Scripting Guide

“The real voyage of discovery consists not in seeking new landscapes, but in having new eyes.”

Marcel Proust
# Table of Contents

**JMP Scripting Guide**

## 1 Introducing JSL

### Tutorials and Demonstrations

- Hello, World .................................................. 1
- Modify the script ........................................... 3
- Save your script ............................................ 6
- Save your log ................................................. 7
- Saving and sharing your work ................................. 7
- Capturing scripts for data tables ............................. 7
- Capturing scripts for analyses ............................... 8
- A general method for creating scripts ....................... 9
- Use JMP interactively to learn scripting .................. 10

### Samples of JSL Scripts

- Data table programming .................................... 11
- Customized analyses ......................................... 13
- Custom displays ............................................... 18
- Matrix algebra with JSL .................................... 19
- Scripting graphs ............................................. 23
- Scripting instructional simulations ....................... 25

## 2 JSL Building Blocks

### Learn the Basic Components of JSL

- First JSL Script ............................................. 29
- The JSL Language ........................................... 31
  - Lexical rules of the language ........................... 31
  - Data elements ........................................... 32
  - Operators ................................................. 38
- Context: meaning is local ................................ 40
  - Programming vs. scripting ............................. 40
  - Data table context .................................... 41
  - Scoping operators ....................................... 42
  - Graph context ........................................... 42
- Name resolution ............................................. 42
  - Name-binding rules ................................... 43
3 JSL Operators
The basic functions in JSL

Numeric Functions ........................................... 61
  Arithmetic operators ........................................ 61
  Assignment operators ....................................... 61
  Constants ..................................................... 62
  Additional numeric functions ............................. 62
  Transcendental functions ................................ 63
  Trigonometric Functions .................................. 65
  Random Functions ......................................... 66

Character Functions ....................................... 67

Comparison and Logical Operators ......................... 74
  Comparison operators ...................................... 74
  Logical operators ......................................... 76
  Special cases .............................................. 76
  Missing values ............................................ 77
  Missing character values ................................ 78
  Short-circuiting behavior ................................ 78

Date/Time Operators ....................................... 79
  Constructing dates ........................................ 79
  Extracting parts of dates ................................. 79
  Arithmetic on dates ....................................... 80
  Converting date/time units ............................... 80
  Y2K-ready date handling ................................ 80
  Date/time notation ....................................... 81

Currency ..................................................... 85

Inquiry Functions ........................................... 85

Functions that communicate with users ..................... 86
  Writing to the log ......................................... 87
  Send information to the user ............................. 87
## Programming Functions

### Script Control
- Programming Example ........................................... 93
- Gluing expressions together ................................... 93
  - Glue ...................................................................... 93
  - First ...................................................................... 94
  - Example ............................................................... 94
- Iterating ................................................................... 94
  - For ......................................................................... 94
  - While ................................................................. 96
  - Summation ............................................................ 96
  - Product ............................................................... 96
  - Break and Continue ............................................. 96
- Conditional functions ............................................... 97
  - If .......................................................................... 97
  - Match .................................................................... 98
  - Choose .............................................................. 98
  - Interpolate .......................................................... 99
  - Step ...................................................................... 100
- Controlling script execution ..................................... 100
  - Wait ...................................................................... 101
  - Schedule ........................................................... 101
- Variables .................................................................. 101
  - Global Variables .................................................. 101
  - Local variables ..................................................... 102
- JSL Data Structures .................................................. 103
  - Lists ..................................................................... 103
  - Matrices ............................................................. 107
  - Associative Arrays ............................................... 107

## Data Tables

### Create, open, and manipulate data tables
- Data table basics ..................................................... 113
  - Getting a data table object .................................... 115
  - Objects and messages .......................................... 118
- Messages for data tables ........................................ 122
  - Naming, saving, and printing ................................. 122
  - Journal and Layout .............................................. 123
  - Creating and deleting columns .............................. 124
  - Accessing values ................................................ 125
  - Summarize .......................................................... 129
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Subtopics</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Scripting Platforms</td>
<td>Create, repeat, and modify analyses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scripting analysis platforms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Launching platforms interactively and obtaining the equivalent script</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sending script commands to a live analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Launching platforms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Objects within analysis objects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>General messages for platform windows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scripting specific platforms</td>
</tr>
<tr>
<td>7</td>
<td>Display Trees</td>
<td>Create and use dialogs and results windows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Manipulating displays</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Introduction to display boxes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DisplayBox object references</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sending messages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Using the &lt;&lt; operator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Platform example</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Syntax Reference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Constructing display trees</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Basics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updating an existing display</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Controlling Text Wrap</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interactive Display Elements</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Send messages to constructed displays</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Build your own displays from scratch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Construct display boxes containing platforms</td>
</tr>
</tbody>
</table>

- Working with metadata .................................................. 131
- Commands from the Rows menu ........................................ 136
- Commands from the Tables menu ...................................... 140
- Manipulating columns .................................................. 145
- Setting and getting attributes ...................................... 148
- Row State operators .................................................... 154
- What are row states? ..................................................... 155
- An overview of row state operators .................................. 159
- Each of the row state operators in detail ......................... 163
- Optional: the numbers behind row states ......................... 169
- Calculations ............................................................ 170
- Pre-evaluated statistics .............................................. 170
- Calculator formulas ................................................... 172
- 6 Scripting Platforms
- Create, repeat, and modify analyses .............................. 173
- Scripting analysis platforms ........................................ 175
- Launching platforms interactively and obtaining the equivalent script ........................................ 175
- Sending script commands to a live analysis ..................... 182
- Launching platforms .................................................... 186
- Objects within analysis objects .................................... 187
- General messages for platform windows ........................... 187
- Scripting specific platforms ........................................ 190
- 7 Display Trees
- Create and use dialogs and results windows ..................... 225
- Manipulating displays ................................................ 227
- Introduction to display boxes ...................................... 227
- DisplayBox object references ...................................... 230
- Sending messages ...................................................... 232
- Using the << operator .................................................. 233
- Platform example ...................................................... 234
- Syntax Reference ...................................................... 238
- Constructing display trees ......................................... 240
- Basics ................................................................. 240
- Updating an existing display ....................................... 242
- Controlling Text Wrap ............................................... 244
- Interactive Display Elements ...................................... 244
- Send messages to constructed displays ............................ 257
- Build your own displays from scratch ............................. 258
- Construct display boxes containing platforms ................... 259
Construct a custom platform ........................................ 260
Sheets ......................................................... 263
Journals ......................................................... 265
Picture Display Type ............................................ 265
Modal Dialogs ..................................................... 266
  Constructing modal dialog boxes ............................. 266
  General-purpose modal dialog boxes ....................... 268
  Data column dialog boxes ................................. 271
  Details common to Dialog and Column Dialog .......... 273
  Advanced example ......................................... 275
Scripting the Script Editor ................................... 276

8 Scripting Graphs
Create and edit 2-dimensional plots ......................... 277
Adding Scripts to Graphs ..................................... 279
Adding a Legend to a Graph ................................ 280
Creating New Graphs From Scratch ......................... 280
  Making changes to graphs .................................. 281
Graphing Elements ............................................ 282
  Plotting functions ......................................... 282
  Getting the properties of a graphics frame ............ 286
  Adding a Legend ............................................. 286
Drawing lines, arrows, points, and shapes ............... 287
  Lines ......................................................... 287
  Arrows ....................................................... 289
  Markers ....................................................... 290
  Pies and Arcs ............................................... 291
  Regular Shapes: Circles, Rectangles, and Ovals ....... 293
  Irregular Shapes: Polygons and Contours .............. 295
Adding text .................................................... 297
Colors .......................................................... 298
Transparency ................................................... 300
Fill patterns .................................................... 301
Line types ....................................................... 301
Drawing with pixels .......................................... 301
Interactive graphs ............................................. 302
  Handle ......................................................... 302
  MouseTrap ...................................................... 305
  Drag functions ................................................ 306
  Troubleshooting .............................................. 307
9 Three-Dimensional Scenes

Scripting in Three Dimensions .............................................. 309

About JSL 3-D Scenes ......................................................... 311
JSL 3-D Scene Basics ........................................................... 311
Scene Boxes .......................................................................... 311
Setting the Viewing Space ...................................................... 314
Setting Up a Perspective Scene .............................................. 315
Setting up an Orthographic Scene ........................................... 316
Changing the View ............................................................... 316
The Translate Command .......................................................... 316
The Rotate Command ............................................................. 316
The Look At Command ........................................................... 318
The ArcBall ............................................................................. 319

Graphics Primitives ............................................................... 321
Primitives Example ................................................................. 324
Controlling the Appearance of Primitives ............................... 326
Other uses of Begin and End ..................................................... 331
Drawing Spheres, Cylinders, and Disks ...................................... 331
Construction ............................................................................ 331
Lighting .................................................................................. 332
Drawing Text ............................................................................ 332
Using Text with Rotate and Translate ....................................... 333
Using the Matrix Stack ............................................................. 334

Lighting and Normals .............................................................. 337
Creating Light Sources ........................................................... 337
Lighting Models ...................................................................... 338
Normal Vectors ....................................................................... 339
Shading Model ........................................................................ 339
Material Properties ............................................................... 340
Alpha Blending ........................................................................ 340
Fog ......................................................................................... 341
Example ................................................................................ 341

Bézier Curves .......................................................................... 342
One-Dimensional Evaluators .................................................. 342
Two-Dimensional Evaluators .................................................. 344
Using the mouse ..................................................................... 345
Parameters .............................................................................. 347

10 Matrices

Matrix algebra in JMP ............................................................ 349

Basics ..................................................................................... 351
Contents

Definitions ........................................................................... 351
Constructing matrices ......................................................... 351
Numeric operators ................................................................. 353
Manipulating values ............................................................. 355
Subscripts ............................................................................ 356
Matrix operators ................................................................. 360
Solving linear systems ......................................................... 360
Further construction functions ........................................... 363
Decompositions and normalizations ................................... 365
Build your own matrix operators ........................................ 369
Matrices with the rest of JMP ................................................ 369
Matrices and data tables ...................................................... 369
Matrices and reports ........................................................... 370
Statistical examples ............................................................. 370
Regression example ............................................................ 370
ANOVA example ................................................................. 371

11 Production Environments

Scripting tools for production settings .................................. 375
Real-time data capture ......................................................... 377
Create a datafeed object ....................................................... 377
Manage a datafeed with messages ....................................... 378
Using sockets in JSL ............................................................ 381
Socket-related commands .................................................... 382
Messages for sockets .......................................................... 383
Database access (Windows and Linux only) ....................... 384
Scheduling actions .............................................................. 385
Working with SAS ............................................................... 387
Making a SAS DATA step ..................................................... 387
SAS Variable Names ............................................................ 387
Connecting to a SAS Metadata Server ............................... 387
OLE Automation ................................................................. 391
Automating JMP through Visual Basic ............................... 391
Automating JMP through Visual C++ ............................... 398

12 Advanced Concepts

Complex scripting techniques and additional functions .......... 401
Advanced Programming Concepts ....................................... 403
Throwing and catching exceptions ...................................... 403
Functions ........................................................................... 404
Recursion ........................................................................... 405
A  JSL Syntax Reference

Summary of operators, functions, and messages ................................................................. 437

Functions, Variables, and Other Basic Tools ................................................................. 439

Constants .......................................................................................................................... 442

Arithmetic and Assignment Operators ............................................................................ 442

Numeric Functions .......................................................................................................... 448

Transcendental Functions ............................................................................................... 449

Trigonometric Functions ................................................................................................. 452

Random Functions .......................................................................................................... 455

Character/String Functions ............................................................................................ 458

Comparison/Logical Functions ......................................................................................... 469

Date/Time Functions ........................................................................................................ 475

General Utility Functions ................................................................................................. 480

Loop Controls and Conditional Statements .................................................................... 494

Lists ................................................................................................................................. 496

Matrices ........................................................................................................................... 497

Associative Arrays ............................................................................................................ 510
Data Tables, Rows, and Columns ................................................................. 512
Platforms ......................................................... 528
Windows, Reports, and Dialogs ................................................................. 531
  Display Boxes ..................................................... 531
  Modal Dialogs .................................................... 548
  Graphs ............................................................... 549
  3d Scenes ............................................................ 559
Statistical and Probability Functions ....................................................... 559
  Probability Functions ............................................... 560
  Discrete Probability Functions ........................................... 567
  Specialty Probability Functions .......................................... 570
  Log Density Functions .................................................. 570
  Statistical Functions ................................................... 572
  Column Statistics ...................................................... 576
  Additional Statistical Functions ........................................ 579
Advanced Concepts ................................................................. 582
SAS Integration ............................................................... 598
Tools for Production Settings ............................................................. 630

B  Glossary
Terms, concepts, and placeholders ..................................................... 637

Index
Syntax Reference ................................................................. 641

Index
JMP Scripting Guide ................................................................. 651
Credits and Acknowledgments

Origin
JMP was developed by SAS Institute Inc., Cary, NC. JMP is not a part of the SAS System, though portions of JMP were adapted from routines in the SAS System, particularly for linear algebra and probability calculations. Version 1 of JMP went into production in October, 1989.

Credits
JMP was conceived and started by John Sall. Design and development were done by John Sall, Chung-Wei Ng, Michael Hecht, Richard Potter, Brian Corcoran, Annie Dudley Zangi, Bradley Jones, Craig Hales, Chris Gotwalt, Paul Nelson, Xan Gregg, Jianfeng Ding, Eric Hill, John Schroedl, Laura Lancaster, Scott McQuiggan, and Peng Liu.
In the SAS Institute Technical Support division, Wendy Murphrey and Toby Trott provide technical support and conduct test site administration. Statistical technical support is provided by Craig DeVault, Duane Hayes, Elizabeth Edwards, Kathleen Kiernan, and Tonya Mauldin.
Nicole Jones, Jim Borek, Kyoko Keener, Hui Di, Joseph Morgan, Wenjun Bao, Fang Chen, Susan Shao, Hugh Crews, Yusuke Ono and Kelci Miclaus provide ongoing quality assurance. Additional testing and technical support is done by Noriki Inoue, Kyoko Takenaka, and Masakazu Okada from SAS Japan.
Bob Hickey is the release engineer.
The JMP manuals were written by Ann Lehman, Lee Creighton, John Sall, Bradley Jones, Erin Vang, Melanie Drake, and Meredith Blackwelder, with contributions from Annie Dudley Zangi and Brian Corcoran. Creative services and production was done by SAS Publications. Melanie Drake implemented the help system.
Jon Weisz and Jeff Perkinson provided project management. Also thanks to Lou Valente, Ian Cox, Mark Bailey, and Malcolm Moore for technical advice.
Thanks also to Georges Guirguis, Warren Sarle, Gordon Johnston, Duane Hayes, Russell Wolfinger, Randall Tobias, Robert N. Rodriguez, Ying So, Warren Kuhfeld, George MacKensie, Bob Lucas, Warren Kuhfeld, Mike Leonard, and Padraic Neville for statistical R&D support. Thanks are also due to Doug Melzer, Bryan Wolfe, Vincent DelGobbo, Biff Beers, Russell Gonsalves, Mitchel Soltys, Dave Mackie, and Stephanie Smith, who helped us get started with SAS Foundation Services from JMP.

