains row labels`
- `16 = Do not issue error if supplied `colIndex` or `rowIndex` is not the correct length`

Suppose the range of cells, `C7:F12` in sheet `Projection`, the third sheet in the Excel workbook, has been labelled in Excel with the name `Costs`. Then the following are each equivalent and return a 2-D array, indexed by local indexes `.Row=[7,8,9,10,11,12]` and `.Column=['C','D','E','F']`:

- `SpreadsheetRange(workbook,'Costs')`
- `SpreadsheetRange(workbook,'Projection!C7:F12')`
- `SpreadsheetRange(workbook,'C7:F12',sheet:'Projection')`
- `SpreadsheetRange(workbook,'C7:F12',sheet:3)`

When the optional `what` parameter is not specified, the value of the cell is retrieved. Use what to read other types of cell information: `'Value'`, `'Formula'`, `'RelativeFormula'`, `'NumberFormat'`, `'TextColor'`, `'BackColor'`, `'FontName'`, `'FontSize'`, `'FontStyle'`, or `'Border[Left|Right|Up|Down][Color|Style|Weight]'` (e.g., `'BorderRightColor'`).

**SpreadsheetSetRange( workbook,range,value,colIndex,rowIndex,sheet )**

Writes `value` (often an array of text or numbers) to a cell range or named range. `Workbook` must be an object obtained from a previous call to `SpreadsheetOpen()`. `Sheet` is the name or number of the worksheet containing the range. `Range` may be a coordinate range (e.g., `'Sheet1!C7:F15'`, or when the optional `sheet` parameter is also specified `range` may be just `'C7:F15'`) or the name of a named range. `Value` is the value written, and `colIndex` and `rowIndex` are the indexes of `value` (when `value` is an array).

**SpreadsheetInfo(workbook,item)**

Reads a property of a workbook. `Workbook` must be an object obtained from a previous call to `SpreadsheetOpen`. `Item` is one of the following:

- `'AcceptLabelsInFormulas'`: Flag. 1 if cell formulas can use labels (named ranges) in their formulas. 0 if these are not allowed.
- `'ActiveSheet'`: Textual name of the active worksheet.
- `'Author'`: Author of workbook.
- `'CalculationMode'`: This is either `'Automatic'`, `'Manual'` or `'Semiautomatic'`, indicating whether cells are automatically recomputed when something changes.
- `'Semiautomatic'` means that cells are calculated automatically except those that are part of a data table.
- `'CodeName'`: The textual code name field of the workbook, if one has been specified.
- `'Date1904'`: Flag. 1 if the excel date origin is 1 Jan 1904. 0 if it is 1 Jan 1900.
- `'Filename'`: Full path and filename of the workbook file.
Chapter 23  Analytica Enterprise

Reading data from the internet

- 'Name': The filename part of the workbook.
- 'Names': A list of all the named ranges in the spreadsheet.
- 'ReadOnly': Flag. 1 if the workbook is read-only, 0 if it can be changed.
- 'Saved': Flag. 1 if there are no unsaved changes. 0 if it has been "dirtyed".
- 'SelectedRange': The range of cells that is currently selected. When you have the spreadsheet open interactively in Excel at the same time your model is using it, this can be used to detect which range of cells you have highlighted, and can be used as a way to make a selection. An example return value is 'Sheet1!B2:C4'.
- 'SelectedRangeR1C1': The range of cells currently selected in R1C1 notation, such as 'Sheet1!R[1]C[1]:R[3]C[2]'.
- 'Sheets': A list of the worksheet names in the workbook.
- 'Title': Textual title (if it has been filled in) for the workbook.

SpreadsheetSetInfo(workbook,item,value)

Sets a property of a workbook. Workbook must be an object obtained from a previous call to SpreadsheetOpen. Item is one of the following:

- 'ActiveSheet': specify the name of a worksheet for the value parameter. The worksheet with that name becomes selected.
- 'Author': sets the author property of the workbook to the indicated textual value.
- 'CalculationMode': Value should be one of 'Automatic', 'Manual' or 'Semiautomatic'. Sets the calculation mode.
- 'CodeName': Sets the textual code name property of the workbook.
- 'Date1904': Value should be 0 or 1. When 1 sets the date origin used by Excel to 1-Jan-1904, the same used by default by Analytica. When 0, uses 1-Jan-1900.
- 'SelectedRange': Selects an indicated range of cells. Can be used to indicate a range of cells to a user when Excel is open interactively. The value parameter should be something like 'Sheet1!B2:B4'.
- 'Title': Sets the textual title of the workbook.

Reading data from the internet

The ReadFromUrl function, on the Database menu, reads text from URL sources, such as HTTP web pages or FTP sites.

ReadFromUrl(url, method, formValues, formFields, formIndex)

Reads text or images from a URL (Universal Resource Locator) location on the web. In most cases, this is called with a single parameter -- the URL. When reading from a web page, the URL should begin with 'http://...'. When reading a text file from an FTP site, it should begin with 'ftp://...'. The entire text from the requested location is returned as a text value. In most cases, you will need to write Analytica expressions to parse the data. When the URL is an http request that points to an image (*.jpg, *.png, *.gif, etc), a picture object is returned that can be assigned to the pict attribute of objects. With the exception of http requests to known image types, the source data must be textual, and if not, the resulting data will be garbled.

Some web pages and most FTP sites require user and password authentication to access. To authenticate, embed the user name and password in the URL as follows:

ReadFromUrl('http://user:password@www.site.com/dir/page.htm')
ReadFromUrl('ftp://user:password@site.com/directory/file.txt')

The optional second through fourth parameters are used to submit data to web forms. There are two methods used to submit data to HTTP web forms: method: 'POST' or method: 'GET'. The field names and the values submitted for those fields are specified in formValues and form-
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 encrypt and optionally Save as a browse-only model copy to save an encrypted copy — that is a file scrambled into a non-human-readable form.
4. Distribute the encrypted 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 browse-only (unprotected) copy. Until the model is stored in an encrypted form (step 3), an end user is not prevented from un-hiding your definitions, or from viewing them by other means (e.g., by loading the Analytica model file into a text editor).

Tip
An encrypted model file cannot be decrypted, 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 encrypted 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.
Making a browse-only model and hiding definitions

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 encrypted. In this case, encryption 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 encrypted 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 encrypted copy of your model
When you are ready to save an encrypted 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.

Check **Lock and encrypt the copy** 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.

Check **Save as a browse-only model** 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.

Optionally check **Save everything in one file by embedding linked modules** to produce a single file that can be distributed. If your model is utilizing unlocked linked modules, the content in those may remain exposed unless you use this option (alternatively, you can link to locked copies of those modules in your main model before saving your main model in a locked form). Even if you do not lock your model, this option can provide a convenient way to distribute your model as a single file to end-users, or to bundle it for upload to the Analytica Cloud Player. **Publish to cloud** on the File menu saves everything in a single file automatically during the upload.

A browse-only model is always encrypt to prevent anyone from editing the source Analytica file. Thus, it automatically checks **Lock and encrypt 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. Enterprise and Optimizer-level features are available to users viewing your model through the Analytica Cloud Player (ACP).

When a browse-only model (saved as such from Enterprise) is loaded into Analytica Professional, it runs it in Power Player mode, so that database access, etc., is available.

### Linked libraries and submodules

**Warning!!** When your model utilizes linked libraries or submodules, you need to be very careful that the module files you distribute to your users are indeed encrypted or locked as you intend, while at the same time ensuring that you do not accidentally encrypt your master copies of these modules.

Because locking a model as encrypted or browse only is an irreversible operation, it is extremely important that you don’t accidentally lock your master versions. If your model uses linked libraries or linked modules, to avoid inadvertently making these mistakes, it is highly recommended that you embed all modules and libraries check in your encrypted copy by checking **Save everything in one file by embedding linked modules**.

When you save a copy of your model, without checking the **Save everything in one file by embedding linked modules** option, Analytica 4.2 and later will save a locked copy only of the top-level model file. The locked status of any linked modules remain in their original state, such that individual module files may remain non-encrypted or editable. If you don’t want to embed all linked modules in your encrypted copy, then you’ll need to save an encrypted copy of each submodule individually, and then use “**Add Module...**” with the **Link and Merge** options to switch your model to using the locked copy, prior to saving a copy of your top-level model.

If you ever use a pre-4.2 release of Analytica, take extreme caution when encrypting a model containing linked modules -- release 4.1 and earlier will also encrypt the linked module files. Thus, it is imperative that you make a copy of all linked submodules before saving an encrypted copy.

In Analytica 4.2 and later, you can distribute locked copies of libraries or modules, allowing other model developers to utilize those libraries without being able to view your proprietary definitions (when they are encrypted) or being able to modify the libraries (when locked as browse-only).

When using locked libraries, certain operations involving objects in those modules are restricted. You cannot embed an encrypted sub-module within a non-encrypted module, but you can use it as a linked module. Embedding an encrypted module in an encrypted module is allowed. Various operations that might allow a user to gain access to your hidden definitions are disallowed, such
as moving an object with a hidden definition from an encrypted module to a non-encrypted module.

Warning!! If you distribute locked module files for use by others to use in their models, it is important to ensure that they are using Analytica 4.2 or later. Loading an encrypted or browse-locked module into a model from Analytica 4.1 or earlier will result in the entire model being placed in the same locked state. The ability to utilize a combination of locked and unlocked modules in the same model requires Analytica 4.2 or later.

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.

Scripts in other input controls
You can also include scripts in Checkbox and Choice input nodes, which get executed when the user uses the control to change the value. You can access the Script attribute for these from the Attribute Panel. If you enable the Script attribute from the Attributes dialog on the Objects menu, then the Script attribute will also display in the Object Window.

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 367.

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:

\[
\text{Variable } A := 100
\]
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.

One useful application of assigning to a global variable is to save the results of a long computa-
tion. 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 statis-
tical 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 affects, 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.