Acknowledgments
We owe special gratitude to the people that encouraged us to start JMP, to the alpha and beta testers of JMP, and to the reviewers of the documentation. In particular we thank Michael Benson, Howard Yetter (d), Andy Mauromoustakos, Al Best, Stan Young, Robert Muenchen, Lenore Herzenberg, Ramon Leon, Tom Lange, Homer Hegedus, Skip Weed, Michael Emptage, Pat Spagan, Paul Wenz, Mike Bowen, Lori Gates, Georgia Morgan, David Tanaka, Zoe Jewell, Sky Alibhai, David Coleman, Linda
Blazek, Michael Friendly, Joe Hockman, Frank Shen, J.H. Goodman, David Iklé, Barry Hembree, Dan Obermiller, Jeff Sweeney, Lynn Vanatta, and Kris Ghosh.

Also, we thank Dick DeVeaux, Gray McQuarrie, Robert Stine, George Fraction, Avigdor Cahaner, José Ramirez, Gudmundur Axelson, Al Fulmer, Cary Tuckfield, Ron Thisted, Nancy McDermott, Veronica Czitrom, Tom Johnson, Cy Wegman, Paul Dwyer, DaRon Huffaker, Kevin Norwood, Mike Thompson, Jack Reese, Francois Mainville, and John Wass.

We also thank the following individuals for expert advice in their statistical specialties: R. Hocking and P. Spector for advice on effective hypotheses; Robert Mee for screening design generators; Greg Piepel, Peter Goos, J. Stuart Hunter, Dennis Lin, Doug Montgomery, and Chris Nachtsheim for advice on design of experiments; Jason Hsu for advice on multiple comparisons methods (not all of which we were able to incorporate in JMP); Ralph O’Brien for advice on homogeneity of variance tests; Ralph O’Brien and S. Paul Wright for advice on statistical power; Keith Muller for advice in multivariate methods, Harry Martz, Wayne Nelson, Ramon Leon, Dave Trindade, Paul Tobias, and William Q. Meeker for advice on reliability plots; Lijian Yang and J.S. Marton for bivariate smoothing design; George Milliken and Yuriii Bulavski for development of mixed models; Will Potts and Cathy Maahs-Fladung for data mining; Clay Thompson for advice on contour plotting algorithms; and Tom Little, Damon Stoddard, Blanton Godfrey, Tim Clapp, and Joe Ficalora for advice in the area of Six Sigma; and Josef Schmee and Alan Bowman for advice on simulation and tolerance design.

For sample data, thanks to Patrice Strahle for Pareto examples, the Texas air control board for the pollution data, and David Coleman for the pollen (eureka) data.

Translations

Erin Vang coordinated localization. Noriki Inoue, Kyoko Takenaka, and Masakazu Okada of SAS Japan were indispensable throughout the project. Special thanks to Professor Toshiro Haga (retired, Science University of Tokyo) and Professor Hirohiko Asano (Tokyo Metropolitan University for reviewing our Japanese translation. Special thanks to Dr. Fengshan Bai, Dr. Xuan Lu, and Dr. Jianguo Li, professors at Tsinghua University in Beijing, and their assistants Rui Guo, Shan Jiang, Zhicheng Wan, and Qiang Zhao, for reviewing the Simplified Chinese translation. Finally, thanks to all the members of our outstanding translation teams.

Past Support

Many people were important in the evolution of JMP. Special thanks to David DeLong, Mary Cole, Kristin Nauta, Aaron Walker, Ike Walker, Eric Gjertsen, Dave Tilley, Ruth Lee, Annette Sanders, Tim Christensen, Jeff Polzin, Eric Wasserman, Charles Soper, Wenjie Bao, and Junji Kishimoto. Thanks to SAS Institute quality assurance by Jeanne Martin, Fouad Younan, and Frank Lassiter. Additional testing for Versions 3 and 4 was done by Li Yang, Brenda Sun, Katrina Hauser, and Andrea Ritter.

Also thanks to Jenny Kendall, John Hansen, Eddie Routten, David Schlotzhauer, and James Mulherin. Thanks to Steve Shack, Greg Weier, and Maura Stokes for testing JMP Version 1.

Thanks for support from Charles Shipp, Harold Gugel (d), Jim Winters, Matthew Lay, Tim Rey, Rubin Gabriel, Brian Ruff, William Lisowski, David Morganstein, Tom Esposito, Susan West, Chris Fehily, Dan Chilko, Jim Shook, Ken Bodner, Rick Blahunka, Dana C. Aultman, and William Fehlner.

Technology License Notices

The ImageMan DLL is used with permission of Data Techniques, Inc.
Scintilla is Copyright 1998-2003 by Neil Hodgson <neilh@scintilla.org>. NEIL HODGSON DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL NEIL HODGSON BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

XRender is Copyright © 2002 Keith Packard. KEITH PACKARD DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL KEITH PACKARD BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
This introduction shows you the basics of JMP Scripting Language (JSL). The chapter starts with a simple, progressive tutorial example to show you where to type a script, how to submit it, and how to modify and save it. The purpose of this tutorial is to give you the basic techniques for working with any script, whether it’s one you write or one you get from someone else. Next is a showcase of examples to demonstrate how scripting might be useful in a variety of settings—for classroom simulations, advanced data manipulations, custom statistics, production lines, etc.

Confusion alert! Throughout this book, special shaded “confusion alerts” like this one call your attention to important concepts that could be unfamiliar or more complicated than you might expect—or where JMP might be a little different from other applications. These alerts appear whenever a particularly good example of a potential problem arises in the text, and although you will find them under topics that might not apply to your immediate needs, the ideas presented are always general and important. Please be sure to take a look even when you’re skipping pages and looking for something else.

You can quickly locate these pointers by looking up “confusion alert” in the “Index,” p. 651.
Contents

Hello, World ................................................................. 3
Modify the script ......................................................... 5
Save your script .......................................................... 6
Save your log ............................................................... 7
Saving and sharing your work ......................................... 7
Capturing scripts for data tables .................................... 7
Capturing scripts for analyses ....................................... 8
A general method for creating scripts ............................ 9
Use JMP interactively to learn scripting ............................ 10
Samples of JSL Scripts .................................................. 11
Data table programming ............................................... 11
Customized analyses .................................................... 13
Custom displays ......................................................... 18
Matrix algebra with JSL ................................................. 19
Scripting graphs .......................................................... 23
Scripting instructional simulations .................................. 25
Hello, World

This exercise is simple and hearkens back to a classic. Perhaps you'll recognize it.

1 Start JMP.
2 If the log window isn't open, open it by selecting View > Log (Windows and Linux) or Window > Log (Macintosh). Note that the Macintosh command won't be enabled until something is written to the log.
3 In the JMP Starter window, click New Script.
   New scripts can also be created from the menus. On Windows and Linux, select File > New > Script. On the Macintosh, select File > New > New Script.

Figure 1.1 Starting a New Script

4 In the resulting script editor, type these lines:
   A="Hello, World";
   Show(A);
5 From the Edit menu, select Run Script.
   Shortcut: press Control-R on Windows and Linux or ⌘-R on Macintosh.
Figure 1.2 Running a Script

The result is shown in the Log window. Besides showing results and errors, this window is also a script editor.

Figure 1.3 The JMP Log

This is how you enter and submit JSL. You have just created a global variable, A, assigned a value to it, and shown its value. Notice that the log echoes your script first, and then echoes any results—in this case, the output for the `Show(A)` command.

Go To Line

As your scripts grow in size, it is handy to jump to specific lines in the script. Use the Go To Line command from the Edit menu to jump to a specific line.

This command is also useful during the debugging process, since error messages frequently mention the line of the script where the error occurred.
Modify the script

Now try making this script do a little more work by adding a `For`-loop to greet the world not just once but four times. Also, `Print` this time instead of `Show`, and make a few other changes to see how the `For`-loop works.

1. In the script window, change the script to this:
   ```javascript
   for(i=1, i<5, i++,
     X=i;
     A="Hello, World";
     print(X,A);
     print("done");
   )
   print("done");
   ```

2. Submit the script as before.

(To submit part of the text in a window instead of all text, select the text first and then press Control-R on Windows or ⌘-R on Macintosh)

```
1 "Hello, World"
2 "Hello, World"
3 "Hello, World"
4 "Hello, World"
"done"
```

`Print` is similar to `Show` except that it doesn’t label the results with the variable name. It just prints the value.

Stopping the script

When you run a script, the `Run Script` command in the `Edit` menu changes to `Stop Script`. While a script is executing, you can select this item (or type Esc on Windows or ⌘-Period on Macintosh) to stop it. Many scripts execute more quickly than you can stop them.

Punctuation and spaces

Notice the indented lines inside the `For`-loop. This isn’t necessary for JMP’s sake; it just makes it a little easier to read. You could write this script like this if you want:

```javascript
for R(i=
  1,i<5,i++
  ,X=i;A="Hello, World";print(X,A));print("done");
```

Words in JMP’s scripting language are separated by parentheses, commas, semicolons, and various operators such as +, –, and so on. As far as JMP is concerned, spaces, tabs, returns, and blank lines inside or between operators or within JSL words don’t exist. This is because most JSL words come from JMP’s usual graphical user interface (GUI), and most of those commands have spaces in them. For instance, you will see later on that to do `Fit Model`, the JSL would be `Fit Model(and some arguments inside parentheses)`. Most people would rather see “`Fit Model`” than “`fitmodel`.”

JSL isn’t case-sensitive, so you don’t have to worry about reaching for the Shift key all the time.
You do have to be a little careful inside a text string. In “Hello, World” extra spaces would affect the output, because text strings inside double quotation marks are taken literally, exactly as you type them. Also, you would get errors if you put spaces between the two plus signs in i++, (i+ +), or in numbers, (4 3 is not the same as 43).

Generally, JSL uses:

- **commas (,)** between arguments all the parentheses for a command.
- **parentheses ()** to surround all the arguments of a command. Many JSL words have parentheses after them even if they don’t take arguments; for example π() has the parentheses even though π is just the number 3.14… Pi doesn’t expect an argument and will complain about any argument you do give it. Therefore, Pi(mincemeat) would be considered an error (although it seems heretical to say so). But the parentheses are still required.
- **semicolons (;)** to separate commands but also to glue them together. In other words, you use a semicolon to separate one complete message from the next, e.g. a=1;b=2, but what the semicolon really does is tell JMP to continue and do more things. For example, the For-loop example showed how to put several statements in the place of one statement for the fourth argument, because the semicolon effectively turned all three statements into one argument.

Trailing semicolons (extras at the end of a script or at the end of a list of arguments) are harmless, so you can also think of semicolons as terminating characters. In fact, terminating each complete JSL statement with a semicolon is a good habit to adopt.

- **quotation marks (" ")** to enclose text strings. Anything inside quotation marks is taken literally, exactly as is, including spaces and upper- or lower-case. Nothing that could be evaluated is evaluated; if you have π()^2 inside quotation marks, it is just a sequence of six characters, not a value close to ten. See “Quoted Strings,” p. 34, for ways to include special characters and quotation marks in strings.

For a more formal discussion of punctuation rules, see “Lexical rules of the language,” p. 32.

**Save your script**

If you would like to save your script, just do this:

1. Make the script window active (click the “Untitled” window to make it the front-most window).
2. From the File menu, select Save or Save As.
3. Specify a filename, including the extension .jsl (e.g., hello.jsl).
4. Click Save.

Scripts are saved as text files, and you can edit them with any text editor. However, if you do edit scripts with applications other than JMP, be careful to save them as plain text files. If you preserve the .jsl extension, you can double-click a script file to launch JMP.

To reuse a script, use Open from JMP’s File menu, double-click a .jsl file, or drag and drop the file onto JMP’s application icon.

When opening a JSL file, the actual script is always opened in its own script window. However, it may be distracting to some users to see this window. To keep a script from opening in a script window, put

`//!`
on the first line of the script.
Hold down the Control key (Option key on Mac) when choosing the Open command to bypass run-on-open.

**Save your log**

You can also save logs as text files, which can be viewed with any text editor. Double-clicking a log file does not launch JMP.

1. Make the log window active (click the Log window to make it the front-most window).
2. From the File menu, select Save or Save As.
3. Specify a filename, including the extension .txt on Windows (e.g., hello.txt).
4. Click Save

---

**Saving and sharing your work**

Here's something just about everybody will find useful sooner or later: JMP can create scripts to duplicate your data tables and analyses. For example:

1. Suppose you need to describe an analysis process in detail, from beginning to end, such as to create an audit trail for a governing agency or for peers reviewing your journal article.
2. Suppose you have a set of analysis steps that should be followed routinely by your lab technicians.
3. Suppose you fit the same model to new data every day, and you're tired of clicking the same buttons over and over again.
4. Suppose you're working with somebody in another city who can't simply look over your shoulder to see how you've put something together.

You can use JMP interactively as usual, then save scripts, and in the future just run those scripts. Next are some examples showing how this works.

**Capturing scripts for data tables**

1. Open a data table and make all kinds of changes—add rows and columns, change values, rearrange columns, sort the rows, make a formula column, make a row state column, and so on.
2. When you're done, open a script window and type this in:
   ```javascript
   current data table()<<get script;
   ```
3. In the Script window, click and drag to select (highlight) the script.
   Then, run the script and take a look at the output shown in the Log window:
4. From the Edit menu, select Run Script.
Saving and sharing your work

Figure 1.4 The Get Script Command and the Log

Now try running the script in the log window:
1 In the Log window, click and drag to select (highlight) the script, starting with `New Table`.
2 From the **Edit** menu, select **Run Script**.

The script produces a perfect clone of your data table.

Capturing scripts for analyses

Launch a platform—any platform, such as Fit Model. Take a look at the default results and then go exploring. Try options in the pop-up menus to see related tests and graphs. Work with the red pop-up menus in the report surface to get exactly the report you want.

When you're done, get a script to recreate your results. There are two methods of getting a script for your results. One is to use the Script submenu located at the bottom of each platform's popup menu. This method is detailed in “Use JMP interactively to learn scripting,” p. 10.

1 First you need to figure out what JMP calls your analysis. This can be tricky in some cases, discussed in the “Scripting Platforms” chapter, but usually you can read it right off the title of the analysis window, after the name of the data table. For example, a window titled “Big Class: Fit Least Squares” would be called `Fit Least Squares`.
2 Now you have to specify “which one.” You might have fit several models before getting the one you want to keep. You need to tell JMP which one you want by supplying a subscript, which is just a number inside brackets after the name. If the third model you fit is the one you want, you would specify it as `Fit Least Squares[3]`.
3 You're ready to get the script from the object:
Fit Least Squares[3]<<get script;
The results might look something like this, depending on which steps you performed:

\[
\text{Fit Model}(\text{Y}(:weight), \text{Effects}(\text{:sex}, \text{:height}, \text{:sex } \times \text{:height}), \text{Personality}(\text{Standard Least Squares}), \text{Run Model}(\text{Profiler}(\text{Confidence Intervals}(1), \text{Desirability Functions}(1)), \text{Contour Profiler}(\text{Surface Plot}(1))))
\]

Try running the script:
1. In the Log window, click and drag to select (highlight) the script
2. From the Edit menu, select Run Script.

The script produces a perfect clone of your analysis. If you want a journal, that's easy to do:

Fit Least Squares[3]<<journal window;

A general method for creating scripts

1. Did you use an existing data table as is? Write an Open statement for it:
   \[dt=\text{open}("\$\text{SAMPLE\_DATA/Big Class.JMP}"");\]
2. Did you create a new table or make changes to an existing data table? Did you work with row states, such as to color and label points in your plots? Did you exclude some rows? Did you fix errors? If so, you should get a script to recreate your data table.
   \[\text{current data table}()\ll<\text{get script};\]
3. Which platforms did you launch, work with, and keep for your final results? Get scripts to recreate them. You'll learn the details in the "Scripting Platforms" chapter; here you'll just try some examples.
   \[\text{Bivariate}[1]\ll<\text{get script};\]
   \[\text{Fit Least Squares}[1]\ll<\text{get script};\]
   \[\text{spinning plot}[1]\ll<\text{get script};\]
4. Now edit the log into a complete script. Be sure to put semicolons (;) in between statements. You might have something like this:
   \[
   \text{New Table("Big Class", Add Rows(40), New Column("name", Character, Nominal, Set Property(Notes, "...usually used as a label variable in plots"))}, \text{New Column("age", Numeric, Ordinal, Set Property(Notes, "Explore data adventurously"))}, \text{New Column("sex", Character, Nominal, Set Property(Notes, "Explore data adventurously"))}, \text{New Column("height", Numeric, Continuous, Set Property(Notes, "Explore data adventurously"))})
   \]
Introducing JSL

Chapter 1

Saving and sharing your work

Bivariate(Y(weight), X(height), Fit Line({Confid Curves Fit(1)}), Where(:sex == "F")); Bivariate(Y(weight), X(height), Fit Line({Confid Curves Fit(1)}), Where(:sex == "M"));
Fit Model(Y(:weight), Effects(:sex, :height, :sex * :height), Personality(Standard Least Squares), Run Model(Profiler(Confidence Intervals(1), Desirability Functions(1)), :weight << {Plot Actual by Predicted(1), Plot Residual by Predicted(1), Plot Effect Leverage(1)}));
Scatterplot 3D(Y(age, weight, height));

5 Save the script.
6 If you share your JSL file with your colleagues, don’t forget to include any data tables or additional files along with the script.

Use JMP interactively to learn scripting

One of the simplest ways to accomplish anything is to get somebody else to do it for you, and writing JSL is no exception. The best JSL-writer you’ll ever find is JMP itself. This example shows how to work in JMP interactively and then save the results as a script to reuse later on. With simple modifications, this script can serve as a template for speeding up routine tasks.

What JMP can and cannot do to help you write scripts

JMP can automatically save scripts to reproduce any data table or analysis in its current state. You can pause any time in your analysis to save a script to a script window, in a data table, or in an analysis report. You can then modify the automatically-generated script as needed for future projects.

JMP cannot record scripts while you are working. While script-recording is a useful feature in some other scripting languages, it is less important for a program like JMP, where the important thing is the results. You cannot use script-recording to observe how a sequence of interactive steps are performed. However, remember that you can save a script when you’re finished, and that script will reproduce everything you have accomplished.

JMP’s scripting language is not intended to be an alternative command-line interface for using the program. JSL is intended for recreating results and for extending JMP’s capabilities beyond JMP’s intended use in data discovery.

There’s more than one way to do it

Since JSL is a very flexible language, you can accomplish things many different ways.

Typically the script that JMP saves for you specifies every detail of your analysis, even if most of the details happen automatically by default. Does that mean that the scripts you write have to be just as complete and detailed? Not at all. You usually just need to specify the details you would specify when using the graphical user interface (GUI). For example, if you open Big Class and want to launch Distribution for height, weight, and sex, you would only need to do this in JSL:
Chapter 1

Introducing JSL

Samples of JSL Scripts

What can you do with JSL? The rest of this chapter has examples of scripts and their output.

Data table programming

JMP has many functions and operators for creating formula columns to do special things. JMP also has a number of Tables operations such as Sort, Subset, Summary, Transpose, etc. If you need to do something more elaborate, you might be able to write a script to do the work for you.

This section shows some examples for scripting advanced data table operations.

Use the right tool for the job

First, a disclaimer. JMP is not intended to be a powerhouse data management tool. JMP has an intentionally limited set of data capabilities, because its real power is in data discovery. If you need to manage large amounts of data or do complex data manipulations, consider using the SAS System® instead.

M-row products

This example multiplies values in a column \( m \) rows at a time and stores the products in a column of a new data table.

\[
m = 10; \quad // \text{compact } m \text{ rows to 1}
\]

\[
// \text{open the table Big Class.jmp}
sourceDt = open("$SAMPLE_DATA/Big Class.JMP");
\]

\[
// \text{create a new table with rows and columns to hold results}
destDt = new table("Products");
destDt <* new column("result");
destDt <* add rows(nrow(sourceDt)/m);
\]

\[
\text{for}(i = 0, i < \text{nrow(sourceDt)}/m, i++, \text{ result}=1; \text{ for}(j = 1 \ast m, j < i \ast m+m, j++, \text{ result} *= (\text{data table("Big Class"):weight})[j+1]); (\text{data table("Products"):result})[i+1]=\text{result};
\]
Introducing JSL

Chapter 1

Samples of JSL Scripts

You could make a similar script for sums instead of products by replacing "Products" with "Sums", *= with +*, and changing result=1 to result=0. Another way would be to make a formula column to define m-row groups, and then use JMP's built-in Summary capability:

```julia
dt<<new column("Group", formula(Floor((Row() - 1)/10) + 1));
dt<<summary(Group(Group), sum(weight), output table name("newDt"));
```

Compare two data tables

Here is a script to open two data tables, compare them, and report all results to the log window. This script makes heavy use of the For operator to loop operations over all the rows (loops with i), nested with another For operator to loop operations over all the columns (loops with j).

```julia
/* COMPARE THE CONTENTS OF TWO DATA TABLES */
// First identify the two data tables to be compared.
// Since you don't specify arguments, Open() presents the
// host-system dialog box for opening a file.
caption("Please locate the first data table.",spoken(1));
dt1=open();
caption("Please locate the second data table.");
dt2=open();

// Check for differing sizes.
if(nrow(dt1)!=nrow(dt2),
   write("Differing number of rows. "),
   write("Same number of rows. "));
if(ncol(dt1)!=ncol(dt2),write("Differing number of columns. "),
   write("Same number of columns. "));
nr = min(nrow(dt1),nrow(dt2));
nc = min(ncol(dt1),ncol(dt2));

// Check for differing column names.
for(j=1,j<=nc,j++,
   current data table(dt1);n1=char(ColumnName(j));
current data table(dt2);n2=char(ColumnName(j));
if(n1!=n2,
   write("\!rDiffering column names in column "||char(j));
show(n1,n2));

// check for differing values
for(i=1,i<=nr,i++,
   for(j=1,j<=nc,j++,
      v1 = Column(dt1,j)[i];
v2 = Column(dt2,j)[i];
if(v1!=v2,
   write("\!rDiffering values at row "||char(i)||
   " in column "||char(j));show(v1,v2);e++));
if(e==0,
   write("Matching cells have the same values."));```
write("Number of cells that exist in both tables ");
write("and have differing values: ");

// check for differing row states
for(i=1,i<=nr,i++)
  current data table(dt1); r1 = char(rowstate(i));
  current data table(dt2); r2 = char(rowstate(i));
  if(r1!=r2,
    write("Differing row states at row ");
    show(r1,r2));

Observe how Current Data Table is used to switch between two open windows.
Notice that the last step used Char() on the row state values, because row states are returned as expressions. The Char operator quotes its argument so it is treated as a string, which makes it possible to compare and display the row states.

There are other ways that data tables could be different. You might also want to take metadata such as table and column properties into account, for example. To improve the script still further, you could study the “Display Trees” chapter to learn how to present the results in a custom report window similar to those produced by JMP’s analysis platforms.

**Customized analyses**

Scripting presents new opportunities to customize JMP’s built-in analyses to meet your exact needs. This section presents two examples with brief explanatory comments.

To learn more about building your own custom analyses, first learn about analysis platforms in general in the chapter “Scripting Platforms,” p. 173, and then turn to “Constructing display trees,” p. 240.

**Add new features**

Suppose you wanted to have the Bivariate platform fit several orthogonal regressions of different variance ratios interactively through a slider, instead of one at a time through the menus.

With JSL, you can customize Bivariate to do exactly that:

```jsl
myVarRatio = 1;
myBiv= new window("Interactive Bivariate",
  v list box (  
    biv=Bivariate(Y(weight), X(height),DensityEllipse(.9),
      Fit Orthogonal(10^myVarRatio)),
    outline box("Adjustable variance ratio",
      vlistbox(  
        text box("Drag to change variance ratio, 10E-5 to 10E+5"),
        slider box (-5,5, myVarRatio,
          biv << (curve[2]<<remove fit);
          biv << fit Orthogonal (10^myVarRatio)<<reshow)))));
```
Introducing JSL
Chapter 1
Samples of JSL Scripts

Figure 1.5  Example: Interactive Bivariate

Here's a brief explanation of how this script works. First launch Bivariate with the height and weight variables and request an initial spline fitting with tension 1000:

```julia
biv=Bivariate(Y(weight), X(height), Fit Orthogonal(10^myVarRatio));
```

Then make a slider box that goes from –5 to 5 and sets a value for the global variable myVarRatio. The slider box then has a script that sends two messages to the Bivariate platform, the first to remove the current fit, and the second to add a new fit with the chosen variance ratio.

```julia
slider box (-5,5, myVarRatio, 
  biv << (curve[2]<<remove fit);
  biv << fit Orthogonal (10^myVarRatio)<<reshow)
```

Use a text box to label the slider with instructions to the user:

```julia
text box("Drag to change variance ratio, 10E-5 to 10E+5")
```

Stick both together in a vertical list inside an outline node:

```julia
outline box ("Adjustable variance ratio",
  vlistbox(
    text box(...)),
  biv << (curve[2]<<remove fit);
  biv << fit Orthogonal (10^myVarRatio)<<reshow)
```
Then glue the outline node to the platform’s results inside another vertical list, and present the whole thing in a window:

```
myBiv= new window ("Interactive Bivariate",
    v list box ( 
        biv=Bivariate(...),
        outline box (...)
    ) 
);
```

Finally, assign the whole thing to a global variable `myBiv` so that you can send later commands to it, such as:

```
myBiv<<journal window;
```

**Simplify repetitive or tedious work**

Suppose you want some way to make it easier to set parameters for large numbers of analyses. Perhaps your work involves looking at capability analyses for many columns at a time. You might find it tedious to specify upper and lower specification limits and target values for each and every one by hand. Further, you might only need to pay close attention to those results whose $C_p$ result is less than one, so you might want a way to skip over all the others.

Many situations such as this can be handled simply with a script that includes the details and uses a `For`-loop to handle the repetitions, like this:

```
collist={}; //Build a list of 36 column names
For(i=1, i<=36, i++, InsertInto(collist,column name(i)));
Distribution(Y(eval(collist)), Capability Analysis(LSL(15), USL(19), Target(17)));
```

However, when each column needs different limits, writing a script for several dozen launches with different limits would be almost as tedious as launching several dozen analyses and setting the parameters interactively. A better solution is to write a script that reads specifications from a second data table. That way, you can quickly and easily type in a big table of specifications and reuse it with future data.

1. Start by opening a data table with current readings (such as these artificial brewery data).
```
data=open("$SAMPLE_DATA/Readings.JMP");
```
2 Create another table that has the parameters needed, in this case LSL, USL, and Target:

```julia
specs = New Table("SpecLimits.jmp", Add Rows(5), New Column("ColName", Character, Nominal, width(30), Values("pH", "Specific gravity", "Temperature, degrees C", "Sediment, ppb", "Alcohol")), New Column("LSL", Numeric, Nominal, Values([5.1, 1.018, 2, 85, 4.2])), New Column("USL", Numeric, Nominal, Values([5.35, 1.003, 6, 115, 4.3])), New Column("Target", Numeric, Nominal, Values([5.25, 1.002, 4, 102, 4.25])));
```

Figure 1.7 Spec Limits Data Table

3 Build all the analyses without actually showing them. First create a vertical list box that starts out with just a simple text box inside it, called `vl`.

```julia
vl = vListBox(textbox("Spec Limits Script"));
```

4 Then use For Each Row to iterate on the SpecLimits table, using Substitute to plug values from the table into a generic Distribution launch-command expression.

Having built the command for one row of the SpecLimits table, you can switch to the Readings table and run the command inside a vertical list box called `vv`, which you don't actually see because no window was created for it. However, you can look inside `vv` to check whether the current $C_p$ result is greater than 1, and if so close its main outline node.

5 Next Append the result (closed or not) to that vertical list box created at the beginning, called `vl`. 
6 Then switch back to the SpecLimits table and loop through the remaining rows.

```julia
CurrentDataTable(specs);
for each row(
    // build commands using values from the SpecLimits table
    command = substitute(
        expr(Distribution(Y(_C), Quantiles(0), Moments(0),
           Capability Analysis(LSL(_LSL), USL(_USL), Target(_Target)))),
        expr(_C), ColName,
        expr(_LSL), lsl,
        expr(_USL), usl,
        expr(_Target), target);
    show(command);

    // do the analysis, store results in display tree vv
    currentDataTable(data);
    vv=vListBox(command);

    // test the CP by looking in display tree
    cp = vv["Capability Analysis"][tableBox(2)][NumberColBox(1)]<<get(1);
    if (cp>1,vv[OutlineBox(1)]<<close);

    // collect reports
    vl<<append(vv);
    currentDataTable(specs)
);
```

7 Finally, create a New Window to show all the displays in vl.