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, vari-
able, 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
two numbers, with an index $I$ of 20 elements, $Time$ has 30 elements, and $Samplesize = 1000$, would
use about $20 \times 30 \times 1000 \times 12 = 7,200,000$ bytes or 7.2 Megabytes. Analytica uses an efficient
representation for arrays with many zeroes (sparse arrays) or many repeated values. An array
that is an exact or partial copy of another array 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 com-
puter 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 usu-
ally much slower than RAM, using virtual memory often causes the application to slow down sub-
stantially. 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 appli-
cation. 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, **RunCon-
soleProcess 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")
```
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:

\[ y_1, y_2, y_3, \ldots, y_m \]

You can estimate the mean and standard deviation of y using the following equations:

\[ \bar{y} = \frac{1}{m} \sum_{i=1}^{m} y_i \]  

\[ s^2 = \frac{1}{m-1} \sum_{i=1}^{m} (y_i - \bar{y})^2 \]

This leads to the following confidence interval with confidence \( \alpha \), where c is the deviation for the unit normally enclosing probability \( \alpha \):

\[ (\bar{y} - c \frac{s}{\sqrt{m}}, \bar{y} + c \frac{s}{\sqrt{m}}) \]

Suppose you wish to obtain an estimate of the mean of y with an \( \alpha \) confidence interval smaller than w units wide. What sample size do you need? You need to make sure that:

\[ w > 2c \frac{s}{\sqrt{m}} \]

Or, rearranging the inequality:

\[ m > \left( \frac{2cs}{w} \right)^2 \]

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:

\[
m > \left( \frac{2 \times 2 \times 40}{20} \right)^2 = 8^2 = 64
\]

(7)

So, to get the required precision for the mean, you should set the sample size to about 64.

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:

\[ y_1 \leq y_2 \leq \ldots \leq y_m \]

c is the deviation enclosing probability \( \alpha \) of the unit normal. Then the following pair of sample values constitutes the confidence interval:

\((y_i, y_k)\)

where:

\[
i = \left\lfloor mp - c \sqrt{mp(1-p)} \right\rfloor
\]

(8)

\[
k = \left\lceil mp + c \sqrt{mp(1-p)} \right\rceil
\]

(9)

**Note:** The brackets in equations (8) and (9) above mean round up \( \left\lceil \right\rceil \) and round down \( \left\lfloor \right\rfloor \), since they are computing numbers that need to be integers.

Suppose you want to achieve sufficient precision such that the \( \alpha \) confidence interval for the \( p \)th fractile \( Y_p \) is given by \((y_i, y_k)\), where \( y_i \) is an estimate of \( Y_{p - \Delta p} \) and \( y_k \) is an estimate of \( Y_{p + \Delta p} \). In other words, you want \( \alpha \) 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:

\[
i = m(p - \Delta p), \quad k = m(p + \Delta p)
\]

(10)

Thus:

\[
k - i = 2m\Delta p
\]

(11)

From equations (8) and (9) above, you have:

\[
k - i = 2c \sqrt{mp(1-p)}
\]

(12)

Equating the two expressions for \( k - i \), you obtain:

\[
2m\Delta p = 2c \sqrt{mp(1-p)}
\]

(13)

\[
m = p(1-p)\left( \frac{c}{\Delta p} \right)^2
\]

(14)

For example, suppose you want to be 95% confident that the estimated fractile \( Y_{0.90} \) is between the estimated fractiles \( Y_{0.85} \) and \( Y_{0.95} \). So you have \( \Delta p = 0.05 \), and \( c = 2 \). Substituting the numbers into equation (14), you get:

\[
m = 0.90 \times (1 - 0.90) \times (2/0.05)^2 = 144
\]

(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:

\[
m = 0.5 \times (1 - 0.5) \times (2/0.01)^2 = 10,000
\]  

(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.
Appendices

Appendices

Appendix B: Menus

File menu

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Model</td>
<td>Starts a new model.</td>
</tr>
<tr>
<td>Open Model</td>
<td>Opens an existing, previously saved model.</td>
</tr>
<tr>
<td>Add Module</td>
<td>Adds a module to the active model.</td>
</tr>
<tr>
<td>Add Library</td>
<td>Opens file finder at Analytica Libraries folder to add a library module.</td>
</tr>
<tr>
<td>Close</td>
<td>Closes the active window.</td>
</tr>
<tr>
<td>Close Model</td>
<td>Closes the model after prompting you to save the file if it has changed.</td>
</tr>
<tr>
<td>Save</td>
<td>Saves the model in its file. If the model has not been saved before, prompts for a file name and folder. If it has linked modules that have changed, it also saves them.</td>
</tr>
<tr>
<td>Save As</td>
<td>Saves the active model, file module, or file library as a new file, after asking for new file name and folder.</td>
</tr>
<tr>
<td>Save A Copy In</td>
<td>Saves a copy of the active model (or file module) into a new file, after prompting for a file name, leaving the original file name for future saves.</td>
</tr>
<tr>
<td>Import</td>
<td>Imports the contents of a text or data file into the selected variable definition. See “Importing and exporting” on page 334.</td>
</tr>
<tr>
<td>Export</td>
<td>Exports the contents of the selected field or cells into a file. See “Importing and exporting” on page 334.</td>
</tr>
<tr>
<td>Publish to cloud</td>
<td>Uploads the model to the Analytica Cloud Player (ACP), so that you can invite others to view it from web browsers. The first time you use this, it will assist you in setting up an ACP account. Later you can also change your account settings from the dialog that pops up when this option is selected.</td>
</tr>
<tr>
<td>Manage published models</td>
<td>Jumps to your model listing page in Analytica Cloud Player (ACP). This page displays in a web browser.</td>
</tr>
<tr>
<td>Print Setup</td>
<td>Opens a dialog for selecting paper size, orientation, and scaling options for printing.</td>
</tr>
<tr>
<td>Print Preview</td>
<td>Opens a view showing where page breaks occur before the current window is printed.</td>
</tr>
<tr>
<td>Print</td>
<td>Opens a dialog for selecting the printer, number of copies you want to print, and other printing options.</td>
</tr>
<tr>
<td>Print Report</td>
<td>Opens a dialog for printing multiple diagrams, Object windows, and result windows at the same time. See “Printing” on page 23.</td>
</tr>
<tr>
<td>Recent files</td>
<td>Lists the six most recently opened Analytica model files. Select one to open that model.</td>
</tr>
<tr>
<td>Exit</td>
<td>Quits the Analytica application, after prompting to save any model changes to file.</td>
</tr>
</tbody>
</table>
## Edit menu

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Undo</strong></td>
<td>Undoes your last action. “Can’t Undo” appears in this location if an undo is not possible.</td>
</tr>
<tr>
<td><strong>Cut</strong></td>
<td>Cuts the selected text, node(s), or table cells into the clipboard temporarily for pasting.</td>
</tr>
<tr>
<td><strong>Copy</strong></td>
<td>Copies the selected text, node(s), graph, or table cells into the clipboard temporarily for pasting. See “Copying and pasting” on page 328.</td>
</tr>
<tr>
<td><strong>Paste</strong></td>
<td>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 328.</td>
</tr>
<tr>
<td><strong>Paste Special</strong></td>
<td>Brings up a dialog to select the format of data to OLE link into an edit table.</td>
</tr>
<tr>
<td><strong>Clear</strong></td>
<td>Deletes the selected text or node(s).</td>
</tr>
<tr>
<td><strong>Select All</strong></td>
<td>Selects all the text in an attribute field, nodes in a diagram, or cells in a table.</td>
</tr>
<tr>
<td><strong>Duplicate Nodes</strong></td>
<td>Duplicates the selected nodes onto the same diagram. See “Duplicate nodes” on page 47.</td>
</tr>
<tr>
<td><strong>Copy Diagram or Copy Table</strong></td>
<td>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 328.</td>
</tr>
<tr>
<td><strong>Insert Rows or Insert Columns</strong></td>
<td>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 178.</td>
</tr>
<tr>
<td><strong>Append Rows or Append Columns</strong></td>
<td>Appends an item to the end of a list, or row or column of a table. This item is only available when the last item in the list, row or column is selected. See “Editing a table” on page 178.</td>
</tr>
<tr>
<td><strong>Delete Rows or Delete Columns</strong></td>
<td>Deletes the selected item or items in a list, or rows or columns in a table. See “Editing a table” on page 178.</td>
</tr>
<tr>
<td><strong>Preferences</strong></td>
<td>Opens the Preferences dialog (page 54) to examine or change various options.</td>
</tr>
<tr>
<td><strong>OLE Links</strong></td>
<td>Opens a dialog to let you change properties for OLE links from external applications into your model. See Chapter 19, “Importing, Exporting, and OLE Linking Data.”</td>
</tr>
</tbody>
</table>
## Object menu

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find</td>
<td>Opens a <strong>Find</strong> dialog to search for an object by its identifier or title. If a table is in focus, brings up the <strong>Find in Table</strong> dialog. See “Finding variables” on page 342.</td>
</tr>
<tr>
<td>Find Next</td>
<td>Finds the next object that partially matches the previously defined text value. See “Finding variables” on page 342.</td>
</tr>
<tr>
<td>Find Selection</td>
<td>Finds an object by its identifier that matches the currently selected text. See “Finding variables” on page 342.</td>
</tr>
<tr>
<td>Make Alias</td>
<td>Creates an alias for the selected object(s). See “Alias nodes” on page 50.</td>
</tr>
<tr>
<td>Make Importance</td>
<td>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 297.</td>
</tr>
<tr>
<td>Make Input Node</td>
<td>Creates an input node for the selected node(s).</td>
</tr>
<tr>
<td>Make Output Node</td>
<td>Creates an output node for the selected node(s). See “Using output nodes” on page 124.</td>
</tr>
<tr>
<td>Show By Identifier</td>
<td>Shows the identifier instead of title of each object in the current diagram, edit table, <strong>Result</strong> window, or <strong>Outline</strong> view. Toggle to show title again.</td>
</tr>
<tr>
<td>Show With Values</td>
<td>Shows the mid values of the variable and all its inputs in each <strong>Object</strong> window. See “Showing values in the Object window” on page 22.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Opens the <strong>Attribute</strong> dialog to set the visibility of attributes and define new attributes. See “Managing attributes” on page 343.</td>
</tr>
<tr>
<td>Hide Definition(s)</td>
<td>Marks the currently selected node or module as hidden, so that their definitions are invisible. (Analytica Enterprise only)</td>
</tr>
<tr>
<td>Unhide Definition(s)</td>
<td>Unhides the currently selected node or module. This overrides any settings in parent modules to hide definitions. (Analytica Enterprise only)</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Edit Definition</strong></td>
<td>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 54).</td>
</tr>
<tr>
<td><strong>Edit Time</strong></td>
<td>Opens the Object window for the Time system variable. See “The Time index” on page 316.</td>
</tr>
<tr>
<td><strong>Paste Identifier</strong></td>
<td>Opens the Object Finder dialog for examining functions and variable identifiers, entering function parameters, and pasting them into definitions. See “Object Finder” on page 110.</td>
</tr>
<tr>
<td><strong>Show Invalid Variables</strong></td>
<td>Displays a window listing all variables with invalid or missing definitions. See “Invalid variables” on page 345.</td>
</tr>
<tr>
<td><strong>Math</strong></td>
<td>See “Math functions” on page 143.</td>
</tr>
<tr>
<td><strong>Array</strong></td>
<td>See Chapter 12, “Arrays and Indexes,” and Chapter 13, “More Array Functions.”</td>
</tr>
<tr>
<td><strong>Distribution</strong></td>
<td>Displays a list of unusual or less commonly used functions in the Special library.</td>
</tr>
<tr>
<td><strong>Statistical</strong></td>
<td>See “Statistical functions” on page 290.</td>
</tr>
<tr>
<td><strong>Operators</strong></td>
<td>Arithmetic, comparison, logical, and conditional operators. See “Operators” on page 140.</td>
</tr>
<tr>
<td><strong>System Variables</strong></td>
<td>System Variables submenu (see below).</td>
</tr>
<tr>
<td><strong>Matrix</strong></td>
<td>See “Matrix functions” on page 218.</td>
</tr>
<tr>
<td><strong>Text Functions</strong></td>
<td>See “Converting number to text” on page 145.</td>
</tr>
<tr>
<td><strong>Financial</strong></td>
<td>See “Financial functions” on page 233.</td>
</tr>
<tr>
<td><strong>Advanced Math</strong></td>
<td>See “Converting number to text” on page 145.</td>
</tr>
<tr>
<td><strong>Database</strong></td>
<td>Appears only in Analytica Enterprise. See “Database functions” on page 398.</td>
</tr>
<tr>
<td><strong>Optimizer</strong></td>
<td>Appears only if you have the Optimizer activated. See Optimizer Guide for more.</td>
</tr>
<tr>
<td>your libraries</td>
<td>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 21, “Building Functions and Libraries.”</td>
</tr>
</tbody>
</table>
### System Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnalyticaPlatform</td>
<td>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.”</td>
</tr>
</tbody>
</table>
| 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:  
  \[10K \cdot \text{Major} + 100 \cdot \text{Minor} + \text{SubMinor}\]  
  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 146. |
| Pi                     | The ratio of circumference to the diameter of a circle.                                                                                                                                                       |
| Run                    | The index for uncertainty sampling, defined as \(\text{Sequence}(1, \text{Samplesize})\).                                                                                                                   |
| SampleSize             | The number of sample iterations for probabilistic simulation. See “Uncertainty Setup dialog” on page 251.                                                                                                     |
| 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 285.                                                    |
| SvdIndex               | The \(\text{SingularValueDecomp()}\) function returns three matrices, \(U\), \(W\), and \(V\). To return all three at once, the return value is an array indexed by \(\text{SvdIndex}\), which is equal to \([U, W, V]\). |
| Time                   | The index variable identifying the dimension for dynamic simulation (the \(\text{Dynamic()}\) function). See “The Time index” on page 316.                                                                           |
| True                   | The logical (Boolean) constant that evaluates numerically to nonzero.                                                                                                                                         |
Result menu

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show Result</td>
<td>Opens a Result window for the selected object. See “The Result window” on page 26.</td>
</tr>
<tr>
<td>Mid Value</td>
<td>Displays the mid or deterministic value. See “Uncertainty views” on page 29.</td>
</tr>
<tr>
<td>Mean Value</td>
<td>Displays the mean of the uncertain value. See “Uncertainty views” on page 29.</td>
</tr>
<tr>
<td>Statistics</td>
<td>Displays statistics of the uncertain value in a table as set in the Uncertainty Setup dialog. See “Uncertainty views” on page 29.</td>
</tr>
<tr>
<td>Probability Bands</td>
<td>Shows probability bands (percentiles) as set in the Uncertainty Setup dialog. See “Uncertainty views” on page 29.</td>
</tr>
<tr>
<td>Probability Density</td>
<td>Displays a probability density graph for an uncertain value. For a discrete probability distribution, Probability Mass replaces this command.</td>
</tr>
<tr>
<td>Cumulative Probability</td>
<td>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 29.</td>
</tr>
<tr>
<td>Sample</td>
<td>Displays a table of the values determined for each uncertainty sample iteration. See “Uncertainty views” on page 29.</td>
</tr>
<tr>
<td>Graph Setup</td>
<td>Displays a dialog to specify the graphing tool, graph frame, and graph style. See “Graphing roles” on page 83.</td>
</tr>
<tr>
<td>Number Format</td>
<td>Displays a dialog to set the number format for displays of results. See “Number formats” on page 78.</td>
</tr>
<tr>
<td>Uncertainty Options</td>
<td>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 251.</td>
</tr>
</tbody>
</table>
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 74.

Set Node Style Displays **Node style** dialog to set arrow display and font for the selected node(s). See “Node Style dialog” on page 75.

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 73.

Align Selection To Grid Aligns selected node(s) to the diagram grid. See “Align to the grid” on page 69.

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 74.

Move Into Parent Moves the selected object from the current diagram to its parent diagram. See “The Object window” on page 20.

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 69.

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 69.

Edit Icon Opens a window to draw or edit an icon for the selected node. See “Adding icons to nodes” on page 126.

Align submenu

<table>
<thead>
<tr>
<th>Left Edges</th>
<th>Aligns left edges.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centers Left and Right</td>
<td>Aligns centers along the same horizontal line.</td>
</tr>
<tr>
<td>Right Edges</td>
<td>Aligns right edges.</td>
</tr>
<tr>
<td>Left and Right Edges</td>
<td>Moves and changes width so left and right edges are aligned vertically.</td>
</tr>
<tr>
<td>Top Edges</td>
<td>Aligns top edges.</td>
</tr>
<tr>
<td>Centers Up and Down</td>
<td>Aligns centers along the same vertical line.</td>
</tr>
<tr>
<td>Bottom Edges</td>
<td>Aligns bottom edges.</td>
</tr>
</tbody>
</table>

Make Same Size submenu

<table>
<thead>
<tr>
<th>Width (→, RIGHT)</th>
<th>Makes all nodes the same width.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height (→, DOWN)</td>
<td>Makes all nodes the same height.</td>
</tr>
<tr>
<td>Both (→, =)</td>
<td>Makes all nodes the same width and height.</td>
</tr>
</tbody>
</table>

Space Evenly submenu

<table>
<thead>
<tr>
<th>Across (=, A)</th>
<th>Spaces nodes evenly horizontally between leftmost and rightmost node.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Down (=, D)</td>
<td>Spaces nodes evenly vertically between top and bottom node.</td>
</tr>
<tr>
<td>Across, on grid  (=, A)</td>
<td>Spaces nodes evenly horizontally, ensuring all nodes are on the grid horizontally. May alter positions of left and right nodes.</td>
</tr>
<tr>
<td>Down, on grid    (=, D)</td>
<td>Spaces nodes evenly vertically, ensuring all nodes are on the grid vertically. May alter positions of top and bottom nodes slightly.</td>
</tr>
</tbody>
</table>
### Window menu

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bring To Front</td>
<td>Displays a list of the current windows; select one to display on top.</td>
</tr>
<tr>
<td>Close All But Active</td>
<td>Closes all windows except one, leaving the topmost window as the only window still open.</td>
</tr>
<tr>
<td>Show Memory Usage</td>
<td>Opens a window showing memory usage. See “Numbers and arrays” on page 428.</td>
</tr>
<tr>
<td>Show Page Breaks</td>
<td>Shows page breaks for the active diagram.</td>
</tr>
<tr>
<td>Cascade</td>
<td>Rearranges all open windows using a standard size, organized so that you can see the title bar of each one.</td>
</tr>
<tr>
<td>Tile Top to Bottom</td>
<td>Rearranges all open windows so that they fill the application window horizontally.</td>
</tr>
<tr>
<td>Tile Left to Right</td>
<td>Rearranges all open windows so that they fill the application window vertically.</td>
</tr>
</tbody>
</table>

### Help menu

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User guide</td>
<td>Opens the User Guide.</td>
</tr>
<tr>
<td>Optimizer</td>
<td>Opens the Optimizer Manual (only appears in Optimizer-enabled version of Analytica).</td>
</tr>
<tr>
<td>Tutorial</td>
<td>Opens the Analytica Tutorial.</td>
</tr>
<tr>
<td>Analytica Wiki</td>
<td>Jumps to the Analytica Wiki home page in your web browser, and logs you in automatically if your login credentials are stored. The Wiki is the main source for extended Analytica reference materials.</td>
</tr>
<tr>
<td>Wiki login info</td>
<td>Allows you to store or change your Analytica Wiki login and password, so that when you click on a link to a Wiki page (e.g., a “More information” link), Analytica can log you in automatically.</td>
</tr>
<tr>
<td>Email tech support</td>
<td>Opens your email system to send an email to Analytica Tech Support.</td>
</tr>
<tr>
<td>Contact Lumina</td>
<td>Provides contact information for Lumina.</td>
</tr>
<tr>
<td>Update license</td>
<td>Displays your current Analytica license information and allow you to update the license code.</td>
</tr>
<tr>
<td>About Analytica</td>
<td>Displays useful information such as the application’s edition, release number, your license code, and contact information.</td>
</tr>
</tbody>
</table>

**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 | Sends 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. |

Right mouse button menus

- Undo
- Cut
- Copy
- Paste
- Select All
- Duplicate Nodes
- Copy Diagram
- Preferences...
- Edit Definition
- Show Result
- Bring To Front
- Send To Back
- Align
- Make Same Size
- Space Evenly
- Set Diagram Style...
- Set Node Style...
- Show/Hide Color Palette
## Appendix C: Analytica Specifications

<table>
<thead>
<tr>
<th>Hardware and software</th>
<th>CPUs supported</th>
<th>Pentium or higher and equivalent AMD processors recommended</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Software</td>
<td>Windows XP, Vista, Windows 7, Server</td>
<td></td>
</tr>
<tr>
<td>Memory requirements</td>
<td>128 MB (512 MB+ recommended)</td>
<td></td>
</tr>
<tr>
<td>Application size</td>
<td>Approximately 6 MB</td>
<td></td>
</tr>
<tr>
<td>Maximum memory usable by Analytica process.</td>
<td>Analytica 64-bit: 128GB (limited by max pagefile size)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Analytica 32-bit:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4GB on Windows x64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3GB if /3GB flag added in C:\boot.ini</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2GB without /3GB flag in C:\boot.ini</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Objects</th>
<th>Number of system objects</th>
<th>738</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Maximum user-defined objects</td>
<td>31,900</td>
</tr>
<tr>
<td></td>
<td>Maximum number of local variables</td>
<td>No fixed limit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Uncertainty</th>
<th>Probability methods</th>
<th>Random Latin HyperCube</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Median Latin HyperCube</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Monte Carlo</td>
</tr>
<tr>
<td>Maximum sample size</td>
<td>99,999,999 for Analytica Enterprise, Optimizer, Power Player, and ADE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30,000 (other Editions)</td>
<td>limited by available memory</td>
</tr>
<tr>
<td>Random sampling methods</td>
<td>Minimal Standard</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L'Ecuyer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Knuth</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Numbers and arrays</th>
<th>Number precision</th>
<th>15 significant digits for floating-point numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>9 digits for integers</td>
</tr>
<tr>
<td>Maximum elements in a dimension</td>
<td>99,999,999 (Analytica Enterprise, Optimizer, Power Player, and ADE)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum dimensions in an array</td>
<td>15</td>
</tr>
</tbody>
</table>

### 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 pagefile 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. The memory usage displays in two sizes, depending on whether you check Expanded View. The expanded view contains a large number of memory statistics.
The above window appears automatically when Analytica runs low on memory.

Tip

Expanded view:

Maximum memory (RAM+pagefile) currently available for use/

Button appears when computing.

Link opens Analytica Wiki page describing memory statistics and what they mean.

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 pagefile.
   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. Analytica 64-bit will directly benefit from any RAM you add.

5. Add a solid-state drive to your computer and locate your page file on it.

6. 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.

To see all identifiers starting with the prefix “Con”:

3. Type List "^Con", followed by Enter.

Note: Any regular expression can appear between the quotes.
Appendices

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 54). 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 389 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.
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.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>INF</td>
<td>Infinity, such as 1/0.</td>
</tr>
<tr>
<td>NaN</td>
<td>Not A Number. Results from invalid functions such as Sqrt(-1), or 0/0.</td>
</tr>
<tr>
<td>NULL</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 428).
Appendices

Appendix F: Forward and Backward Compatibility

Backward compatibility
Models created in earlier releases of Analytica can be loaded, viewed, evaluated, and modified with Analytica 4.5. No file conversions are required.

There are many bug fixes included with every new release of Analytica, so if your model utilizes an undocumented feature that was really a bug, a change in model behavior could result.

Show result warnings
When you are trying a model for the first time in 4.5, 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, most potential backward-compatibility issues are reported to you.