```julia
// display reports together in one window
newFit = NewWindow("Spec Limits Collection",vl);
```

The results look like this:
Custom displays

You can also use scripting to create new displays from results that JMP already knows how to make. For example, quality engineers frequently study six common capability analysis displays for a process variable: an $\bar{X}$-chart, a moving range chart, a histogram, a box plot, quantiles, and basic descriptive statistics. JMP provides each of these displays, but by default they appear in two different platforms: the two classic QC displays are in Control Chart, and the others are performed by Distribution.

To view these results together in one window, you simply need to build a New Window that contains one parent Outline Box node with two results, one from Control Chart and the other from Distribution. To do this, place the usual commands for launching the platforms inside VListBox, which is simply a vertical container, and then glue the two containers together side-by-side with HListBox, which is a horizontal container. The HListBox goes inside one OutlineBox, and that comprises the contents of the New Window.

```javascript
    csp=NewWindow("Capability Sixpack",
                  OutlineBox("Capability Sixpack",
                                HListBox( 
                                    VListBox(cc=Control Chart(chart Col(Height, Individual Measurement,Moving Range),K Sigma(3))),
                                )))
```

Figure 1.8 Spec Limits Collection
The process for working with JMP’s display tree containers is simple:

1. First, create each of the results you need by launching platforms through the usual menus.
2. Then, use Save Script from each platform to get the corresponding JSL.
3. Finally, glue the JSL pieces together inside containers in a New Window.

If you all you want to do is arrange results from several analyses into a single report, you can use JMP’s Layout window, discussed in the “Reports” chapter of JMP User Guide. Scripting adds the ability to combine live results with which you can interact and perform further tests, and the ability to make the same arrangements over and over again without all the clicking and dragging.

**Matrix algebra with JSL**

JSL adds an important new functionality to JMP: the ability to work directly with matrices. JMP can do all the usual elementwise numeric operations and also a broad range of matrix operations, such as matrix multiplication and division, determinants, eigenvalue decomposition, and so on. Thus you can now perform custom calculations of your own.
For example, JSL has a straightforward notation for entering matrices:

\[
L = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6,
7 & 8 & 9 & 10 & 11 & 12
\end{bmatrix};
\]

\[
M = \begin{bmatrix}
0 & 1 & 2,
2 & 1 & 0,
0 & 1 & 1,
2 & 0 & 0
\end{bmatrix};
\]

\[
N = \begin{bmatrix}
1 & 2 & 3 & 4,
4 & 3 & 2 & 1,
0 & 1 & 0 & 1
\end{bmatrix};
\]

\[
O = \begin{bmatrix}
0 & 1 & 2,
2 & 1 & 0,
1 & 2 & 0
\end{bmatrix};
\]

Simple operations look like this:

\[
R = L + M; \quad // \text{matrix addition}
\]

\[
\begin{bmatrix}
1 & 3 & 5,
6 & 6 & 6,
7 & 9 & 10,
12 & 11 & 12
\end{bmatrix}
\]

\[
Q = L \times M^T; \quad // \text{matrix multiplication of } L \text{ by } M\text{-transpose}
\]

\[
\begin{bmatrix}
8 & 4 & 5 & 2,
17 & 13 & 11 & 8,
26 & 22 & 17 & 14,
35 & 31 & 23 & 20
\end{bmatrix}
\]

You can even solve a linear system, that is, find the vector \( x \) for the square, nonsingular matrix \( A \) and vector \( b \) you specify such that \( x = A^{-1}b \):

\[
A = \begin{bmatrix}
1 & -4 & 2,
3 & 3 & 2,
0 & 4 & -1
\end{bmatrix};
\]

\[
b = [1, 2, 1];
\]

\[
x = \text{solve}(A, b);
\]

\[
[-17, 5, 19]
\]

For numerical work needing matrix factors, JMP offers operations such Cholesky, eigenvalue, and singular-value decomposition:

\[
E = \begin{bmatrix}
5 & 4 & 1 & 1,
4 & 5 & 1 & 1,
1 & 1 & 4 & 2,
1 & 1 & 2 & 4
\end{bmatrix};
\]

\[
L = \text{cholesky}(E);
\]

\[
\begin{bmatrix}
2.2360679774998 & 0 & 0 & 0,
1.788543819998 & 1.3416407864999 & 0 & 0,
0.4472135955 & 0.1490711985 & 1.9436506316151 & 0,
0.4472135955 & 0.1490711985 & 0.91465912076 & 1.7149858514251
\end{bmatrix}
\]

You can read the numeric columns of a data table into a matrix to perform custom calculations:

\[
\text{my matrix} = \text{current data table()}<<\text{get as matrix};
\]

Regression example

Teachers could use JSL’s matrix algebra capabilities in the classroom to demonstrate how linear regression models are estimated. Here is a simple example with just three observations for a response \( Y \) and predictor \( X \):

\[
Y = \begin{bmatrix}
4 \\
5 \\
7
\end{bmatrix}, \quad X = \begin{bmatrix}
1 \\
3 \\
5
\end{bmatrix}
\]

You want to solve for \( \beta_0 \) and \( \epsilon \) in the equation \( Y = \beta_0 + \beta_1X + \epsilon \), which can be rewritten as \( Y = \beta X + \epsilon \), where \( \beta \) is the matrix composed of vectors \( \beta_0 \) and \( \beta_1 \), and \( X \) is a vector of ones for the constant term and the vector of measurements for the linear term, \( X \). The task is to solve for \( \beta \):

\[
Y = [4, 5, 7];
\]

\[
X = [1, 1, 1] \quad // \text{the constant term}
\]
The heart of the least squares technique is to minimize estimates of $\beta$, denoted $b$, by solving the normal equation:

$$b = (X^TX)^{-1}X^TY$$

First rewrite this expression in JMP's matrix notation:

```javascript
b = inverse(x'y*(x'y));
```

Now find fitted values of $Y$, denoted $\hat{Y}$, by multiplying $b$ and $X$:

```javascript
Yhat = X*b;
```

The model's residuals are the differences between the $Y$ values and the $Y$ estimated values:

```javascript
e = Y - Yhat;
```

You see the results at each step when you submit the script one line at a time, but normally you would probably run the whole script at once and use a `Show` command at the end:

```javascript
show(b, yhat, e);
```

JMP's Fit Model platform gets the same results:

```javascript
dt = new table("myData.jmp");
dt << new column("Y", values([4,5,7]));
dt << new column("X", values([1,3,5]));
myFit = Fit Model(Y( :Y), Effects( :X), Personality(Standard Least Squares), Run Model);
myFit << save columns(residuals);
```
dt = new table("myData.jmp");
dt << new column("Y",values([4,5,7]));
dt << new column("X",values([1,3,5]));
myFit = Fit Model(Y(:Y), Effects(:X), Personality(Standard Least Squares), Run Model);
myFit << save columns(residuals);

An Add Rows command is not necessary, since new rows are automatically created with the values in the New Column command. If you had inserted an Add Rows command before adding columns, the default column Column 1 would have persisted in the data table.

A more elaborate regression example appears in “Statistical examples,” p. 370, in the “Matrices” chapter.
Scripting graphs

Through scripting, you can extend JMP's built-in graphic displays or create your own graphs from scratch. For example, here is a JMP script that shows how kernel density estimates are formed by adding kernel functions together.

The idea of a kernel density estimate is to show a smooth shape for a set of data points, which can help assess whether the points come from a normal distribution. A common method is to look at histograms. However, even with JMP's hand tool for adjusting histograms' bin sizes interactively, it can be difficult to determine which choices best convey the shape of the underlying data. One problem is that the jagginess caused by a histogram's squared-off bins can be distracting.

This example draws a normal density curve at each data point with some standard deviation given by $\sigma$. Next it adds together the heights of each curve to draw one summation curve. Finally, the script adds a **Handle** control for setting the $\sigma$ value interactively.

```jsl
// Kernel Density Estimate Demonstration

data = [5,7,8,9.5,10,10.5,11,11.2,12,13,15,17,18];
sigma = 1.5;

i=NewWindow("Kernel Addition",
            GraphBox(FrameSize(400,300), double buffer,
                     XScale(min(data)-3*sigma, max(data)+3*sigma), yScale(0,2.5),
                     n = nrow(data);

                     // draw gray kernels at each data point
                     pen color("gray");
                     for(i=1,i<=n,i++,
                        xx = data[i,1];
                        YFunction(Normal Density((x-xx)/sigma)/sigma, x);
                     );

                     // draw a red curve showing sum of kernels
                     Pen Color("red");
                     YFunction(summation(i=1, n,
                                          Normal Density((x-data[i,1])/sigma)/sigma), x);

                     // add a black "slider" handle for interactively controlling the
                     // width of the underlying kernels
                     PenColor("black");
                     Handle(sigma, 2.25, sigma=x);
                     text({sigma,2.3},"sigma=",sigma,"; drag handle to change")

                     ));

Here is the resulting graph using the initial setting, $\sigma=1.5$.```
Figure 1.11 Kernel Addition: sigma=15.

The **Handle** is the small black marker at the top of the graph; click and drag this to change the sigma value. Here are a few snapshots after dragging to other sigma values:

**Figure 1.12** Kernel Addition: Handle Dragged to the Left
You can try data of your own as well, but make sure to adjust the coordinates for Handle to suit the range of your data points. Foster, Stine, and Waterman’s illustration use these data, which you might try substituting into the script above:

```javascript
data = [0,3,5,6,7,9]; // Foster, Stine, Waterman's data
```

For more about graph scripting, see “Scripting Graphs,” p. 277.

**Scripting instructional simulations**

The previous example, “Scripting graphs,” p. 23, showed JSL’s capability to make interactive graphics—scientific graphics that respond to a user’s clicking and dragging. This capability presents exciting opportunities for educators.

For example, many basic statistical methods use $t$-statistics—tests that rely on the $t$ distribution and the fact that a $t$ distribution with sufficiently many degrees of freedom approximates a normal distribution. The degrees of freedom for a test is defined as the number of subjects in the sample (the sample size, often denoted $n$), minus the number of parameters in the test. For example, a $t$-test compares the means of two groups, so a $t$-test has $n - 2$ degrees of freedom.

Again, the $t$ distribution with sufficiently many degrees of freedom approximates a normal distribution.

How many degrees of freedom are sufficient? In other words, how many subjects do you need in your random sample to get reliable results?

Many textbooks state the authors’ favorite values for $n$ without much explanation, and many authors’ favorite number seems to be 30. Do you need a sample size of 30 before you can use statistics based on the normal distribution?

This script plots the density function for a normal distribution in blue, then draws the density function for a $t$ distribution with just one degree of freedom. There is also a handle that you can click and drag to change the value for $df$ and see how it affects the shape of the $t$ distribution’s density function.

```javascript
df=1;
```
New Window("t-distribution",
GraphBox(Framesize(300,150), Xscale(-5,5), Yscale(0,.5),
  Double Buffer,
  pen color("blue"); YFunction(Normal Density(x), x);
  pen color("red"); YFunction(t density(x, df),x);
  text({0, df/80+0.01}, "df = ",df);
  Handle(0, df/80, df=floor(80*y)+1)));

Figure 1.14 t-Distribution: df=1

Certainly there appears to be big differences between these two curves, especially at the tails.

Now click the red marker next to the “df = 1” label, and drag that handle up and down in the graph.

Watch how the shape of the distribution changes as df changes.

Here are some snapshots. At df = 5, the similarity is pretty encouraging.

Figure 1.15 t-Distribution: df=5

At df = 14, it’s hard to tell the difference.
Figure 1.16  t-Distribution: df=14

At $df = 30$, the plots are almost identical.

Figure 1.17  t-Distribution: df=30
JMP is scripted by a very simple language called JMP Scripting Language, or JSL. You may not need to ever learn JSL, because almost every feature in the product is accessible through a direct user interface, as well as through the scripting language. Even if you use JSL, you can usually get JMP to write the scripts for you rather than typing them in yourself. JSL will be most useful to the power user that wants to extend JMP past its normal operating realm, or for the production user who wants to automate a regularly scheduled analysis.

JSL is used in many places in JMP internally.

- Column Formulas are implemented internally in JSL.
- Platforms are launched using JSL internally.
- Platforms are interactively modified using JSL internally.
- Some graphics are performed through JSL.

This chapter shows how to recognize valid JSL in terms of how the language is written in text. It covers syntax. The next chapter, “JSL Operators,” p. 59, details many of the basic functions of JSL. The chapter “Programming Functions,” p. 91, discusses how to write meaningful scripts that do something useful. It covers semantics. First you have to learn how to write words and position commas and operators. Then you learn what the words mean and how to use them.

Confusion alert! As you will learn in the section on logical operators, a single pipe symbol (|) represents a logical OR. In the interests of brevity, programming and scripting manuals commonly use a | to represent the word or when discussing alternative values.

For example, a filepath can be either absolute or relative. So when we show an argument to a filepath function as \texttt{absolute|relative}, this means that you enter either \texttt{absolute} to indicate an absolute filepath, or \texttt{relative} to indicate a relative filepath. More than two options can be strung together with an or pipe in this way.

So, when you see words separated with a |, read it as or.
Contents

First JSL Script ................................................................. 31
The JSL Language .......................................................... 31
  Lexical rules of the language ......................................... 32
  Data elements ................................................................ 37
  Operators ...................................................................... 38
Context: meaning is local .................................................. 40
  Programming vs. scripting ............................................. 40
  Data table context ......................................................... 41
  Scoping operators .......................................................... 42
  Graph context ................................................................ 42
Name resolution ................................................................. 42
  Name-binding rules ......................................................... 43
  Frequently-asked questions ............................................ 45
  Function resolution rules ................................................ 46
Using the Script Editor and Debugger ................................. 46
  The Script Editor ............................................................ 47
  The JSL Debugger .......................................................... 52
Help with JSL ................................................................. 54
  JSL Browsers ................................................................. 54
  Show Commands ........................................................... 55
  Show Properties ............................................................ 56
First JSL Script

Go to the JMP Starter window, and click the New Script button. In the resulting text window, enter the text:

X=12+8;
A="Hello, World";
Show(X,A);

Now select the text in that window by dragging the mouse cursor over it. Select Run Script from the Edit menu (or press Control-R on Windows, or ⌘-R on Macintosh). The result should appear in a Log text window.

X:20
A:"Hello, World"

This is how you enter and submit JSL. You have just created two global variables, X and A, and given values to them.

The JSL Language

JSL consists entirely of a nested fabric of message names, with message contents enclosed in parentheses:

Message Name ( argument 1, argument 2, ... )

JSL expressions hold data, manipulate data, and send commands to objects.

The meaning of JSL phrases is dependent on who the message is sent to. The same name might mean one thing in one context and something entirely different in another context.

Almost anything that obeys certain punctuation rules, such as matching parentheses, is a valid JSL expression, though it might not be understood by the object it is sent to.

Here is a valid JSL expression:

Dialog(
    Title ( "A Box" ),
    "Hello, World",
    Text box ( "-----" ),
    Button ( "OK" )
);

Notice the following:

- Names can have embedded blanks.
- Message contents are enclosed in parentheses, which must be balanced.
- Items are separated by commas.
- You can use UPPER-CASE and lower-case characters interchangeably.
- Messages are commonly nested inside other messages.
Lexical rules of the language

The language consists of the following kinds of tokens. Each are discussed briefly in subsections below.