Sort (collation) order
Analytica 4.5 uses regional sort order when comparing and sorting text, whereas Analytica 4.4 and earlier use pure ANSI (ASCII) order. Region awareness allows more intelligent handling of accented characters and other language subtleties. In the unlikely event that your model relies on ANSI ordering, you can configure it to use ANSI ordering in Analytica 4.5 as follows: press F12 to open the typescript window and type: TextLocale:ANSI.

Character codes
Every character has a numeric character code associated with it, which is used by the Chr() and Asc() functions. The characters associated with codes from 128 to 160, 173, 178 and 179 have changed to be consistent with the Unicode standard. Your model is only impacted by this if it relies on the specific numeric value passed to Chr() or received from Asc(). If uses these characters in text or attributes, the conversion in handed automatically or you will see no difference.

Recommended Modifications
After you upgrade to Analytica 4.5, there are several setting changes we recommend you make to your model, although these are optional. Analytica will initially configure these to operate a legacy mode to minimize the chance of introducing a difference when you start using your model in the new release; however, legacy modes prevent you from taking full advantage of improvements and fixes.

ClearType fonts
Analytica 4.4 and later can render influence diagram nodes using ClearType fonts, which are smooth anti-aliased fonts that look nicer than the fonts used pre-4.4. To turn on ClearType fonts in legacy models, open the Set Diagram Style... dialog for your top-level model diagram from edit mode and check the ClearType fonts checkbox (the checkbox will not be present for models started in Analytica 4.4 or later, which already use ClearType). Then visit each module and adjust any nodes that word wrap in an undesirable fashion. For most diagrams, you can do this by pressing Ctrl+A (select all) and then pressing Ctrl+T (Adjust size) multiple times until an acceptable appearance is obtained. With ClearType support, Ctrl+T cycles through multiple criteria for adjusting node sizes.

Domain as self index
You should turn off the Domain acts as self index preference. A small percentage of models created in Analytica 4.2 or earlier might be impacted, and only in those cases should this preference be on, at least until you update the model appropriately. The older domain treatment is inferior in many ways, and some features are only available when this preference is off.

Proactive evaluation
We recommend that you turn off the Proactively evaluate indexes preference. This will generally speed up the time it takes to load your model. This preference is on in models created initially in 4.3 or earlier.

Forward compatibility
It is also possible to run models created or edited in Release 4.5 in earlier releases of Analytica, such as Analytica 4.4, 4.0 or even 3.1, provided you don’t rely on functions, features, characters, or functionalities new to Analytica 4.5. The models load into earlier releases of Analytica, although they might encounter problems during parsing or evaluation in the places where 4.5 features are used. A few 4.5 features might be stripped out of the model if it is re-saved from an earlier
release, such as values for system attributes or system variables that don’t exist in the earlier version.

**UTF-8 file format** Models that contain Unicode characters (characters not present in the ASCII, or ISO-8859-1, character set, generally those with character codes above 255, including Chinese, Japanese, Korean, Cyrillic, Greek and other alphabets and esoteric mathematical and graphical symbols) are saved using the UTF-8 character encoding, whereas Analytica 4.4 and earlier assume model files to be ASCII-encoded. These UTF-8 files will load into Analytica 4.4 and earlier, but extended characters (any character with an ASCII code above 127) become mangled into two or three characters. The user is warned about the problem when the model is loaded into the earlier release.

This is not a problem if you have not used any Unicode characters in your model, since in that case, Analytica 4.5 will save your model using the ASCII encoding. If your model does utilize Unicode characters, then you need to tell your users that 4.5 is required, since Unicode support is new to Release 4.5.
Appendix G: Bibliography


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!


<table>
<thead>
<tr>
<th>Function List</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic Math</strong></td>
</tr>
<tr>
<td>Abs, Sign, Mod, Sqr, Sqrt, Exp, Ln, Logten, Round, Ceil, Floor, Factorial, Radians, Degrees, Sin, Cos, Tan, Arctan</td>
</tr>
<tr>
<td><strong>Advanced Math</strong></td>
</tr>
<tr>
<td>Arccos, Arccsin, Arctan2, Bessel*, BetaFn, BetaI, BetaInv, BetaIInv, Binomial, Combinations, Cosh, CumNormal, CumNormalInv, Erf, ErfI, ErfInv, GammaFn, GammaI, GammaIInv, Lgamma, Logit, Permutations, Sign, Sinh, Tanh</td>
</tr>
<tr>
<td><strong>Complex numbers</strong></td>
</tr>
<tr>
<td>Abs, ComplexDegrees, ComplexRadian, Conjugate, imaginaryPart</td>
</tr>
<tr>
<td><strong>Creating Arrays</strong></td>
</tr>
<tr>
<td>[ ... ] m...n, Array, CopyIndex, DetermineTable, IntraTable, Sequence, SubTable, Table</td>
</tr>
<tr>
<td><strong>Array Reducing</strong></td>
</tr>
<tr>
<td>Area, Argmin, Argmax, Average, Max, Min, Product, Subindex, Sum, CondMin, CondMax, PositionInIndex</td>
</tr>
<tr>
<td><strong>Transforming Arrays</strong></td>
</tr>
<tr>
<td>Aggregate, CumProduct, Cumulate, Dispatch, Integrate, Normalize, Rank, Sort, SortIndex, Uncumulate</td>
</tr>
<tr>
<td><strong>Selecting from Arrays</strong></td>
</tr>
<tr>
<td>x[i=v], x[i&lt;&gt;n], x[Time=n], Choice, Slice, Subscript</td>
</tr>
<tr>
<td><strong>Interpolation</strong></td>
</tr>
<tr>
<td>CubicInterp, LinearInterp, Steplinear</td>
</tr>
<tr>
<td><strong>Sets</strong></td>
</tr>
<tr>
<td>SetContains, SetDifference, SetIntersection, SetsAreEqual, SetUnion</td>
</tr>
<tr>
<td><strong>Other Array Functions</strong></td>
</tr>
<tr>
<td>Concat, ConcatRows, FFT, FFTInv, Size, Subindex, Subset, Unique, Rank, Indexes0f, IndexNames, IndexValue</td>
</tr>
<tr>
<td><strong>Relational to Array conversions</strong></td>
</tr>
<tr>
<td>MDArraYToTable, MTable</td>
</tr>
<tr>
<td><strong>Matrix Functions</strong></td>
</tr>
<tr>
<td>Decompose, Determinant, DotProduct, Invert, MatrixMultiply, Transpose, EigenDecomp, SingularValueDecomp</td>
</tr>
<tr>
<td><strong>Operators</strong></td>
</tr>
<tr>
<td><strong>Complex Numbers</strong></td>
</tr>
<tr>
<td>Abs, ComplexDegrees, ComplexRadian, Conjugate, imaginaryPart</td>
</tr>
<tr>
<td><strong>Excel Access Functions</strong></td>
</tr>
<tr>
<td>SpreadsheetOpen, SpreadsheetSave, SpreadsheetCell, SpreadsheetRange, SpreadsheetInfo, SpreadsheetSetCell, SpreadsheetSetRange, SpreadsheetSetInfo</td>
</tr>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<tr>
<td>IsHandle, IsNaN, IsNull, IsNumber, IsRealNumber, IsReference, IsText, IsUndefined, TypeOF</td>
</tr>
<tr>
<td><strong>System Variables</strong></td>
</tr>
</tbody>
</table>
Function List

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.
DefineOptimization, OptFindIIS, OptObjectiveSA, OptObjective, OptRead, OptSolution,
OptStatusNum, OptStatusText, OptWrite, OptWriteIIS
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 50.

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 6.

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 6 and Chapter 23, “Analytica Enterprise” on page 391.

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 6.

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 6.

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 6.

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 6.

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 6.

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 150.

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 47.

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 47.

Attribute | A property or descriptor of an object, such as its title, description, definition, value, or inputs. See “Managing attributes” on page 343.

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 20.

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 44.

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 37.

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
Glossary

Browse-only model with Analytica Enterprise. Browse-only models are also encrypted, meaning that the model file is encrypted and not readable or editable. See “Making a browse-only model and hiding definitions” on page 404.

**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 19.

**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 17.

**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 117.

**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 17.

**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 267.

**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 270 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 246.

**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 44.

**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 32.

**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 393.

**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 17.

**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 103.

**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 404.

**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 355.

**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 216.

**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 29.

**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 17.

**Determinable**  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 150.

**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 261.

**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 246.

**Domain**  The possible outcomes of a variable. It indicates whether an uncertain numeric variable is discrete or continuous. It defines the ordering of labels (when not alphabetical) as they appear on graph axes. The possible outcomes are used for model validation to catch errors that might get introduced into the model. It determines the possible values presented by the `Choice()`, `Probtable()`, and `Determinable()` functions. In structured optimization problems, it defines a decision’s bound and search space. When a PDF or CDF in generated, it is used to disambiguate whether a variable with numeric outcomes is discrete numeric or continuous. When graphing categorical (e.g., textual) outcomes, the domain can be used to define an ordering among the possible values. Several domain types (`Continuous`, `Integer`, `Boolean`, `Discrete`) specify the general type, possibly with bounds but without an explicit enumeration of values, while others types (`Explicit Value`, `Copy from index`) list the possible values. The domain can be computed using an arbitrary `Expression`, which should return with a call to one of the functions `Continuous()`, `Discrete()`, or `Integer()`, or `GroupedInteger()`. A `GroupedInteger` domain type is used to express exclusivity constraints in integer optimization problems.

**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 393.

**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 315.

**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 176 and “Editing a table” on page 178.

**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 176.

**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 45.

**Encrypted**  Saved in a non-human-readable form. Encryption provides a mechanism for protecting intellectual property. Analytica Enterprise users can distribute encrypted copies of their models to their end-users. In Analytica, encryption 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 404.

**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 137.

**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
Glossary

Analytica User Guide 443

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 345.

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 345.

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, I)” on page 292.

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 17.

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 28.

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 83.

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 46.

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 297.

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 150.

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 27.

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 17.

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 48.
Glossary

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 20.

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 189 and “Table(i1, i2, … in) (u1, u2, u3, … um)” on page 191.

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 122.

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 48.

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 122.

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 166.

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 83.

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 292.

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 361.

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 169.

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 169.

Matrix  A two-dimensional array of numbers with indexes of equal length. See “Matrix functions” on page 218.

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 291.

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 29.

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 32.

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 14.
Glossary

**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 17.

**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 71 and “Show module hierarchy preference” on page 340.

**Multimodal distribution**  A probability distribution that has more than one mode. See “How many modes does it have?” on page 247.

**Multivariate distribution**  A joint probability distribution defined simultaneously over two or more scalar quantities, as contrasted with a “Univariate distribution” that describes uncertainty for a single scalar quantity. The quantities in a multivariate distribution are usually statistically dependent (e.g., are correlated in some way). Standard parametric distribution functions for multivariate distributions are available in Analytica by using Add Library... on the file menu to add the “Multivariate Distributions.ana” library.

**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 17.

**Normal distribution**  The bell-shaped curve, also known as a Gaussian distribution. See “Normal(mean, stddev)” on page 271.

**Obfuscated**  A synonym for encrypted, or saved in a non-human-readable form. Use of this term within Analytica has been replaced by the term encrypted. Earlier releases had used this term.

**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 17.

**Object Finder**  A dialog used to browse and edit the functions and variables available in a model. See “Object Finder dialog” on page 110.

**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 20.

**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 17.

**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 392.

**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 328.

**Operator**  A symbol, such as a plus sign (+), that represents a computational process or action such as addition or comparison. See “Operators” on page 140.

**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 189 and “Table(i1, i2, … in) (u1, u2, u3, … um)” on page 191.

**Outline window**  A view of a model that lists the objects it contains as a hierarchical outline. See “The Outline window” on page 341.

**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 124.
## Glossary

**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 48.

**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 124.

**Parameters**  
Values or expressions passed to a function, in parentheses after the function name, sometimes termed arguments. See “Function calls and parameters” on page 143.

**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 \( p \)th percentile. Compare to “Fractile.” See “GetFract(x, p, l)” on page 292.

**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 \( \text{run} \). See “Probabilistic calculation” on page 250.

**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 254.

**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 32.

**Probability distribution**  
A probability distribution describes the relative likelihood of a variable having different possible values. See “Probability distributions” on page 260 and “Probabilistic calculation” on page 250.

**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 32.

**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 266.

**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 29.

**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 192.

**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 16.
<table>
<thead>
<tr>
<th><strong>Glossary</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Result view</strong></td>
</tr>
<tr>
<td><strong>Sample</strong></td>
</tr>
<tr>
<td><strong>Sampling method</strong></td>
</tr>
<tr>
<td><strong>Scalar</strong></td>
</tr>
<tr>
<td><strong>Scatter plot</strong></td>
</tr>
<tr>
<td><strong>Self</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Sensitivity analysis</strong></td>
</tr>
<tr>
<td><strong>Side effects</strong></td>
</tr>
<tr>
<td><strong>Skewed distribution</strong></td>
</tr>
<tr>
<td><strong>Skewness</strong></td>
</tr>
<tr>
<td><strong>Slice</strong></td>
</tr>
<tr>
<td><strong>Slicer</strong></td>
</tr>
<tr>
<td><strong>Splicing</strong></td>
</tr>
<tr>
<td><strong>SQL</strong></td>
</tr>
<tr>
<td><strong>Standard deviation</strong></td>
</tr>
<tr>
<td><strong>Suffix notation</strong></td>
</tr>
<tr>
<td><strong>Symmetrical distribution</strong></td>
</tr>
<tr>
<td><strong>System function</strong></td>
</tr>
<tr>
<td><strong>System variable</strong></td>
</tr>
<tr>
<td><strong>Glossary</strong></td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td><strong>Table</strong></td>
</tr>
<tr>
<td><strong>Tail</strong></td>
</tr>
<tr>
<td><strong>Title</strong></td>
</tr>
<tr>
<td><strong>Uncertain value</strong></td>
</tr>
<tr>
<td><strong>Uniform distribution</strong></td>
</tr>
<tr>
<td><strong>Univariate distribution</strong></td>
</tr>
<tr>
<td><strong>Units</strong></td>
</tr>
<tr>
<td><strong>User-defined function</strong></td>
</tr>
<tr>
<td><strong>Value</strong></td>
</tr>
<tr>
<td><strong>Variable</strong></td>
</tr>
<tr>
<td><strong>Variance</strong></td>
</tr>
</tbody>
</table>
- (subtraction) operator 140
^ (exponentiation) operator 140
:: (scoping) operator 141
:= (assignment) operator 367
.. (sequence) operator 173
... (list) operator 172
@ (position) operator 183
* (multiplication) operator 140
\ (reference) operator 358, 378
& (concatenation) operator 226
# (dereference) operator 222, 378
+ (addition) operator 140
< (less than) operator 140
<= (less than or equal to) operator 140
<> (not equal) operator 141
= (equal) operator 140
> (greater than) operator 140
>= (greater than or equal to) operator 140

A
About Analytica command 426
Abs() function 143, 232
accept button 106
Across command 425
Across, on grid command 425
Add Library command 419
Add Module command 419
Add newline within cell 179
Adjust size 68
Adjust Size command 425
Advanced Math command 422
Aggregate() function 212
aliases
  compared to original 51
  creating 50
  definition 440
  illustration 50
  input nodes 52, 122
  output nodes 52, 124
  vs. originals 52
Align Selection to Grid command 425
Align submenu 425
All qualifier 358
alphabetical ordering, text 141
.ana file extension 14
Analytica Cloud Player 132, 419
  group plans 132
  individual accounts 132
Analytica Decision Engine, description 440
Analytica Enterprise, description 440
Analytica Player, description 440
Analytica Power Player, description 440
Analytica Professional, description 440
Analytica Trial, description 440
Analytica Wiki 10
  Help menu command 426
Analyticaployment system variable 423
Analyticaplatform system variable 423
Index

Analytica version system variable 423
Append Cells command 420
Append Columns command 420
Append Rows command 420
application integration 412
Arcos() function 231
Arcsin() function 231
Arctan() function 145
Arctan2() function 231
Area() function 197
Argmax() function 195
Argmin() function 195
arithmetic operators
  array abstraction 153
  meanings 140
Array command 422
Array() function 189
arrays
  abstraction, see intelligent array abstraction
  changing index of 190
  combining 153–158
  defining 190
  definition 440
  ensuring abstraction 374
  example variables 188
  functions 189
  IF a THEN b ELSE c 160, 167
  matrices 218
  modeling 149–186
  multidimensional 197
  one-dimensional 153, 155, 157, 336
  qualifiers 357
  reducing functions 192
  removing indexes 177
  scalar 153
  serial correlation 284
  slicing 181
  subarrays 181
  three-dimensional 158, 337
  two-dimensional 157, 158, 337
  using in Dynamic() 321
  values 23
  working with 150–167
  zero-dimensional 153
arrows
  across windows 49
  alias nodes 50
  arranging 68
  arrow tool 18, 440
  automatically drawn 47
  between modules 49, 51
  bold 347
  creating 47
  definition 440
  display settings 74
  double-headed 49
  drawing 47–51
  dynamic 323
  gray 48
  hiding and unhiding 47, 70, 74
  illustrations 48
  influence cycles 48
  input 444
  keyboard shortcuts 186
  linking to module nodes 48
  outputs 446
  removing 47
  small arrowhead 49
Asc() function 224
AskMsgChoice() function 387
AskMsgNumber() function 387
AskMsgText() function 387
assignment operator 367
associational correspondence 181
atoms
  about 150
  array abstraction 375
  qualifier 357
  values 23
Attrib Of Ident 344
Attribute panel
  closing 22
  definition 440
  using 20
attributes
  canceling edits 52
  controlling display 343
  copying and pasting 328
  creating 344
  definition 440
  displaying 344
  displaying check attribute 118
  domain 265
  editing 52
  functions 343, 355
  managing 343–344
  modules 343
  renaming 344
  user-created 343
  variables 343
Attributes command 421
Attributes dialog 118, 343
authors
  adding 44
  attribute 343
  definition 440
Average() function 194
axes, display settings 88
B
  background printing 24
  backward compatibility 434
Balloon help 15
bar graphs 87
behavior analysis
  definition 440
  overview 38
  results 39
  understanding 41
Bernoulli() function 261
Bessel() function 231
Bessel functions 231
Bessel K() function 231
BesselJ() function 231
BesselY() function 231
Beta() distribution function 272
BetaFn() function 241
Beta() function 241
Betaaln() function 241
Betaalninv() function 241
bibliography 436
binding precedence, operators 142
Binomial() function 261
BiNormal() distribution function 283
Boolean
  number format 79
  operators 141
  values 139
  variables 247
Boolean() function 117
Both command 425
Cascade command 426
categorical plots 91
CDF() function 295
Ceil() function 143
Ceil() function 231
cells
  adding 171, 179
  copying and pasting 178
  deleting 171, 179
  editing 178
  selecting 178
Centers Left and Right command 425
Centers Up and Down command 425
Certain() distribution function 279
chance variables
  definition 441
  representation 17
Chancedist() function 269
Change Picture Format command 425
charts, selecting type 86
check attribute
  defining 119
  definition 441
  displaying 118
  edit table cells 119
  failure 120
  features 343
  triggering 119
ChiSquared() distribution function 277
choice menus 184
Choice option 123
Choice() function 183
Chr() function 224
Class attribute 343
classes
  changing for objects 53
  definition 441
Clear command 420
ClearType fonts 69
cloaking 441
Close All But Active command 426
Close command 419
Close Model command 419
Coerce qualifier 359
colors
  adding to influence diagrams 73
  background 74, 90
  changing 73
  displaying in nodes 75
  graphs 89
  grouping nodes 73
  input and output nodes 125
  opening palette 73
  screenshots 76
columns
  adding and deleting 179
  display significance 156
  separating 395
  trading places with rows 156
Combinations() function 241
comments in definitions 105
comparison operators 140
compatibility, backward and forward 434
Index

complex conjugate 233
Complex number functions 232
Complex number, magnitude. 232
complex numbers, enabling 232
ComplexDegrees() function 232
ComplexRadians() function 232
CompressMemoryUsedBy 352
computation time 416
ComputedBy() function 367
Concat() function 212
concatenation operators 226
ConcatRows() function 213
conditional dependencies
  creating 267
  definition 441
CondMax() function 195
CondMin() function 195
confidence intervals 416, 417
conglomeration functions 205
Conj() function 233
conjugate transpose 220
cache processes, running 412
constants
  definition 441
  representation 17
Constraint, display of result 82
constructs, programming 365
Contact Lumina command 426
correlation qualifier 356
ContextSample qualifier 357
continuous distributions
  definition 441
  overview 247
continuous plots 91
continuous variables 441
Continuous() function 116
controls, resizing 125
conventions, typographic 10
Copy command 420
Copy Diagram command 420
Copy Table command 420
CopyIndex() function 175
Correlate_dists() distribution function 283
Correlate_with() distribution function 282
Correlation() function 293
correspondence types 181
Cos() function 145
Cosh() function 232
CostCapme() function 239
CostCapmm() function 239
Covariance() function 293
Created attribute 343, 441
cross-hatching pattern
  switching off 76
  use of 56, 107
CubePlan 8
Cubicinterpl() function 208
Cumdist() distribution function 277
Cumipmt() function 233
CumNormal() function 241
CumNormalInv() function 242
Cumprinc() function 234
Cumproduct() function 199