Table 2.1 Kinds of tokens in JSL

<table>
<thead>
<tr>
<th>Token</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commas and parentheses</td>
<td>, and ()</td>
</tr>
<tr>
<td>Names</td>
<td>sqrt Distribution</td>
</tr>
<tr>
<td></td>
<td>Net Income</td>
</tr>
<tr>
<td>Numbers and Dates</td>
<td>12.3</td>
</tr>
<tr>
<td></td>
<td>1.24E9</td>
</tr>
<tr>
<td>Quoted strings</td>
<td>&quot;ABC&quot;</td>
</tr>
<tr>
<td>Matrices</td>
<td>[1 2, 3 4]</td>
</tr>
<tr>
<td>Lists</td>
<td>{oil,vinegar}</td>
</tr>
<tr>
<td>Associative Arrays</td>
<td>{{&quot;gps&quot;, [78.7812 35.7873]},{&quot;high schools&quot;,{&quot;Cary&quot;, &quot;Green Hope&quot;, &quot;Panther Creek&quot;}},{&quot;population&quot;, 127640},{&quot;state&quot;, &quot;NC&quot;},{&quot;weather&quot;,{&quot;sunny&quot;, &quot;warm&quot;}}}</td>
</tr>
<tr>
<td>Operators</td>
<td>+ - */ etc.</td>
</tr>
<tr>
<td>File Paths</td>
<td>&quot;$SAMPLE_DATA/Big Class.jmp&quot;</td>
</tr>
<tr>
<td>Comments</td>
<td>// the rest of a line after // is a comment</td>
</tr>
<tr>
<td></td>
<td>/* these symbols start and stop comments of any length */</td>
</tr>
</tbody>
</table>

**Commas**

Commas (,) separate items, such as items in lists, rows in matrices, or named arguments to a function.

```julia
mylist={1,2,3};
yourlist=List(4, 5, 6);
mymatrix=[[3 2 1,0 -1 -2]];
```

**Note:** when you have a sequence of commands to execute, you do not separate them with commas. Rather, you glue them together with semicolons, as discussed under “Glue,” p. 93.

**Parentheses**

Parentheses () are used to group operations in an expression and to delimit arguments to a function, such as in `root(49)`. Parentheses also delimit the end of a function name even when arguments are not needed. Consider `e()` below, where the empty argument "( )" after "e" is what distinguishes JSL's function for e from the name "e."

```
for(i=0,i<10,i++,show(i,e()^i));
```

Be careful that parentheses match—every ( needs an ), or else errors result.

On Windows, the script editor can match fences (parentheses, brackets, and braces). Press Control-] with your cursor in any part of a script. The editor searches for fences, highlighting the text between the first set of opening and closing fences it finds. Repeat this process to highlight the next-higher fence.
Names

A name is exactly what you think: something to call a thing. For instance, when you assign the numeric value 3 to a global variable in the statement `a=3`, “a” is a name. Commands and functions have names, too. In the statement `Log(4)`, the word “Log” is the name of JMP’s logarithm function. A name is any language token that isn’t a number or a string or a special symbol operator (such as + or – or ++ or ^).

Names have a few rules:

- Names must start with an alphabetic character or underscore, and can continue with those as well as numeric digits, spaces, tabs, vertical tabs, line and page delimiters, double-byte characters, apostrophes, percent signs, periods, and backslashes:
  
  a-z A-Z 0-9 _ ' % . 

- When comparing names, the white-space characters (like blanks, tabs, and newlines) are ignored, and upper and lower case is not distinguished. For example the names `Forage` and `for age` are equivalent.

- Actually, you can still have a name that is any other sequence of characters, but if it doesn’t obey the rules above, it needs to be quoted and placed inside a special parser directive called `Name( )`. For instance, to have a global variable with the name `-(*)$/%(*&$%A`, you have to put it inside quotation marks and `Name( )` every time you use it in JSL:
  ```julia
  Name("-(*)$/%(*&$%A")=42;
  foo=4; print(foo+Name("-(*)$/%(*&$%A"));
  46
  ```
  
  `Name` is harmless when it isn’t needed; for example, `foo` and `Name("foo")` are exactly the same thing.

Numbers

Numbers can be written as integers, decimal numbers, in scientific notation with an E preceding the power of ten, and as dates, times, or date/time values. A single period by itself is the missing numeric value (sometimes called NAN for “not a number”).

For example, these are all numbers:

```
.   1   12   1.234  3E3  0.314159265E+1  1E-20
```

If you precede a number with a minus sign, the minus sign is usually interpreted as a prefix operator, not part of the number. You can follow a number immediately with an E followed by a power of ten you want to scale the number by. For example `3E2` means 3 times 10 to the power 2, or 300. If you need a negative exponent of ten in E notation, that minus sign is part of the number.

Dates and times

JMP supports date/time values in a variety of common date/time notation formats. Date/time values are stored and computed as a number of seconds since midnight, January 1, 1904. However, you can enter a date/time value literally, using the format `ddMonyyyy:hh:mm:ss`, as in:
JSL Building Blocks
The JSL Language

x = 14Feb2002:19:15:00;

Several shorter forms can also be used for date/time literals:

x = 14Feb2002:19:15;
x = 14Feb2002;
y = 19:15:00;
y = 19:15;

JMP has numerous operators for converting date/time values to strings in common notations, e.g.:

invitation = "Would you like to dine with me on "
|| long date(x) ||"?";

"Would you like to dine with me on Thursday, February 14, 2002?"

These and other date/time subjects are detailed in “Date/Time Operators,” p. 79 in the “JSL Operators” chapter.

Quoted Strings

Strings are put in double quotes. Be careful to put in the end quote, or it will gobble up unintended text until it finds the next double quote.

How do you put a double quote inside a quoted string? Inside quoted strings, you can use a special escape sequence \! (backslash-bang) to precede a code for special characters. For example, code \\
\"!\" to mean the character string containing a double quote.

Note: The null character is dangerous to use, because it is normally treated as the end of the string.

Table 2.2  Escape Sequences for Quoted Strings

<table>
<thead>
<tr>
<th>!b</th>
<th>!t</th>
<th>!r</th>
<th>!n</th>
<th>!N</th>
<th>!f</th>
<th>!0</th>
<th>&quot;t&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank</td>
<td>tab</td>
<td>carriage return only</td>
<td>linefeed (newline) only</td>
<td>inserts line breaking characters appropriate for the host environment</td>
<td>formfeed (page break)</td>
<td>null character (see warning)—type the number zero, not the letter O.</td>
<td>double quote</td>
</tr>
</tbody>
</table>

a. On Macintosh, this is CR (carriage return character, hex ’0D’). On Linux, this is LF (line feed character, hex ’0A’). On Windows, this is CR LF (carriage return followed by a linefeed, hex ’0D0A’).

Sometimes, long passages require a lot of escaped characters. In these cases, use the notation \[...\] and everything between the brackets does not need or support escape sequences. Following is an example where \[...\] is used inside a double-quoted string.

jslPhrase = “The JSL to do this is :\[
  a = “hello”;
  b = a||” world.”;
  show(b);
]\ and you use the Submit command to run it.”;
Matrices

Matrices can be specified as signed numbers inside brackets, with values separated by blanks or other white space characters, and rows separated by commas or semicolons. Brackets are also used to represent subscripting, when brackets occur after another expression. You can represent an empty matrix of no rows and no columns by \[
\]
A=[1 2, 3 4];
two=A[1,2]; // subscript picks the first row, second column element from A
Empty=[];

Matrices are discussed at greater length in the "Matrices" chapter.

Lists

Like matrices, lists are compound data structures, but they are more flexible than matrices. Lists are a way to store numerous items of different type: numbers, text strings, expressions, matrices, and even other lists. Lists can be expressed as arguments for the function List( ), or simply between \{\} curly braces:
A = List(1,2,B,sqrt(3), List(a,b,3));
A = \{1,2,B,sqrt(3),\{a,b,3\}\};

Lists are discussed at greater length under "Lists," p. 103 in the "Programming Functions" chapter.

Associative Arrays

An associative array maps unique keys to values (possibly non-unique). Other languages call this type of data structure a "dictionary" a "map", a "hash map" or a "hash table". A key is a single quote string, while the value associated with that key can be strings, numbers, dates, matrices, lists, etc. For example:

cary=Associative Array();
cary["state"] = "NC";
cary["population"] = 127640;
cary["gps"] = [78.7812 35.7873];
cary["weather"] = {"sunny", "warm"};
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};

Associative arrays are discussed at greater length under "Associative Arrays," p. 107 in the "Programming Functions" chapter.

Operators

Operators are one- and two-character symbols for common arithmetic actions. Operators come in several varieties: infix (with arguments on either side, such as + in 3+4, or = in a=7), prefix (with one argument on its right side, such as !a for logical negation), or postfix (with one argument on its left side, such as a++ for incrementing a).

JSL’s operators all have Function() name equivalents, also, and a complete list appears in the “JSL Operators” chapter. These are the operators, listed in order of precedence (the order in which operations are performed):

\{
\}, [ ], ++, --,
^,
-, !,
File Paths

In JMP, the preferred file path format is the POSIX (aka UNIX) format, with forward slashes as separators. Each host still accepts its native format for compatibility. This, along with path variables, often eliminates the need for if(host is(...)) logic to open files in a portable script.

Path variables are supported at the beginning of a POSIX path. JMP recognizes HOME, DOCUMENTS, SAMPLE_DATA, SAMPLE_IMPORT_DATA, SAMPLE_SCRIPTS, and TEMP as path variables. They are used with a dollar sign at the beginning of a path.

Open("$SAMPLE_DATA/Big Class.jmp")

Users can also add their own path variables, or override some of the built-in ones with the JSL functions

posixPath = Set Path Variable(varName, posixPath);
posixPath = Get Path Variable(varName);

varName is case-sensitive and does not include the dollar sign.

Confusion Alert: You cannot override HOME, DOCUMENTS, or TEMP, and Get Path Variable does not retrieve the settings for them. Instead, use this JSL function:

posixPath = Convert File Path("$HOME");

There are JSL functions for accessing the default directory:

posixPath = Set Default Directory(posixPath); // resulting current dir is returned
posixPath = Get Default Directory();

The default directory is used for resolving relative paths. The file search list is also used for resolving relative paths for opening files (not for saving).

You can convert among file paths using the Convert File Path command.

Convert File Path (path, <absolute|relative>, <POSIX|windows>, <base(dir_path)>);

For the <optional> arguments above, the defaults are absolute, POSIX, and a base path of the default directory. The input path may be in Windows or POSIX style.
Comments

Comments are notations that you add to your code that are ignored by the parser. Comments can be started by either // or /*. The // form continues until the end of a line. The /* form continues until a closing */. Comments cannot be inside quoted strings.

Table 2.3 Comments in JSL scripts

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>//</td>
<td>// comment</td>
<td>Start a comment line; doesn’t have to be at beginning of line, but</td>
</tr>
<tr>
<td></td>
<td></td>
<td>everything following to the end of the line is a comment.</td>
</tr>
<tr>
<td>/* */</td>
<td>/* comment */</td>
<td>A comment that can appear in the middle of a line of script. Script text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>before and after the comment is undisturbed. The /<em>...</em>/ punctuation is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a convenient way to remove portions of script temporarily. The following</td>
</tr>
<tr>
<td></td>
<td></td>
<td>are equivalent:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c=1+ /* comment */ 2;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c=1+2;</td>
</tr>
</tbody>
</table>

Note: Add //! to the beginning of a script to prevent it from being seen by users of the script.

Data elements

Generally the language tokens translate into JSL data elements and expressions. JSL uses the following basic elements in the language:

- integers
- numbers (floating point)
- character strings
- names
- lists—a list holds a number of other values, including nested lists and expressions
  
  Doe family = List( John Doe, Jane Doe, Baby Doe );
  Doe family = { John Doe, Jane Doe, Baby Doe };

- associative arrays—an associative array maps keys to values, which can be just about any other data element
- matrices—a matrix is a row-by-column table of numbers
  
  Design Matrix = [1 1, 1 -1];

- expressions—expressions can be treated as complex data and manipulated by certain operators
- references to scriptable objects:
  
  **data table reference** obtained from the Open() or Current Data Table() function (represented by dt in syntax summaries)
  
  **data column reference** obtained from the Column() function (represented by col in syntax summaries)
  
  **scriptable object reference** obtained from platform launches or Scriptable class subscripting (represented by obj in syntax summaries)
  
  **displayBox reference** reports or parts of reports in windows (represented by db in syntax summaries)
Operators

In order to make writing algebraic expressions natural, JSL has adopted certain special character operators, which are translated into the same meaning as if the phrase had been written as a message or function. For example, the following two statements are equivalent:

\[
\text{Net Income After Taxes} = \text{Net Income} - \text{Taxes};
\]
\[
\text{Assign(Net Income After Taxes), Subtract(Net Income, Taxes))}
\]

The assignment operation can be written either as a function Assign or as an infix operator =. Similarly, subtraction can be done with the Subtract function, or the infix minus sign; they are equivalent inside JMP.

Another common operator is the semicolon (;). The semicolon is a gluing operator that is used to both separate yet join one expression to another in a programming sequence. The function equivalent of this is Glue, so \(a;b\) is the same as \(\text{Glue}(a, b)\). The semicolon or Glue operator returns the result of its last argument. It is also legal to end an expression with a semicolon. This may lead you to think of it as a statement terminator like some other languages, but it is designed as an infix operator. Terminating semicolons are allowed at the end of a script stream and before a closing parenthesis or closing brace: \), or \}.  

Following are operators and their Function( ) equivalents. The operators are grouped in their order of precedence, where the binding priority decreases with each group. For example, in \(a^b+c\), the multiplication \(a^b\) is done before the addition of \(c\). For details, see “JSL Operators” chapter.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>List</td>
<td>Construct a list.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Subscript</td>
<td>Subscripts identify specific elements within a data element (a), where (a) could be a list, a matrix, a data column, a platform object, a display box, etc.</td>
</tr>
<tr>
<td>++</td>
<td>Post Increment</td>
<td>Adds one (1) to (a), in place.</td>
</tr>
<tr>
<td>--</td>
<td>Post Decrement</td>
<td>Subtracts one (1) from (a), in place.</td>
</tr>
<tr>
<td>^</td>
<td>Power</td>
<td>Raise (a) to exponent power (b). With only one argument, 2 is assumed as the power, so Power(x) computes (x^2).</td>
</tr>
<tr>
<td>-</td>
<td>Minus</td>
<td>Reverses sign of (a).</td>
</tr>
<tr>
<td>!</td>
<td>Not</td>
<td>Maps nonzero values to 0, maps 0 values to 1.</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
<td>Multiplies (a) by (b).</td>
</tr>
</tbody>
</table>
### Table 2.4 Operators and function() equivalents, in precedence order (Continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>*:</code></td>
<td><code>a:*b</code></td>
<td>Elementwise multiplication for matrices <code>a</code> and <code>b</code>.</td>
</tr>
<tr>
<td><code>/</code></td>
<td><code>a/b</code></td>
<td><code>Divide(a, b)</code> divides <code>a</code> by <code>b</code>.</td>
</tr>
<tr>
<td></td>
<td><code>Divide(a, b)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>Divide(x)</code></td>
<td><code>Divide(x)</code> interprets the argument as a denominator and implies 1 as the numerator, yielding the reciprocal <code>1/x</code>.</td>
</tr>
<tr>
<td><code>/:</code></td>
<td><code>a/b</code></td>
<td>Elementwise division for matrices <code>a</code> and <code>b</code>.</td>
</tr>
<tr>
<td><code>+</code></td>
<td><code>a+b</code></td>
<td>Adds <code>a</code> and <code>b</code>.</td>
</tr>
<tr>
<td><code>-</code></td>
<td><code>a-b</code></td>
<td>Subtracts <code>b</code> from <code>a</code>.</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>`</td>
</tr>
<tr>
<td></td>
<td><code>Concat(a, b)</code></td>
<td>Horizontally concatenate matrices.</td>
</tr>
<tr>
<td></td>
<td>`Concat(matrix1</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>Concat(matrix1, matrix2)</code></td>
<td></td>
</tr>
<tr>
<td><code>\</code></td>
<td><code>matrix1||matrix2</code></td>
<td>Vertically concatenate matrices.</td>
</tr>
<tr>
<td><code>::</code></td>
<td><code>a::b</code></td>
<td>For matrices, generates the integers from <code>a</code> to <code>b</code>.</td>
</tr>
<tr>
<td></td>
<td><code>Index(a, b)</code></td>
<td>(Colons are also used as infix operators for scoping, where <code>:a</code> means data table column <code>a</code>, and <code>::a</code> means JSL global variable <code>a</code>. See “Global scoping operator,” p. 45.)</td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td><code>object &lt;&lt; message</code></td>
<td>Send <code>message</code> to <code>object</code>.</td>
</tr>
<tr>
<td><code>==</code></td>
<td><code>a==b</code></td>
<td>Booleans to compare values. Return 1 if true, 0 if false.</td>
</tr>
<tr>
<td><code>!=</code></td>
<td><code>a!=b</code></td>
<td>Missing values in either <code>a</code> or <code>b</code> propagate missing values.</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td><code>a&lt;b</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td><code>a&lt;=b</code></td>
<td></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>a&gt;b</code></td>
<td></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td><code>a&gt;=b</code></td>
<td></td>
</tr>
</tbody>
</table>
The JSL language is used both for scripting and programming. Programming emphasizes results, which build from the inside out. Scripting emphasizes context, which builds from the outside in. Understanding this distinction is crucial to understanding how JMP executes a script.

### Programming vs. scripting

For example, consider how a typical programming statement evaluates:

\[ x = \log(a \cdot b \cdot (c + d)) \; ; \]

The goal of this program is to find a result to assign to \( x \). According to the rules of precedence for arithmetic operators, this starts on the inside by evaluating \( c \) and \( d \) and adding them together. Next \( b \) is evaluated and then raised to that power \((c + d)\), and that result in turn is multiplied by the evaluation of \( a \).
That result is next passed to the Log function, which finds the logarithm, and finally that result is assigned to x. Evaluation started on the inside with c + d and worked its way out.

Now consider a typical scripting statement:

Fit Model(Effects(a, b, a*b), Y(c), Run Model);

The goal of this script is to launch an analysis platform and return a report. Evaluation starts on the outside with Fit Model, which calls JMP’s platform for fitting models. This platform in turn knows that Effects is specifying the independent terms of a model, and in that context, the phrase a*b takes on a whole new meaning. In the program above, it meant to multiply, but here a*b means the interaction term of a and b, which in turn are evaluated as the data columns assigned to the global variables a and b. The platform also knows that Y means that the column assigned to c is to be the response term. Finally the platform sees Run Model and goes to work. Evaluation started on the outside with Fit Model and worked its way in.

**Data table context**

A formula column is evaluated down the rows of the data table. If a script needs to refer to a specific cell each time, then you give a subscript to a specific row number. For example, this formula column gives the ratio within each row, where height and weight are columns such as in the sample data table Big Class:

new column("Ratio", formula(Height/Weight));

This gives the ratio of each row’s Height to the Weight in row 1:

new column("Ratio B", formula(Height/Weight[1]));

There is a current row number that determines the cells of the data table, which can be set or displayed with the operator Row(), e.g.

Row();  // returns current row number
Row()=3;  // makes the third row current

By default, the current row is always 0, which means that the following statement on its own would return a missing value and be generally useless:

ratio = height/weight;

But if you include something in the script to set the current row to an actual row, say the third, you’ll get a result assigned to the global variable ratio.

row()=3; ratio = height/weight;

Another possibility is to ignore the current row and instead use subscripts to ask for a row specifically. This version divides the height in row 3 by the weight in row 4:

::ratio = height[3]/weight[4];

Still another possibility is to create a column ratio and iterate the division a row at a time down the whole column:

new column("ratio");
for each row(:ratio = height/weight);
If you’re working with global variables, pre-evaluated statistics, and so forth, there’s no need to consider the current row. (Pre-evaluated statistics are single number values computed from the data table, as discussed in “Pre-evaluated statistics,” p. 170 in the “Data Tables” chapter.)

::myCalc = col mean(height) / col mean(weight);

### Scoping operators

The scripts in the previous section used colons : or double colons :: in front of some names. These are scoping operators that tell JMP how to interpret the names that follow in cases that could be ambiguous, such as when you have both a data table column and a JSL global variable with the same name. Scoping operators and the rules for name resolution are discussed in detail under “Name resolution,” p. 42, later in this chapter.

:x; // data table column
::x; // JSL global variable
x; // depends on state when first used

### Graph context

If the script is inside a graph (such as the plot at the top of a Bivariate platform), then the graphics commands are processed by their containing Graph Box. You can store graphics commands, intermixed with expressions, inside any graphics frame. For example, if you wanted to superimpose four sine waves inside a graph, you could right-click (Windows) or Control-click (Macintosh) the graph, and then enter the following JSL into the dialog:

For(i=1, i<=4, i+=.1, YFunction(Sin(x/i),x));

Or if you wanted to make a separate graph of the sine waves, you would enclose the command inside a Graph Box command, which constructs a new graph, and in turn enclose the graph in a New Window command:

new window("Sinusoid",
Graph Box(FrameSize(500,180),XScale(-10,10),yScale(-1,1),
For(i=1,i<=4,i+=.1,YFunction(Sin(x/i),x))));

In either case, it is up to the graph box to determine what x is.

### Name resolution

The following kinds of objects can be identified by name:

- Columns and table variables in a data table
- Global variables which exist for a session
- Scriptable object types
- Parameters and Locals inside formulas
If a name is just part of a script for an object, then its meaning is usually the name of an option or method in the object. Otherwise it is considered by the evaluator using the rules in this section. If the name is followed by parentheses, then it is regarded as a function. Otherwise, the name is looked up as some object.

Global variables are names that exist for the remainder of a session to hold values. Globals can contain many types of values, including numbers, strings, lists, and references to objects. They are called globals because they can be referred to almost anywhere, not just in some specific context.

Most of the time, you can just use an object’s name directly to refer to the object.

```plaintext
Ratio = Height/Weight;
N = N + 1;
Distribution (...)
```

Now the trick is to learn the rules as to when you are referring to a column, when to a global, and when to a type, so that you know when you can use the name directly and when you need to qualify it somehow.

Name-binding rules

From a programming point of view, name resolution is important when getting and setting values in a script. Names that are referred to for name-reference, rather than to get or set a value, are not resolved. Rather they are just passed along, as with a script to launch a platform specifying certain columns to be analyzed.

There are six possible resolutions for any name in a script. JMP tries each of the following in order:

1. if followed by a pair of parentheses ( ), look it up as a function; see “Function resolution rules,” p. 46, for details;
2. then if not prefixed by ::, look it up as a global variable;
3. then if not prefixed by ::, look it up as a data table column or table variable;
4. then look it up as a local variable;
5. then look it up as a platform launch name (e.g. Distribution, Bivariate, Chart, ...);
6. then, if used as the target of an assignment (as an L-value), create and use a global variable.

A name in a script is resolved the first time it is used to get or set a value, and its resolution persists thereafter. The operators : and :: are always resolved respectively as As Column() and As Global().

Exceptions

If a name is resolved to a column in a data table that is closed, then it re-resolves the next time it gets or sets in case the data table has been opened in the meantime. Other exceptions are function definitions, column formulas, and nonlinear formulas.

When an unscoped name refers to a table column

When an unscoped name is resolved to get or set a value, then it refers to a column in a data table, rather than a global

- if no global, local, or parameter of that name already exists,
Name resolution

- and the data table in context has a column of that name,
- and
  - either the current row is set to a positive value,

**Warning:** The current default row is now 0 instead of 1, as in JMP 3 and earlier.

- or the name is subscripted, e.g. A[i].

If the data table has a table variable by that name, then the table variable takes precedence. In all other cases, it binds to a global, local, or parameter.

**Note:** When an unqualified name in JSL is resolved to a data column in a data table, it must be associated with the current data table. This is different than the rule in JMP 4 and earlier. The earlier rule was that if a name in a script was resolved to a data column, then it would stay with that data column even if the data column was not associated with the current data table.

**Exceptions**

Column formulas and nonlinear formulas.

**Column scoping operators**

Column scoping operators can be used to force names to be resolved as columns.

1. The prefix colon (:) means that the name refers to a table column or table variable only, never a global. The prefix : refers to the current data table context. The current data table is the data table that Current Data Table() either returns or is assigned.

   :colName

2. The infix colon (: ) operator extends this notion to specify which data table has the column by using a data table reference (data table references are discussed in the chapter “Data Tables,” p. 113). The function equivalent is As Column.

   dataTableRef:name;
   As Column(dataTableRef, name)

Therefore, these are equivalent:

   :name;
   CurrentDataTable():name;
   As Column(CurrentDataTable(), name);

The Column function can also be used, but it always evaluates its arguments, and you need to specify a row to make it refer to a cell rather than the whole column.

   Column("X");    // refers to column X.
   Column("X",12); // refers to the cell of row 12, column X.
   Column("X")[i]; // refers to the cell of the current row of column X
   Column(a);      // evaluates a and looks up the column of the result.
**Global scoping operator**

A global scoping operator can be used to force names to be resolved as globals. The prefix double-colon (::) means that the name refers to a global only, never a table column. The function equivalent is AsGlobal.

```plaintext
::globalName;
AsGlobal(globalName);
```

<table>
<thead>
<tr>
<th>Table 2.5</th>
<th>Scoping operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
<td>Syntax</td>
</tr>
<tr>
<td>As Column</td>
<td>::name</td>
</tr>
<tr>
<td>As Column</td>
<td>As Column(dt, name)</td>
</tr>
<tr>
<td>As Global</td>
<td>::name</td>
</tr>
<tr>
<td>As Global</td>
<td>As Global(name)</td>
</tr>
</tbody>
</table>

**Frequently-asked questions**

**Should you scope?**

Yes. When in doubt, scope—especially if you are writing scripts that define globals with the same names as data columns that you are not aware of, such as scripts that will be used by many people on a variety of data tables. Since global names always take precedence, you have to work to ensure that the column name references map to a column and that globals do not map to a column.

Prefix scope operators do not take run-time overhead after the first resolution. Infix scope operators always take run-time overhead.

**What is the difference between a column referred to by name, and a column reference?**

**If I have a column reference in a global, how do I assign a value to a cell in the column?**

Try submitting the following examples one line at a time to see how this works. These scripts assume that you have a data table with a column named “A” and you do not have a global named “A.” Each line begins with a `Row()`=3 assignment to set the current row to a valid row, in this case the third row. The current row is irrelevant in the last few examples.

```plaintext
// assumes a data table with a column "A" and sufficient rows
row()=3; A=1; //sets the current row of A to 1
row()=3; :A=2; //sets the current row of A to 2
row()=3; ::"A"=3; //sets the current row of A to 3
ColA=Column("A"); //sets the global ColA to refer to column A
ColA[3]=4; //sets the third row of A to 4
ColA<<Formula(6); //sets the formula for A to be 6
```
Will a scoping operator “stick” to its name?
Yes. Once you use a scoping operator with a name, that name continues to be resolved accordingly for all subsequent occurrences of the name. For example, if you want to protect against clobbering data tables that happen to contain a column named “i” when you use i as an index inside a for-loop, simply declaring ::i at the head of the script is all the protection you need for the whole script.

```js
::i=1;
for(i=1, i<10, i++, doScript);
```

Normally this wouldn't be a problem since at time of execution Row() would typically be 0. However, if Row() is greater than 0, such as inside a For Each Row loop, you should be careful:

```js
::i=1;
for each row(
    for(i=1, i<10, i++, doScript));
```

Which has precedence when scoping, "::" or "[]"?
Scoping occurs before subscripting. This means that these two lines of code are equivalent:

```js
dataTable:colName[i]
(dataTable:colName)[i]
```

Function resolution rules
A name followed by a parenthesized list of arguments is the syntax for a number of contexts in JSL:

- a call to a built-in function
- a call to a user-defined function
- a named argument, or
- a nesting in a script for an object: either an option with an argument, or a sub-object with messages, or any other use recognized by the object itself.

Using the Script Editor and Debugger

JMP provides both an editor and a debugger for writing and troubleshooting your JSL scripts.
The Script Editor

The Script Editor (as shown in Figure 2.1) provides a friendly environment for writing and reading JSL scripts.

Figure 2.1 The Script Editor

The script editor has several useful features:

- color-coding for JSL and SAS code;
- auto-completion for JSL operators (type Control-Space or ⌘-Space on Macintosh);
- tooltips when hovering over JSL operators;
- value-reporting when hovering over a global symbol;
- live brace matching;
- highlighting matching braces;
- find/replace using regular expressions;
- automatic formatting.

The script editor is also used in the log window and anywhere else you may edit or write a script.

Color-Coding

The script window uses the following colors for JSL:

- green for comments;
- blue for JSL operators;
- dark magenta for string values;
- dark cyan and bold for scalar values;
- black for everything else.
Colors may be customized in Preferences. See “Setting preferences for the script editor,” p. 50.

**Type-Ahead**

If you don’t remember the exact name of an operator, you can type part of the name and then type Control-Space or Control-Enter (⌘-Space or ⌘-Enter on Macintosh) to see a list of operators that match what you have typed so far.

For example, if you want to clear your JSL variables, but don’t remember the command, you can type clear, then Control-Space, to see a list of two possible clear commands (as seen in Figure 2.2).

**Figure 2.2** Type-Ahead Example

Click the command you want to replace what you typed with the command.