cumulative probability
  distribution 441
  options 256
  uncertainty view 32
Cumulative Probability command 424
CumWilcoxon() function 263
CumWilcoxonInv() function 263
currency symbols, showing 79
CurrentDataDirectory() function 388
CurrentModelDirectory() function 388
current dependencies 48, 443
cycles, influence 48, 443
cycles, influence 48, 443

D
data
  copying and pasting 328
  copying diagrams 328
  identifying source 393
  import/export format 336
  importing and exporting 327–338
  numerical 338
  OLE linking 328
  pasting from programs 328
  pasting from spreadsheets 328
  source 441
  structures 378
Data Source Name (DSN)
  definition 442
  using 393
Database command 422
Database library functions 398
databases
  configuring DSN 394
  writing to 396
datatype functions 148
date formats
  arithmetic 82
  date numbers 82
  date origin 80, 82
  interpreting 81
  letter codes 81
  literal text 81
  range of dates 82
  settings 80
  type description 79
date functions 228
Date values, entering 139
DateAdd() function 230
DatePart() function 229
DBLabels() function 398
DBQuery() function 398
DBTable() function 399
DBTableNames() function 399
DBWrite() function 396, 399
Index

decision variables
  arranging nodes 68
  definition 441
  identifying 59
  representation 17
Decompose() function 221
defaults
  changing global 94
  views 27
Definition attribute 343
Definition button 18
Definition menu
  overview 422
  pasting from a library 112
definitions
  about 441
  adding identifiers 106
  alphabetical list 439–448
  changes to influence diagrams 107
  changing 107
  comments in 105
  creating 104–106, 124
  cross-hatching 107
  editing 104–106, 112
  exporting 335
  hidden, see hidden definitions
  hiding 404, 441
  importing 334
  including probability distributions in 250
  incomplete 345
  inheritance 404
  invalid or missing 422
  syntax check 106
  unhiding 404
  updating arrows 107
  using 356
  working with 104
Degrees() function 145
Delete Columns command 420
Delete Rows command 420
dependencies
  conditional 267
  cyclic 48, 443
  Dynamic() function 322
depreciation 159
dereference operator 378
Description attribute 343
descriptions
  definition 441
  overview 130
  using 356
Determinant() function 221
deterministic tables
  converting Table to DetermTable 217
  definition 215, 441
  equivalent of using Subscript 218
  expression view 217
  in parametric analysis 217
  relation to ProbTable 268
  splicing 180
  used with discrete distributions 268
  working with 215–217
deterministic values 441
deterministic variables 441
DetermTable() function 215, 268
determinables, see deterministic tables
Diagram menu 425
Diagram Style dialog 74
Diagram window
  description 15
  maximum number of 349
diagrams
  see also influence diagrams
    adding frames 128
    adding graphics 128
    adding text 128
    copying 328
    editing 17, 45
    exporting as image 76
    exporting to image file 328
    keeping compact 71
    opening details 16
    organizing 68, 71
    overriding default styles 75
    tornado 301
dialog functions 385
digits, setting maximum number 79
dimensions
  adding to tables 156
  definition 442
  modeling arrays and tables 150
  reducing 370
Dirichlet() distribution function 283
discrete probability distributions
  creating 269
  definition 442
  vs. continuous 247
discrete variables
  definition 442
  domain attribute 265
  logical and Boolean 247
  probability tables 266
Discrete() function 117
discretizing process 247
Dispatch() function 201
Display dates as numbers 82
Display numbers as dates 82
DIST standard 312
Dist_additive_growth() distribution function 285
Dist_compound_growth() distribution function 285
Dist_reshape() distribution function 283
Dist_serial_correl() distribution function 285
Distributing nodes 70
Distribution button 19
Distribution command 422
Distribution library 260
distributions
  arrays with serial correlation 284
  continuous 441
  correlation or covariance matrix 283
Index

creating dependencies 282
custom continuous 277
exponential 274
gamma 274
gaussian 274
logistic 274
lognormal 271
multivariate 281
normal 445
parametric continuous 270
parametric discrete 261
symmetric vs. skewed 248
transformed beta 273
uniform 262, 270, 448

Domain acts as self index preference 56
Domain attribute 113
domain attributes
  classes 343
  editing 265
  types 265
  use of Index domains 216
  use with DetemTable 216
  viewing in Object window 266
  working with 265
Domain functions 116
Domain, Automatic 114
Domain, Boolean 114
Domain, Continuous 114
Domain, Copy From Index 115
Domain, Discrete 115
Domain, Explicit Values 115
Domain, Expression 115
Domain, Grouped Integer 114
Domain, Integer 114
Domain, semi-continuous 116
domains 442
dot operator 374
dot product 219
Down command 425
Down, on grid command 425
DRIVER attribute 393
DSN, see Data Source Name (DSN)
Duplicate Nodes command 420
Dydx() function 300
dynamic arrows, showing or hiding 323
dynamic loops 323
dynamic models 68
dynamic simulation 316–324
dynamic variables
  definition 442
  initial values 321
  working with 316
Dynamic() function 316–324

E
Edit Definition command 422
Edit Icon command 425
Edit menu 420
Edit Table button 19
Edit Table window 178

edit tables
  see also tables
  adding cells 179
  adding dimensions 156
  blank cells 191
  choice menus 184
  clickable titles 173
  comparing results 34
  copying 179, 328
  copying and pasting cells 178
  creating 152, 191
  date formats 81
  defining 153
  defining variables as 176–178
  definition 442
  deleting cells 179
  display 123
  editing 178–180
  extending indexes 163, 179
  import and export formats 336
  keyboard shortcuts 185
  OLE linking 334
  pivoting 3D 158
  pivoting rows and columns 156
  saving recover info 55
  selecting cells 178
  splicing 180
  totals 162
  two with the same index 154
  using spreadsheet data 178
  working with 178
Edit Time command 422
edit tool
  about 18
  definition 442
  using 45
edits, canceling 52
EigenDecomp() function 221
Elasticity() function 300
Email tech support command 426
EnableComplexNumbers 232
encrypted copies 404
Encrypted, definition 442
Enlarge attribute & table fonts preference 55
Erf() function 242
ErfInv() function 242
Error() function 387
errors
  avoiding out-of-range 369
  custom messages 120
  displaying warnings 56
  evaluation 387, 433
  factor 272
  fatal 433
  invalid number 433
  lexical 432
  message types 432–433
  naming 432
  out of memory 433
  syntax 432
Index

system 433
warnings 147
Evaluate() function 388
EvaluateScript() function 410
evaluation errors 387, 433
evaluation mode qualifiers 356
Example models 15
Exit command 419
Exp() function 144
expected value
definition 444
using 31
expit function 242
exponent number format 79
exponential distribution 274
Exponential() distribution function 273
Export command
exporting images 76
menu item description 419
export format 336
expr (Expression) popup menu 107, 176
Expression assist 105, 108
Expression assist, enabling or disabling 108
Expression assist, function parameter info 109
Expression assist, identifier completion 109
Expression assist, preference 55
expression view 170
expressions
Boolean values 139
conditional 142
definition 442
entering probability distributions as 249
exception values 146
importing definitions 334
in dynamic loops 323
listing 108
number formats 137, 138
parenthesis matching 105
subscript constructs 159
syntax 142
text values 139
truth values 139
types 108, 442
using 137–148
variable definitions 138

F
Factorial() function 145
False system variable 139, 423
fast Fourier transform 209
fatal errors 433
FFT() function 209
FFTInv() function 209
File Info attribute 343, 443
File menu 419
filed libraries
about 361
adding to models 346, 361
creating 345
definition 443
locking 345
removing from models 346
representation 53
saving 346
working with 345
filed modules
adding to models 346
creating 345
definition 443
locking 345
removing from models 346
representation 53
saving 346
working with 345
files
changing locations 330, 334
Financial command 422
financial functions 233
Financial library functions 238
Find command 421
Find dialog 342
Find Next command 421
Find selection 342
Find Selection command 421
FindinText() function 224
fixed point number format 79
Floor 231
Floor() function 144
font settings
graphs 89
nodes 74, 75
Font size preference 55
For loops 369
For...Do function 369
Form class 53
form modules
adding nodes 126
creating 125
working with 125
forward compatibility 434
fractiles
definition 443
estimating confidence intervals 417
frames, adding to diagrams 128
Frequency() function 294
functions
about 17
array 212
array-reducing 192–197
attributes 355
categories 437
conglomeration 205
creating 355
custom discrete probabilities 266
datatype 148
date 228
dialog 385
financial 233
Financial library 238

Analytica User Guide 455
Index

interpolation 207
list of 437
math 143, 231
matrix 218–220
miscellaneous 388
name-based calls 143, 355
pasting 106
position-based calls 143, 355
probability 241
recursive 356, 372
special probabilistic 279
statistical 290
system 447
text 224
transforming 197–201
using 355
Fv() function 234
Gamma() distribution function 274
GammaFn() function 242
Gamma() function 242
GammaInv() function 242
Gaussian probability distributions 271
Gaussian() distribution function 283
general indexes 382
general variables
definition 443
representation 17
generalized linear regression 307
Geometric() distribution function 262
GetFract() function 292
GetRegistryValue() function 389
global defaults, modifying 94
gotolink Expressions.fm
TextLocale 437
Graph Setup command 424
Graph Setup dialog
Axis Ranges tab 88
Background tab 90
Chart Type tab 86
opening 86
Preview tab 91
Style tab 88
Text tab 89
using 85–91
graphics
adding frames 128
adding to diagrams 128
converting image formats 128
exporting diagrams 76
exporting graphs 91
Legacy Bitmap format 128
taking screenshots 76
graphing roles
definition 443
working with 83
graphs
3D effects 87
axis settings 88
bar style 87
changing global default 94
combining settings 93
comparing results 34
converting from tables 28
creating templates 92
customizing 29
definition 443
display settings 88
displaying 28
exporting 328
exporting as image files 91, 328
features 29
line style settings 86
modifying templates 93
plotting methods 91
previewing 91
renaming templates 94
scatter 306
unlinking templates 92
using templates 92
XY 95, 304
grid, aligning to 69
GroupId() function 117
Handle qualifier 358
Handle() function 382
HandleFromIdentifier() function 382
handles
edit tables 173
functions 382
indexes of 382
using 382
viewing 382
Height command 425
Help attribute 343
Help balloons 55
Help menu 426
hexagons 17
hidden definitions
creating 404
description 441
inheritance 404
making unviewable 406
unhiding 404
using 404
Hide Definition(s) command 421
Histogram method for probability density plots 255
hourglass cursor 16
Hypergeometric() distribution function 262
hyperlinks, using 130
Icon window, opening 126
icons
adding to nodes 126
drawing 127
Index

editing 127
Ident(Time-k) function 318
Identifier completion, Expression assist 109
identifiers
  changing 54
  copying 50
  definition 443
  listing those in use 431
  name format 431
  naming 432
  using 46, 106, 355
Identifiers attribute 343
identity matrix 220
IF a THEN b ELSE c
  in arrays 160, 167
  using 142
ignoreNaN parameter 193
ignoreNonNumbers parameter 193
IgnoreWarnings() function 389
image files
  see also graphics
    exporting diagrams as 76
    exporting graphs as 91
ImPart() function 233
Implied_volatility_c() function 239
Implied_volatility_p() function 240
Import command 419
import format 336
importance analysis 443
importance weighting
  setting up 286
  using 285
Index button 180
index position operator 183
Index qualifier 357
index variables 443
indexes
  adding 180
  adding items 163
  automatic updates 164
  by name not position 181
  changing on arrays 190
  correspondence types 181
  creating 151, 169, 177
  defining 398
  definition 443
  dialog box features 177
  example variables 188
  expanding 163, 179
  functions 172, 214
  general 382
  handles 382
  iterating with For and Var 378
  label 392
  local 373
  meta-indexes 382
  modeling 149–186
  OLE linking 331
  omitting parameters 377
  position operator 183
propagating without changing definitions 158
recognizing nodes 17
record 392
reducing when unused 193
removing from arrays 177, 180
selection area 27, 443
self 162, 173
sequence of numbers 154
splicing 180
sums 158
SvdIndex 222
working with 150–167
IndexesOf() function 214, 383
IndexNames() function 214
IndexValue() function 214
INF 146
infinity 146
influence arrows, see arrows
influence cycles 48, 443
influence diagrams
  see also diagrams
    automatically updating 107
    coloring 73
    copying 328
    creating 67–71
    decision variables 59
    definition 444
    examples of good and bad 66
    overview 15
    screenshots 76
    using to create models 58
innermost dimension 444
input nodes
  browsing 19
  creating 123
  definition 444