**ToolTips**

If you are using an operator and don’t quite remember its syntax or what it does exactly, you can hover over it to get a brief explanation (see Figure 2.3). This only works with JSL operator names. They are colored blue in the script editor.

**Figure 2.3** Tooltip for a JSL Operator

The tooltip shows the grammar, any arguments, and a brief explanation of the operator.

You can also hover over variable names to see their current value (see Figure 2.4).

**Figure 2.4** Tooltip for a JSL Variable
After running the first line, hovering over any instance of `myVar` shows that its value is 8. After running the third line, the value changes to `Eight`. If you hover over a variable before running the script, no tip appears. The variable has not yet been added to the list of globals and assigned a value.

**Brace-Matching**

The script editor helps you match parentheses, square brackets, and curly braces in two ways:

- the matching closing brace is added when you type an opening brace;
- when you place your cursor next to either an opening or closing brace, it and its match are highlighted in blue; if it does not have a match, it is highlighted in red;
- If you double-click a brace, everything between the matching braces is selected.

Figure 2.1 “The Script Editor,” p. 47 shows highlighted matching braces in the third line of code.

When you type an opening brace, add code in between, and then type the closing brace, the script editor allows you to type over the brace it added automatically for you, preventing you from accidentally adding an additional closing brace.

You can turn on and off the auto-completion of braces in the Preferences window. See “Setting preferences for the script editor,” p. 50 for details.

**Find/Replace**

The `Edit > Search` menu options are now available in scripts. Searching and replacing is the same as for data tables.

1. From the `Edit` menu, select `Search > Find`.
2. In the `Find what` field, enter the text you want to search for.
3. If you want to find text and replace it, enter the replacement text in the `Replace with` field.
4. Select any other checkboxes you want to use.
5. Click `Find` to find the next occurrence, or click `Replace` to replace the current occurrence and find the next.

See the `JMP User Guide` for details on the Search window.

**Automatic Formatting**

The script editor can format a script for easier reading. Any generated script (for example, by saving a platform script) is automatically formatted with tabs and returns in appropriate places.

If you open or write a script that is poorly formatted (for example, older saved JMP scripts that may have all commands strung together with no whitespace), you can have the script editor format it for you.

From the `Edit` menu, select `Reformat Script`.

**Tip:** This command alerts you if your script is badly formed; for example, if your parentheses aren’t matched.
Setting preferences for the script editor

You can customize several parts of the script editor. Open the Preference Settings window by selecting File > Preferences.

Setting the fonts

On the Fonts page, you can set the typeface and size for all script windows in JMP:

Figure 2.5 Changing the Font for Script Windows

Click Mono and set the font. For more details, see the JMP User Guide.

Setting editor preferences

On the Script Editor page, you can make many other customizations:

Use tabs Check this option to enable tabs in your scripts. If it is unchecked, any tab you type is replaced by spaces. This is on by default.

Tab width Enter how many spaces a tab should indent. If you have disabled tabs, any tab you type is replaced with this many spaces. The default value is 4.

Extra space at bottom of document Check this option to enable scrolling the last line of a script to the top of the script editor. This is on by default.

Auto-complete parentheses and braces Check this option to enable the script editor to automatically add closing parentheses, square brackets, and curly braces when you type an opening one. This is on by default.

Show line numbers Check this option to show the line numbers on the right side of the script editor. This is off by default.

Show operator tips Check this option to see tooltips for JSL operators. This is on by default.

Show indentation guides Check this option to see faint vertical lines that mark indentation. This is on by default.
Show variable value tips  Check this option to see tooltips for variable values. This is on by default.

Spaces inside parentheses  Check this option to cause the script editor to add spaces between parentheses, brackets, and braces and their contents for automatically formatted scripts. This is on by default.

Spaces in operator names  Check this option to cause the script editor to add spaces between words within operator names. For example, turning on this option results in New Window instead of NewWindow. This is on by default.

Setting colors used in the editor
To set your own color for any of the listed types (see Figure 2.6), click the color box and select your color.

Figure 2.6 Color selection

Scripting the Script Editor
The editor is also scriptable, meaning you can write a script to write, change, or get information from another script. First, you need to make a reference to your script window. For example, here is a new script window created and assigned to the variable `ww`:

```
ww = new window("scripttest", <<script, "initial contents");
```

The initial contents may be left off, so that you create a blank script window. Next, you need to get a reference to the display box portion of the script window:

```
ed = ww[scriptbox(1)];
```

Although there is only one script box object, you still must call it by number.

There are a variety of messages you can send your script box object:

<table>
<thead>
<tr>
<th>Table 2.6 Messages for a Script Box Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ed &lt;&lt; get text()</code></td>
</tr>
<tr>
<td><code>ed &lt;&lt; set text(&quot;string&quot;)</code></td>
</tr>
<tr>
<td><code>ed &lt;&lt; append text(&quot;string&quot;)</code></td>
</tr>
</tbody>
</table>
Table 2.6 Messages for a Script Box Object

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ed &lt;&lt; get line text(2)</code></td>
<td>Places only the text from the designated line in a string.</td>
</tr>
<tr>
<td><code>ed &lt;&lt; set line text(2, &quot;string&quot;)</code></td>
<td>Removes the text currently in the designated line and replaces it with the string argument.</td>
</tr>
<tr>
<td><code>ed &lt;&lt; get line count()</code></td>
<td>Returns the number of lines in the script window as an integer.</td>
</tr>
<tr>
<td><code>ed &lt;&lt; get lines();</code></td>
<td>Returns all the text in the script window as a list of strings, one line per string.</td>
</tr>
<tr>
<td><code>ed &lt;&lt; reformat();</code></td>
<td>Reformats the script.</td>
</tr>
<tr>
<td><code>ed &lt;&lt; run();</code></td>
<td>Runs the entire script in the script window.</td>
</tr>
</tbody>
</table>

**The JSL Debugger**

In the JSL debugger, you can run the entire script, or step through the script one line at the time. To use the debugger, the first line of the JSL script should be

```javascript
/*debug run*/
```

or

```javascript
/*debug step*/
```

Make sure to type it exactly as shown above, in the first line of the script, with no extra blanks (including extra blanks within the comment). All letters must be lower case.
The following tasks are possible in the debugger.

**Run Without Debugger**  Runs the script normally, without debugging commands.

**Step**  Steps through the script one command at a time. A green arrow shows the current command.

**Run**  Runs the script from the current command.

**Add Watch**  Adds a variable to the watch list, located at the bottom of the debugger window. Watched variables display their values as the script runs.

**Remove Watch**  Removes a watched variable.

You can add a break point to the script by clicking in the area to the left of a JSL line (the same area where the green arrow circulates). A red circle appears at that line, and the script pauses when the line is executed. Click the circle to remove the break point.
Help with JSL

There are several places within JMP to get help writing or understanding a JSL script.

JSL Browsers

The Help menu has a sub-menu named Indexes for browsing operators, objects, and display boxes in the scripting language:

Figure 2.8 Finding Information for JSL

<table>
<thead>
<tr>
<th>JSL Operators</th>
<th>Shows information about all general operators and functions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Scripting</td>
<td>Shows scriptable objects and what messages each can interpret.</td>
</tr>
<tr>
<td>DisplayBox Scripting</td>
<td>Shows the elements in results windows and their messages.</td>
</tr>
</tbody>
</table>

An entry in the JSL Functions Index includes the syntax and a brief explanation. Many entries also have example code. The Topic Help button for each entry opens the Help system and shows you the entry for the item in the “JSL Syntax Reference,” p. 437. See Figure 2.9 for the entry for If().
Figure 2.9 JSL Functions Index

Show Commands

Show Commands lists all the scriptable objects and operators in the log window, producing a text report that is equivalent to the Operators browser. The default argument (if none is given) is Builtins:

```
show commands();
```

- **Built-in Commands**
  - Add 7 Internal
  - Subtract 7 Internal
  - Multiply 8 Internal
  - Divide 8 Internal

Other arguments are Scriptables (the platforms and their messages), DisplayBoxes (for display tree objects and messages you can send to them—note that some objects are not scriptable), ScriptableObjects (which objects currently have instances), StatTerms (places all the information in the Statistics Index into a JMP data table), and All for all of the above.

```
show commands(scriptables);
```

--- Class ARIMA
  - Show Points [Boolean] [Default On]
  - Show Confidence Interval [Boolean] [Default On]
  - Save Columns [Action]
  - Columns [Subtable] [Scripting Only]

```
show commands(DisplayBoxes);
```

--- Class LayoutBox
--- Class LayoutAtomBox
show commands(scriptableobjects); // objects that have instances
Preferences
Distribution
Bivariate
...
show commands(all); // all of the above

Showing translations

Use show commands(translations) for a list of all objects' JSL commands in English and the localized language. This command creates a data table that lists commands, the English name, and the localized name. Argument translations are enumerated after their commands in entries beginning with the (localized) word Enumeration. For example, in the French release, submit the following to see translations of English JMP commands:

show commands(translations);

Figure 2.10  Localized Commands (French)

Show Properties

Show Properties lists the messages that can be sent to a scriptable object, such as a data table, an analysis platform, or a display, producing a text report that is equivalent to the Objects browser.
Open("SAMPLE_DATA/Big Class.jmp");
show properties(current data table());
  Tables [Subtable]
  Summary [Action]
  Subset [Action]
  Sort [Action]
  Stack [Action]
  Split [Action]
...

bivariate(y(weight),x(height));
show properties(Bivariate[1]); //the analysis platform
  Show Points [Boolean] [Default On]
  Fit Mean [New Entity]
  Fit Line [New Entity]
...

show properties(report(Bivariate[1])); //the platform's display tree
  Close [Boolean]
  Horizontal [Boolean]
  Open All Below [Action]
  Close All Below [Action]
...
This chapter introduces JMP’s operators and their function equivalents. These are the same operators found in JMP’s formula calculator. In fact, the calculator window is driven behind the scenes by JSL. This section considers the calculator’s operators from a scripting perspective.

For the precedence order of all operators, see Table 2.4 “Operators and function() equivalents, in precedence order,” p. 38.

For additional operators, see the “Advanced Concepts” chapter.

*Confusion alert!* Examples in this chapter occasionally use JSL features that haven’t been previously discussed. Do not worry too much about understanding every detail of an example—just focus on the general idea for the topic immediately under discussion.

If you need more information about some other feature that is being used, you can consult the Help system, find quick overviews in the JSL browsers in the Index pane of the JMP Starter Window, or look it up in the “Index,” p. 651 of this book. You might also consider using Find while browsing the online books in Acrobat Reader.
Contents

Numeric Functions ................................................................. 61
  Arithmetic operators ....................................................... 61
  Assignment operators ...................................................... 61
  Constants ...................................................................... 62
  Additional numeric functions .......................................... 62
  Transcendental functions ................................................ 63
  Trigonometric Functions .................................................. 65
  Random Functions ............................................................. 66
Character Functions .............................................................. 67
Comparison and Logical Operators ......................................... 74
  Comparison operators ....................................................... 74
  Logical operators ............................................................. 76
  Special cases ................................................................. 76
  Missing values ............................................................... 77
  Missing character values .................................................. 78
  Short-circuiting behavior ................................................ 78
Date/Time Operators .............................................................. 79
  Constructing dates ......................................................... 79
  Extracting parts of dates .................................................. 79
  Arithmetic on dates .......................................................... 80
  Converting date/time units .............................................. 80
  Y2K-ready date handling ................................................ 80
  Date/time notation .......................................................... 81
Currency ........................................................................ 85
Inquiry Functions ................................................................ 85
  Functions that communicate with users .............................. 86
    Writing to the log ........................................................... 87
    Send information to the user .......................................... 87
Chapter 3

Numeric Functions

This section lists and explains numeric functions, from basic arithmetic operators to trigonometric and random functions. For more advanced statistical functions, see the chapter “Additional Numeric Operators,” p. 431. Each of the functions in this section is also detailed in the JSL Syntax Reference for “Summary of operators, functions, and messages,” p. 437.

Arithmetic operators

The five essential arithmetic operators are as follows. Note that you can add numbers, matrices, and variables that contain either. Each operator has two forms: an operator (e.g., +) and a functional equivalent (e.g., Add()).

\[ a+b \]
Add\((a, b)\)

Adds \(a\) and \(b\).

\[ a-b \]
Subtract\((a, b)\)

Subtracts \(b\) from \(a\).

\[ a*b \]
Multiply\((a, b)\)

Multiplies \(a\) by \(b\). If \(a\) and \(b\) are matrices, does matrix multiplication.

\[ a/b \]
Divide\((a, b)\)

Divides \(a\) by \(b\). If \(a\) and \(b\) are matrices, does matrix division.

\[ a^b \]
Power\((a, b)\)

Raise \(a\) to exponent power \(b\). If \(b\) is not provided—Power\((x)\)—\(x\) is raised to the power of \(2\).

\[-a \]
Minus\((a)\)

Reverses sign of \(a\).

Assignment operators

JSL also provides operators for in-place arithmetic, or assignment operators. These operations are all done in place, meaning that the result of the operation is assigned to the first argument. The most basic assignment operator is the single = operator (or the equivalent function Assign()). For example, if \(a\) is 3 and you do \(a+=4\), then \(a\) becomes 7.

The first argument to an assignment function must be something capable of being assigned (an L-value). You cannot do something like \(3+=4\), because 3 is just a value and cannot be reassigned. You can, however, do something like \(a+=4\), because \(a\) is a global variable whose value you can set.
JSL Operators

Chapter 3

Numeric Functions

\[ a = b \]

Assign\((a, b)\)

Puts the value of \(b\) into \(a\).

\[ a += b \]

AddTo\((a, b)\)

Add the value of \(b\) into \(a\). Equivalent to \(a = a + b\).

\[ a -= b \]

SubtractTo\((a, b)\)

Subtract \(b\) from \(a\), and assign to \(a\). Equivalent to \(a = a - b\).

\[ a *= b \]

MultiplyTo\((a, b)\)

Multiply \(b\) with \(a\), and assign to \(a\). Equivalent to \(a = a \times b\).

\[ a /= b \]

DivideTo\((a, b)\)

Divide \(b\) into \(a\), and assign to \(a\). Equivalent to \(a = a / b\).

\[ a++ \]

Post Increment\((a)\)

Adds one (1) to \(a\), and assign to \(a\). Equivalent to \(a = a + 1\). Most often used as a loop control.

\[ a-- \]

Post Decrement\((a)\)

Subtracts one (1) from \(a\), and assign to \(a\). Equivalent to \(a = a - 1\). Most often used as a loop control.

Constants

JMP provides functions for two useful constants.

**Note:** These functions do not take an argument, but the parentheses are required.

\[ e() \]

Returns the constant \(e\), approximately 2.7182818284590451.

\[ \pi() \]

Returns the constant \(\pi\), approximately 3.1415926535897931.