  original variables 125
  popup menus 123
  resizing 125
  using 122–124
  viewing 19
input variables, date formats 81
inputs
  arrowhead 444
  attribute 343
  examining results 39
  listing 16
  remote 16
  varying 39
Insert Cells command 420
Insert Columns command 420
Insert Rows command 420
Insufficient memory 350
integer number formats 79, 138
Integer() function 117
Integrate() function 200
integration with other applications 412
intellectual property, protecting 404
intelligent array abstraction
  arithmetic operators 153
Index

definition 444
dimensional reduction 370
equation 50
ensuring 374
exceptions 166
financial functions 233
IF THEN ELSE 376
omitted index parameters 377
summary 166
tornado diagrams 303
vertical 358
Intelligent Arrays
  about 150
  main principles 166
  Monte Carlo sampling 165
  probability distributions 281
interpolation functions 207
IntraTable() function 192
invalid variables 345
inverse logit function 242
Invert() function 220
Ipmt() function 234
Irr() function 234
IsNaN() function 148
IsNotSpecified() function 360
IsNumber() function 148
IsReference() function 148
IsResultComputed() function 389
Issampleevalmode system variable 423
IsText() function 148
IsUndef() function 148
Iterate function 371

J
JoinText() function 226

K
Kernel Density Estimation (KDE) 256
Kernel density smoothing 256
key combinations for editing 105, 185
key icon 21
key, in graphs 444
Knuth random number generator 253
kurtosis 444
Kurtosis() function 292

L
L’Ecuyer random number generator 253
labels
  index 392
  listing 108
Large memory usage 350
Large-scale sampling 352
Last Saved attribute 343, 444
Left and Right Edges command 425
Left Edges command 425
Legacy Bitmap format 128
lexical errors 432
Lgamma() function 232
libraries
  adding 346
  creating 361
custom 422
Database 398
definition 444
Distribution 250
Distribution System 108
embedding 347
encrypting 406
filed, see filed libraries
Financial 238
linking to original 347
Multivariate Distributions 282
Performance Profiler 411
representation 53
selecting 111
Special 422
Text 224
Trash 47
user 422
user-defined functions 361
using 354, 361
line style in charts, selecting 86
linear regression 307
Linearinterp() function 208
linked modules
  embedding in model file copy 132
List button 19
List buttons 123
list view 170
ListOfHandles() function 382
lists
  adding cells 171
  autofilling 170
  creating 38
  defining time 319
  definition 444
  deleting cells 171
  editing 171
  labels 444
  mixing numbers and text 170
  navigating 171
Ln() function 144
local indexes
  about 373
  MetaIndex declaration 383
local variables
  assigning 367
  assigning slices 368
  declaring 366
LocalAlias..Do 384
Lock and encrypt 406
logical operators 141
logical variables 247
Logistic() distribution function 274
LogisticRegression() function 310
Logit() function 242
Lognormal() distribution function 271
Logten() function 144
loops, dynamic 323
M
m to n sequence 173
magnification, printouts 23
Magnitude of a complex number 232
Make Alias command 421
Make Importance command 297, 421
Make Input Node command 421
Make Output Node command 421
Make same size 69
Make Same Size submenu 425
Manage published models
  menu item description 419
Math command 422
Math library 143
matrices
  definition 444
  dot product 219
  functions 218–221
  multiplication 219
Matrix command 422
matrix, identity 220
MatrixMultiply() function 219
Max() function 194
MDArrayToTable() function 202, 204, 397
MDTable() function 205
MdXQuery() function 399
mean value
  definition 444
  using 31
Mean Value command 424
Mean() function 291
median
  definition 444
  Latin hypercube sampling method 253
  see GetFract()
Median() function 291
memory
  increasing swapfile size 429
  Memory Usage window 428
  reducing requirements 370
  requirements 428
  understanding usage 411
  usage 426
Memory, configuring available 350
Memory, configuring virtual memory 350
Memory, insufficient 350
Memory, limits 350
MemoryInUseBy() function 412
menus
  choice 184
  command descriptions 419–427
  creating 123
  pull-down 19
  right mouse button 427
MetaIndex declaration 383
meta-indexes 382
MetaOnly attribute 343, 382
MetaVar..Do 383
Mid qualifier 357
Mid Value command 424
mid values
  definition 444
  using 29, 30
Mid() function 295
Min() function 194
Minimal Standard random number generator 253
MIrr() function 235
mixed correspondence 181
Mod() function 144
Model class 53
Model name, how to change 44
models
  behavior analysis 38
  browse-only 404, 440
  building 58
  closing 15
  combining 347
  creating 44
  definition 444
  documentation 61
  dynamic 68
  editing 45–52
  expanding 62
  hierarchy 340
  integrated 347
  large 340–350
  linking encrypted 406
  listing nodes 341
  locking and encrypting 404
  making easy to use 122
  modular 348
  navigating 341
  opening 14
  opening details 20
  protecting intellectual property 404
  reusing 60
  saving encrypted copies 405
  separating columns 395
  specifying attributes 44
  switching 15
  testing and debugging 60
  unexpected behavior 41
  viewing details 16
  working with 14
modes
  definition 444
  determining number of 247
  overview 18
  quantities 247
  switching 45
Module class 53
modules
  about 17
  adding 346
  connecting with arrows 49
Index

definition 445
displaying hierarchy 56
embedding 347
encrypted 406
filed, see filed modules
form 125
hierarchy 340, 445
linking to original 347
opening details 20
organizing hierarchy 71
subclasses list 53
Monte Carlo sampling method
  Intelligent Arrays 165
  using 252
mouse operations 185
Move Into Parent command 425
MsgBox() function 385
multiD tables, converting from relational 202
multiline text in table cells 179
multimodal distribution 445
Multinomial() distribution function 284
Multinormal() distribution function 283
MultiUniform() distribution function 284
multivariate distribution 445
multivariate distributions 281
Multivariate Distributions library
  using 282

N
  name-based calling syntax 143, 355
  name-based subscripting 159
  naming errors 432
  NAN 146
  natural cubic spline 208
  navigation
    shortcuts 105, 185
    toolbar 18
  NegativeBinomial() function 262
  new line
    entering within a table cell 179
  New Model command 419
  Node Style dialog. using 75
  nodes
    about 15
    adding icons 126
    adjusting size 68
    alias 50
    aligning 69
    arranging 68
    arranging front to back 70
    changing size 47
    consistent sizes 67
    creating 45
    creating aliases 50
    customizing 75
    cut, copy, and paste 47
    default size 74
    definition 445
    deleting 47
deselecting 18
distributing 70
duplicating 47
editing title 45
flagging with red triangle 56
font and typeface settings 74
grouping related 71
handles 46
identifying types 17
input, see input nodes
linking arrows to 48
list of attributes 343
moving 46
moving into the same diagram 50
output, see output nodes
redundant 348
selecting 17, 46
selecting multiple 18
shape descriptions 17
shape representations 17
text node type 128
title characteristics 67
undefined 56, 76
visual grouping 73
Z-order 70
Nonnegative qualifier 358
non-procedural programs 364
normal distribution 445
Normal_additive_gro() distribution function 285
Normal_compound_gro() distribution function 285
Normal_correl() distribution function 282
Normal_serial_correl() distribution function 285
Normal() distribution function 271
Normalize() function 201
Nper() function 235
Npv() function 235
Null system constant 146, 423
Number Format command 424
Number format dialog 78
number formats
  case sensitivity 79
  converting to text 145
  currency symbols 79
date numbers 228
expressions 137
import and export rules 338
integers 138
largest and smallest 138
list of types 79
number of decimal digits 79
OLE linking 331
options 138
precision 138, 428
quick reference 470
regional settings 80
settings 78
tables 28
thousands separator 79
trailing zeroes 79
Number qualifier 358
numbers in lists 170
NumberToText() function 227
numeric round-off error 138

O
Obfuscated, definition 445
obfuscated. See encrypted.
Object button 18
Object Finder dialog
definition 445
Distribution library 249
using 110
Object finder dialog 110
Object menu 421
Object window
definition 445
features of 44, 354
maximum number of 349
opening 20
showing values 22
using 20, 44
objective variables
arranging nodes 68
definition 445
representation 17
working with 58
objects
changing class 53
classes 17
definition 445
finding 342
handles 382
identifiers 46
searching for 112
titles 46
viewing definitions 55
ODBC 445
OLE linking
activating other applications 334
auto recompute links 56, 331
automatic vs. manual updates 330, 334
changing file locations 330, 334
definition 445
linking data from Analytica 328–331
linking data into Analytica 331–334
number formatting 331
OLE Links menu command 420
Open Source button 334
Paste Special dialog 333
procedure, from Analytica 329
procedure, to Analytica 331
refreshing links 331
table example 332
terminating links 334
using indexes 331
working with 327
one-dimensional array format 336
Open Model command 419
Open Source button 334
operators
arithmetic 140, 153
binding precedence 142
Boolean 141
comparison 140
definition 445
logical 141
scoping 141
text concatenation 226
Operators command 422
Optimizer command
Definition menu 422
Help menu 426
order of precedence 142
ordering qualifiers 359
originals vs. aliases 52
OrNull qualifier 358
outermost dimension 445
Outline button 18
Outline window 341, 445
out-of-range errors 369
output nodes
browsing 19
creating 124
definition 445
original variables 125
resizing 125
using 124
viewing values 20
writing to databases 398
outputs
arrowhead 446
attribute 343
definition 446
listing 16
remote 16
ovals 17
P
palette, color 73
parallelograms 17
parameters
assigning values 38
attribute 343
changing definition 38
defining as lists 38
definition 446
financial functions 233
parametric analysis 38
qualifiers 354, 356
repeated 360
using 356
varying inputs 38, 39
Parametric analysis 351
parametric sensitivity analysis 157
parent diagram
hiding nodes 404
returning to 20
settings 18
Index

viewing 16
Parent Diagram button 18
parenthesis matching 105
parents, definition 446
ParseDate() function 230
ParseNumber() function 228
Paste command 420
Paste identifier 110
Paste Identifier command 422
Paste Special command 420
PDF() function 295
percent number format 79
percentiles
definition 446
estimating 292
Performance profiler 351
Performance Profiler library
features 411
functions 412
using 411
Permutations() function 241
Pi system variable 423
pictures, see graphics
pivot table
creating array from relational tables 202
edit tables 178
result tables 28
see function MDTable() 205
Player edition 131
Pmt() function 236
Poisson() distribution function 261
PoissonRegression() function 311
positional correspondence 181
position-based calling syntax 143, 355
PositionInIndex() function 196
Positive qualifier 358
Ppmt() function 236
precedence, order of 142
precision, number
formats 138
specifications 428
Preferences command 420
Preferences dialog
changing window limits 350
disabling checking 120
features 54–56
opening 340
Print command 419
Print Preview command 419
Print Report command 24, 419
Print Setup command 419
printing options
background 24
fit to page 23
magnification/scale 23
multiple windows 24
page preview 23
printing to files 335
setting 23
Proactively evaluate indexes 56
Prob qualifier 357
Prob Table button 266
prob values
definition 446
Probvalue attribute 343
using 29
probabilistic variables 446
probability bands
definition 446
settings 254
uncertainty view 31
Probability Bands command 424
probability density
displaying graphs 424
Equal sample probability steps 256
Equal weighted probability steps 255
Equal X axis steps 255
function (PDF) 446
Histogram 255
options 255
samples per PDF step interval 255
Smoothing 256
Smoothing factor 256
uncertainty view 32
probability distributions
array parameters 281
beta 272
button 123
calculating 250
Chi-squared 277
choosing 246–248
computing 260
continuous 247
custom discrete 266
defining variables as 248
definition 446
discrete 247, 269
entering as expressions 249
functions 259–277
Gaussian 271
including in definitions 250
normal 271
triangular 270
truncating 279
uniform 270
probability functions, advanced 241
probability mass functions
definition 446
displaying in graphs 32
menu command 424
probability tables
adding conditional variables 267
conditional 267
creating 266
definition 446
expression view 267
Self index 267
splicing 180
working with 266
Probability() function 292
Index

Analytica User Guide 463
ProbabilityManagement.org 312
Probbands() function 292
Probdist() distribution function 278
Probit function -- see CumNormalInv 242
ProbitRegression() function 311
Probttable() distribution function 266
probtables, see probability tables
ProbWilcoxon() function 263
procedural programs
    constructs 365
    example 364
    using 364
Product() function 194
progressive refinement 166
public variables 348
Publish to cloud 133, 406
    menu item description 419
purchase price 159
pure associational correspondence 181
pure positional correspondence 181
Putoption() function 238
Pv() function 236
Pvgperp() function 240
Pvperp() function 240

Q
qualifiers
    All 358
    array 357
    Atom 357
    Coerce 359
    context 356
    ContextSample 357
    deprecated synonyms 361
    evaluation mode 356
    Handle 358
    Index 357
    Mid 357
    Nonnegative 358
    Number 358
    Optional 359
    ordering 359
    OrNull 358
    parameter 354, 356
    Positive 358
    Prob 357
    Reference 358
    Sample 357
    Scalar 357
    Text 358
    type checking 358
    Variable 357
quantiles 292
quantities
    bounds 247
    discrete vs. continuous 246
    discretizing process 247
    modes 247
    selecting distribution 248

skewed vs. symmetric 248

R
Radians() function 145
random Latin hypercube sampling 253
random number methods 253
random seed 254
Random() distribution function 280
Rank() function 199
Rankcorrel() function 294
Rate() function 236
ReadFromUrl() function 403
ReadTextFile() function 400
RealPart() function 233
Recent files 419
record indexes 392
records 392
recovery info settings 55
rectangles 17
recursion 369
Recursive attribute 343
recursive functions
    attributes 356
    settings 372
    using 372
reducing functions
    definition 446
    working with 192–197
reference operator 378
Reference qualifier 358
references 378
refinement, progressive 166
regional settings
    date formats 82
    number formats 80
regression analysis 307
Regression() function 307
RegressionDist() function 308
RegressionFitProb() function 309
RegressionNoise() function 309
Regular Expressions 228
relational tables, converting from multiD 202
remote variables
    definition 446
    seeing 16
Reorder
    See SortIndex() function
Reordering list items 179
resampling 324
Resize Centered command 425
Result button 18, 20, 124
result graphs, exporting 328
Result menu 424
result tables
    copying 328
    getting data 399
    retrieving 395
result views
    definition 447
Index

setting default 55
Result window
controls 26
default view 27, 55
graph view 28
index selection 27
maximum number of 54, 349
opening 26
table view 28
working with 26–35
results
analyzing 39
comparing 34
graph view 28
recomputing 28
table view 28
viewing 16
Return key, using to enter data 55
Right Edges command 425
Round 231
Round() function 144
round-off error 138
rows
adding and deleting 179
display significance 156
trading places with columns 156
Run system variable
description 177
menu command 423
probabilistic calculation 250
sample values 295
RunConsoleProcess() function 412

S
Sample command 424
Sample qualifier 357
sample size
selecting 416
setting 252
Sample() function 295
samples, definition 447
SampleSize system variable 252, 295, 423
Sampleweighting system variable 423
sampling methods
choosing 253
definition 447
median Latin hypercube 253
Monte Carlo 252
random Latin hypercube 253
selecting 252
Savage, Sam 312
Save A Copy In command 419
Save as browse-only model 406
Save As command 419
Save command 419
Save everything in one file by embedding linked modules 132, 406
scalar (0D) arrays 153
Scalar qualifier 357

scripts
assigning to global variables 408
creating 407
language 407
Sdeviation() function 291
Select All command 420
Selective parametric analysis 351
SelectText() function 224
self indexes 162, 173
definitions 447
semicolon, double 397
Semi-continuous domain 116
Send to Back 70
Send to Back command 427
Send to Front 70
sensitivity analysis
definition 447
functions 299–301
sequence operator 173
Sequence() function 173, 230
Set Diagram Style command 425
Set Node Style command 425
SetContains() function 210
SetDifference() function 210
SetIntersection() function 211
SetsAreEqual() function 211
SetUnion() function 211
shells, stand alone 348
shortcuts, navigation 105, 185
Show By Identifier command 421
Show Color Palette command 425
Show Invalid Variables command 422
Show Memory Usage command 426, 428
Show Page Breaks command 426
Show Result command 424
Show With Values command 421
ShowPdfFile() function 390
ShowProgressBar() function 387
Shuffle() distribution function 279
side effects 447
Sigmoid() function 242
Sign() function 144
Sin() function 145
SingularValueDecomp() function 222
Sinh() function 232
SipDecode() function 313
SipEncode() function 312
Size() function 214
skewed distributions
about 248
analytica user guide 465

comparison to symmetric 248
definition 447
formula 291
skewness 447
Skewness() function 291
slicers
definition 447
working with 85
slices
adding items to indexes 179
assigning to variables 368
construct 181
definition 447
effects of splicing 180
mixing with subscripts 182
preceding time period 182
types of correspondence 181
SLURP, Stochastic Library Unit with Relationships Preserved 312
Snap to Grid command 425
Sort
  Sorting arrays, see SortIndex() function
Sort() function 200
SortIndex() function 175
Space evenly 70
Space evenly submenu 425
Special command 422
Special library 422
splicing
  changing computed indexes 180
default correspondence 181
definition 447
  working with 180
SplitText() function 227
SpreadsheetInfo() function 402
SpreadsheetOpen() function 401
SpreadsheetRange() function 401
SpreadsheetSave() function 401
SpreadsheetSetCell() function 401
SpreadsheetSetInfo() function 403
SpreadsheetSetRange() function 402
SQL
  accessing databases 392
case sensitivity 395
definition 447
  retrieving result tables 395
  specifying queries 394
SqlDriverInfo() function 400
Sqr() function 144
Sqrt() function 144
standard deviation 447
Statistical command 422
Statistics
  functions 290
  setting options 254
  uncertainty view 31
  weighted 296
Statistics command 424
Statistics() function 295
Stepinterp() function 207
Stochastic Information Packets (SIPs) 312
Stochastic Library Unit with Relationships Preserved (SLURP) 312
strings, see text
StudentT() distribution function 275
subarrays
  extracting 181
  slicing and subscripting 181
Subindex() function 195
subscript construct
  mixing with slices 182
  using 181
Subscript() function 181
subscripting
  construct syntax 159
  name-based 159
  value v is array 159
Subset() function 174
  with arrays 214
SubTable() function 218
subtables, using 218
suffix notation
  characters 79
definition 447
  number formats 78
  using 79
Sum() function 162, 193
Svdindex system variable 423
symmetrical distributions
  comparison to skewed 248
definition 447
syntax
  checking in definitions 106
ers 432
  name-based 143, 355
  operators 142
  position-based 143, 355
system constants 146
system functions 447
system variables
definition 447
  Definition menu 112
  menu commands 422, 423
  menu options 423
  sample weighting 286
T
Table() function 191
tables
  converting between 202
  converting from graph view 28
copying 328
creating 190
definition 448
deterministic, see deterministic tables
displaying 28
edit, see edit tables
editable types 442
Index

features 28
import/export data format 336
lookup 207
modeling 149–186
multiple number formats 83
number formats 28
numerical data formats 338
probability, see probability tables
separating columns 395
tables, dependencies between cells 192
tails, definition 448
Tan() function 145
Tanh() function 232
templates
combining settings 93
creating 92
modifying 93
renaming 94
setting associations 94
unlinking 92
using with graphs 92
terminology 439–448
text
adding to diagrams 128
alphabetic ordering 141
combining with numbers 170
concatenation operators 226
converting to numbers 145
evaluating as scripts 410
functions 224
joining 226
reading and writing files 400
values 139
Text functions command 422
Text qualifier 358
TextLength() function 224
TextLowerCase() function 227
TextReplace() function 225
TextSentenceCase() function 227
TextUpperCase() function 227
three-dimensional array format 337
Tile Left to Right command 426
Tile Top to Bottom command 426
time
profiling 411
slices, preceding 182
Time system variable
description 177
details 318–321
erlier time reference 318
menu command 423
using in a model 321
working with 316
Time values, entering 139
Title attribute 343
titles
attribute characteristics 355
characteristics 67
definition 448

document text
editing 45
using 46
Today() function 230
toolbar
features 18
quick reference 470
Top Edges command 425
tornado charts 301
transformed beta distribution 273
transforming functions 197–201
Transpose() function 220
trapezoids
finding area 197
representation 17
Triangular() distribution function 270
True system variable 139, 423
Truncate() distribution function 279
truth values 139
Tutorial command 426
two-dimensional array format 337
type checking qualifiers 358
typographic conventions 10

U
uncertainties
dynamic simulations 324
expressing 246
mean 416
resampling 324
uncertainty factor 272
Uncertainty Options command 424
Uncertainty Sample option 251
Uncertainty Setup dialog 251–257
uncertainty views
cumulative probability 32
list of 30
mean value 31
mid value 30
probability bands 31
probability density 32
probability mass function 32
sample 33
statistics 31
working with 29
Uncumulate() function 198
Undo command 420
Unhide Definition(s) command 421
uniform distribution 448
Uniform() distribution(s) command 421
UniformSpherical() distribution function 284
Unique() function 176
unit identity matrix 220
unit vector 220
units
attribute 343
definition 448
using 355
univariate distribution 448
Update License command 426
Use Excel date origin preference 56
User guide command 426
user interfaces, creating for models 122
user libraries 361, 422
user-created attributes 343
user-defined functions
  attributes 355
  creating 355
  definition 448
  libraries 354, 361
  parameter qualifiers 356
  working with 353–362

V
Value attribute 343
values
  arrays 23
  assigning to parameters 38
  atoms 23
  Boolean 139
  checking bounds 56
  checking validity 117–120
  constants 162
  disabling checking 120
  expected 444
  listing 108
  showing in Object window 22
  text 139
  truth 139
  undefined 205
Variable qualifier 357
variables
  assigning slices 368
  automatic renaming 54
  chance 17, 441
  class checking 56
  classes 17
  comparing lists 161
  constants 17
  continuous 441
  declaring 366
  defining as distributions 248
  defining as edit tables 176–178
  definition 448
  description 15
  discrete 265, 442
  dynamic 316, 442
  examples 188
  finding 342
  general 17, 443
  index 17, 443
  invalid 345
  objective 17
  probabilistic 446
  public 348
  remote 16
setting number formats 78
  uncertain 324
variance
  definition 448
  estimating 291
  Variance() function 291
vector, unit 220
views
  default 27, 55
  uncertainty, see uncertainty views
W
w parameter 296
Wacc() function 240
warning icon 106
warnings, see errors
Web tech support command 426
Weibull() distribution function 276
weighted statistics 296
weighting, importance 285
Whatif() function 301
WhatifAll() function 301
While loops 369
While...Do function 371
Width command 425
Wiki 10
Wiki login info
  Help menu command 426
Wilcoxon() function 263
Window menu 426
windows
  see also Diagram window, Object window, Result window
  browsing 19
  changing number limits 350
  example images 469
  managing 349
  numbers displayed 54
  print settings 24
Windows system software 428
WorksheetCell() function 401
WriteTableSql() function 397
WriteTextFile() function 400
X
Xirr() function 236
XIMrr() function 237
Xnpv() function 237
XY button 97, 304
XY comparison
  dialog features 95
  examples 96, 99
  special menus 96
  using 95
XY plots 304
Y
YearFrac() function 238
Index

Z
Z-order, nodes 70
Zweidler, Daniel 312
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

Number Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suffix</td>
<td>letter denotes order of magnitude, such as M for 10^6 (see table below)</td>
<td>12.35K</td>
</tr>
<tr>
<td>Exponent</td>
<td>scientific exponential</td>
<td>1.235e+004</td>
</tr>
<tr>
<td>Fixed point</td>
<td>fixed decimal point</td>
<td>12345.68</td>
</tr>
<tr>
<td>Integer</td>
<td>fixed point with no decimals</td>
<td>12346</td>
</tr>
<tr>
<td>Percent</td>
<td>percentage</td>
<td>1234568%</td>
</tr>
<tr>
<td>Date</td>
<td>text date</td>
<td>12 Jan 2008</td>
</tr>
<tr>
<td>Boolean</td>
<td>true or false</td>
<td>True</td>
</tr>
</tbody>
</table>

Suffix format

<table>
<thead>
<tr>
<th>Power of 10</th>
<th>Suffix</th>
<th>Prefix</th>
<th>Power of 10</th>
<th>Suffix</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^3</td>
<td>K</td>
<td>Kilo</td>
<td>10^-2</td>
<td>%</td>
<td>percent</td>
</tr>
<tr>
<td>10^6</td>
<td>M</td>
<td>Mega or Million</td>
<td>10^-3</td>
<td>m</td>
<td>milli</td>
</tr>
<tr>
<td>10^9</td>
<td>G</td>
<td>Giga</td>
<td>10^-6</td>
<td>µ</td>
<td>micro (mu)</td>
</tr>
<tr>
<td>10^12</td>
<td>T</td>
<td>Tera or Trillion</td>
<td>10^-9</td>
<td>n</td>
<td>nano</td>
</tr>
<tr>
<td>10^15</td>
<td>Q</td>
<td>Quad</td>
<td>10^-12</td>
<td>p</td>
<td>pico</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10^-15</td>
<td>f</td>
<td>femto</td>
</tr>
</tbody>
</table>

For more, see “Number formats” on page 78.