Additional numeric functions

\[ \text{Abs}(n) \]

Returns a positive number of the same magnitude as the value of \(n\).

\[ \text{Ceiling}(n) \]

Returns the smallest integer greater than or equal to its argument \(n\).
Floor(n)
Returns the largest integer less than or equal to its argument. For example, Floor(2.7) results in 2, while Floor(-.5) results in -1.

Modulo(number, divisor)
Mod(number, divisor)
Returns the remainder when number is divided by divisor. For example, Modulo(6,5) results in 1. Mod is a synonym.

Round(n, places)
Rounds n to the number of decimal places given by the second argument places. For example, Round(3.554, 2) rounds to 3.55 and Round(3.555, 2) rounds to 3.56.

Transcendental functions

Sqrt(n)
Returns the square root of n.

Root(n, r)
Returns the rth root of n, where r defaults to 2 for square root.

Log(n)
Log(n, base)
Returns the natural logarithm (base e logarithm) of n. An optional second argument lets you specify a different base, e.g. Log(n, 3) for the base 3 logarithm of n. The Log argument can be any numeric expression. The expression Log(e()) evaluates as 1, and Log(32,2) is 5. Ln is a synonym.

Log10(n)
Returns the common (base 10) logarithm of n. The argument n can be any numeric expression.

Log1P(n)
Returns the same result as Log(1 + n), except that it is more accurate when n is very small.

Logit(p)
Returns log(p/(1-p)).

Exp(n)
Returns the constant e exponentiated to the power you specify, e.g. Exp(1) is e.

Factorial(n)
Returns product of all numbers 1 through n. For example, Factorial(5), often denoted 5!, evaluates as 120.

NChooseK(n, k)
This function returns the number of n things taken k at a time (“n choose k”) and is computed in the standard way using factorials, as n!/(k!(n-k)!). For example, NChooseK(5,2) evaluates as 10.

Note: This is implemented internally in JMP using lGamma functions. The result is not always an integer.
Gamma(t)
Returns the gamma function for t, or for each row in t if t is a column. Gamma is defined by:
\[ \Gamma(t) = \int_0^\infty x^{t-1} e^{-x} \, dx \]
Solving this integral leads to:
\[ \Gamma(t+1) = t \Gamma(t) = t! \] (t factorial)
and an interesting relationship is:
\[ \Gamma(0.5) = \sqrt{\pi} \].

LGamma(t)
Returns the log gamma function for t, which is the natural log of gamma. You get the same result using the natural log function with the Gamma function; however, the Log Gamma function computes more efficiently and accurately than the pair.

IGamma(t, shape)
Returns the incomplete gamma function for t, which is the same as Gamma except the integral is incomplete and has a second shape parameter called a in the definition:
\[ \text{igamma}(t, a) = \frac{1}{\Gamma(a)} \int_0^\infty x^{a-1} e^{-x} \, dx \]
The formula IGamma(Row(), 1) uses the default shape parameter 1 and computes the incomplete gamma for each row using the row number as the limit of the integral.

Digamma(n)
Returns the digamma function evaluated at n.

Trigamma(n)
Returns the trigamma function evaluated at n. The trigamma function is the derivative of the digamma function.

Beta(m, n)
Returns the beta function which is written in terms of the Gamma function as
\[ B(m, n) = \frac{\Gamma(m)\Gamma(n)}{\Gamma(m+n)} \]

Arrhenius(n)
Converts the temperature n to activation energy. Returns 11605/(n+273.15).

Arrhenius Inv(n)
The inverse of the Arrhenius function. Converts the activation energy n to temperature in Celsius. Returns 11605/(n-273.15).

Scheffe Cubic(x1, x2)
Returns x1*x2*(x1-x2). This function supports notation for cubic mixture models.

Squash(expr)
An efficient computation of the function 1/(1 + e^{expr}), where expr is any numeric column, variable, or expression. The function is S-shaped and goes from 1 to 0 as its argument goes from minus infinity to infinity.
Squish(expr)
An efficient computation of the function \(1/(1 + e^{-expr})\), where \(expr\) is any numeric column, variable, or expression. The function is S-shaped and goes from 0 to 1 as its argument goes from minus infinity to infinity.

Trigonometric Functions
JMP's trigonometric functions expect all angle arguments in radians.
Sine(expr)
Returns the sine. For example, Sine(0) evaluates as 0. Sin is a synonym.
Cosine(expr)
Returns the cosine. For example, Cosine(0) evaluates as 1. Cos is a synonym.
Tangent(expr)
Returns the tangent of an argument given in radians. The expression Tan(.25) evaluates as 0.255342. Tan is a synonym.
ArcSine(expr)
Returns the inverse sine. For example, ArcSine(1) evaluates as 1.57080. ArSin is a synonym.
ArcCosine(expr)
Returns the inverse cosine. For example, ArcCosine(0) evaluates as 1.57080. ArCos is a synonym.
ArcTangent(expr)
Returns the inverse tangent. For example, ArcTangent(0.5) evaluates as 0.46365. ATan is a synonym.
SinH(expr)
Returns the hyperbolic sine. The expression SinH(1) evaluates as 1.175201.
CosH(expr)
Returns the hyperbolic cosine. CosH(0) evaluates as 1.0.
TanH(expr)
Returns the hyperbolic tangent of its argument. The expression TanH(1) evaluates as 0.761594.
ArcSinH(expr)
Returns the inverse hyperbolic sine. ArcSinH(1) evaluates as 0.881374.
ArcCosH(expr)
Returns the inverse hyperbolic cosine. ArcCosH(10) evaluates as 2.99322.
ArcTanH(expr)
Returns the inverse hyperbolic tangent. The expression ArcTanH(0.5) evaluates as 0.549306.
Random Functions

Random functions generate random numbers. The basic random functions are briefly listed below. For more advanced random functions that generate random numbers from various distributions, see “Random Functions,” p. 455 in the Syntax Reference.

Random Functions

**Col Shuffle( )**
Shuffles the values randomly each time evaluated. Takes no arguments, but parentheses are required.

**Random Shuffle( )**
Returns the matrix \( m \) with the elements shuffled into a random order.

**Random Reset(seed)**
Restarts the random number sequences with \( seed \).

**Random Uniform()**

**Random Uniform(x)**
Generates random numbers uniformly between 0 and 1. This means that any number between 0 and 1 is as likely to be generated as any other. The result is an approximately even distribution. You can shift the distribution and change its range with constants. No argument is expected, but the parentheses are required. For example:

- **Random Uniform(x)** generates numbers between 0 and \( x \).
- **Random Uniform(high, low)** generates numbers between \( low \) and \( high \).

**Random Normal()**
Generates random numbers that approximate a normal distribution with mean 0 and variance 1. The normal distribution is bell shaped and symmetrical.

**Random Integer(n)**

**Random Integer(k, n)**
Returns a random integer from 1 to \( n \) or from \( k \) to \( n \).

**Random Exp()**
Returns a random number distributed exponentially from 0 to infinity. Equivalent to the negative log of **Random Uniform**. This function takes no argument.

**Random Index(n, k)**
Returns a \( k \) by 1 matrix of random integers between 1 and \( n \) with no duplicates.

**Resample Freq()**
See below for a discussion of the **Resample Freq** function.

Note that the below functions have been deprecated:

- **Random Seed**
- **Random Seeded Normal**
- **Random Seeded Uniform**
More About Resample Freq

The `Resample Freq` function for column formulas generates a random selection with replacement frequency counts, suitable for use in bootstrapping.

- `Resample Freq()` generates 100% resample
- `ResampleFreq(fraction)` generates `fraction*nrow` frequency sample
- `ResampleFreq(n)` generates an `n` frequency sample
- `Resample(fraction or n,FreqColumn)` generates a resample with respect to an existing frequency column

A typical use of this generates a column with many 1’s, some 0’s, some 2’s, etc., corresponding to which rows were randomly assigned any of `n` randomly selected rows.

A typical use of this with an existing frequency column will produce a new frequency column whose values are similar to the old frequency column (have that expected value), but vary somewhat due to random selection at the rates corresponding to the old frequency column.

The following example shows how to calculate the bootstrap distribution of the standard error of the slope parameter in the regression. It assumes the sample data file `Big Class.jmp` is already open.

```jsl
dt = currentDataTable();
resultsTable = newTable("resampleResults.jmp",
    newColumn("StdErr",numeric));

freqCol = dt<<NewColumn("Resample",numeric,formula(ResampleFreq()));
ntrials = 50;
for(i=1,i<ntrials,i++,
    freqCol<<evalFormula; dt<<runFormulas;
    obj = dt<<Bivariate(invisible,Y( :weight), X( :height),
        Freq(:Resample),Fit Line);
    se1 = (obj<<report)["Parameter Estimates",ColumnBox("Std Error")][2];
    obj<<CloseWindow;
    resultsTable<<AddRow({StdErr=se1});
    wait(0);
);

dt<<DeleteColumns({"Resample"});
resultsTable<<Distribution(Y(:StdErr));
```

**Character Functions**

Most character functions take character arguments and return character strings, although some take numeric arguments or return numeric data. Arguments that are literal character strings must be enclosed in quotation marks.

The basic character functions are briefly described below. Further details of some of these functions appear after the table.
Other related functions are discussed in “Hexadecimal and BLOB Functions,” p. 429 in the “ADVANCED CONCEPTS” chapter. For more information on using patterns and regular expressions, see “Pattern Matching and Regular Expressions,” p. 412 in the “ADVANCED CONCEPTS” chapter.

Character Functions

Char(Expr(expr))

Char(name)

Char(number, width, decimal)

Converts an expression or numeric value into a character string. The expression must be quoted with Expr(); otherwise its evaluation is converted to a string.

Width and decimal are optional arguments for formatting numbers; the default is 18 for width and 99 for decimal. The formatted value is fit to the specified width, with leading blanks trimmed so that the number may be less than the specified width.

A special code of -1 (the default) can be used to represent whatever width is needed to represent the number. Since floating point numbers are represented with roughly 17-digit precision, the default width does not exceed 24 characters, allowing positions for sign, decimal point, and scientific “E” suffix.

Num("string")

Produces a numeric value that corresponds to its character string argument when the character string consists of numbers only. If a character string contains a non-numeric value, the result is a missing value. For example, Num("1.123") evaluates as the number 1.123.

Uppercase("text")

Converts any lower case character found in its argument to the equivalent uppercase character.

Lowercase("text")

Converts any upper case character found in its argument to the equivalent lowercase character.

a||b

Concat(a, b)

Returns the second string appended to the first. For example, "Dr." || " " || name produces a new string consisting of the title “Dr.” followed by a space and the contents of the name string.

See “Concat,” p. 71 for more details.

Set Clipboard(arg)

Evaluates the argument, looking for a character result, which is then placed on the clipboard.

Substr("text", start, length)

Extracts the characters that are the portion of the first argument beginning at the position given by the second argument and ending based on the number of characters specified in the third argument. The first argument can be a character column or value, or an expression evaluating to same. The starting argument and the length argument can be numbers or expressions that evaluate to numbers.

For example, to show the first name only,

Substr("Katie Layman", 1, 5);

starts at position 1, reads through position 5, and ignores the remaining characters, which yields
"Katie."

Missing start values, or start values past the end of the string, result in an empty string.

Trim("text")
Produces a new character string from its argument, removing any trailing blanks.

Word(n, "text")
Word(n, "text", "delimiters")
Extracts the $n^{th}$ word from a character string, according to the delimiters given as the optional third argument. The default delimiter is space; if you include a third argument, any and all characters in that argument are taken to be delimiters.

Word(2, "Katie Layman"); //returns "Layman."

Word is the same as Item except that Item treats each delimiter character as a separate delimiter, and Word treats several adjacent delimiters as a single delimiter.

Item(4, "the quick brown fox");
"brown"

Word(4, "the quick brown fox");
"fox"

Words("text", "delimiters")
Extracts the words from text according to the delimiters listed in the optional second argument. The default delimiter is space; if you include a second argument, any and all characters in that argument are taken to be delimiters.

Words("the quick brown fox");
{"the", "quick", "brown", "fox"}

Words("Doe, Jane P.", ", .");
{"Doe", "Jane", "P"}

Item(n, "text")
Item(n, "text", "delimiters")
Extracts the $n^{th}$ item from a character string, according to the delimiters given as the optional third argument. The default delimiter is space; if you include a third argument, any and all characters in that argument are taken to be delimiters.

Item is the same as Word except that Item treats each delimiter character as a separate delimiter, and Word treats several adjacent delimiters as a single delimiter.

Item(4, "the quick brown fox");
"brown"

Word(4, "the quick brown fox");
"fox"

Length("text")
Calculates the length (number of characters) of its argument. For example, Length("Elizabeth") evaluates as 9.
Contains("whole", "part")
Contains(string1, string2)
Contains(list, list item)
Contains(matrix, n)

Returns the numeric position within the first argument of the first instance of the second argument, if it exists. If the second argument is not found within the first argument, a zero is returned.

Left(string, n)
Left(string, n, filler)

Returns a truncated or padded version of the original string. The result contains the left n characters padded with any filler on the right if the length of string is less than n.

Also works for lists.

Right(string, n)
Right(string, n, filler)

Returns a truncated or padded version of the original string. The result contains the right n characters padded with any filler on the left if the length of string is less than n.

Also works for lists.

Starts With(string, substring)

Returns 1 if the if substring appears at the start of the string.

Ends With(string, substring)

Returns 1 if the if substring appears at the end of the string.

Munger("text", offset, find/length)
Munger("text", offset, find, replace)

Computes new character strings from text by inserting or deleting characters. It can also produce substrings, calculate indexes, and perform other tasks depending on how you specify its arguments.

Offset is a numeric expression indicating the starting position to search in the string. If the offset is greater than the position of the first instance of the find argument, the first instance is disregarded. If the offset is greater than the search string's length, Munger uses the string's length as the offset.

See “Munger,” p. 72 for more details.

Repeat(source, a)
Repeat(matrix, a, b)

Return a copy of source concatenated a times, or return a matrix composed of a row repeats and b column repeats.

See “Repeat,” p. 72 for more details.

Sequence(from, to, stepsize, repeatTimes)

Produces an arithmetic sequence of numbers across the rows in a data table, where the from and to are specified. stepsize and repeatTimes are optional; they both default to 1.

See “Sequence,” p. 73 for more details.
Character Functions

Concat

In the Concat function, expressions yielding names are treated like character strings, but globals that have the name values will be evaluated. The following example demonstrates that if you have a stored name value, you need to either use Char before storing it in a global, or Name Expr on the global name.

```plaintext
n = { abc };  
c = n[1] || "def";  
show(c);  
//result is "abcdef"

m = expr(mno);  
c = m || "xyz";  
show(c);  
//result is an error message that mno is unresolved

m = expr(mno);  
c = Name Expr(m) || "xyz";  
show(c);  
//result is "mnoxyz"

m = char(expr(mno));  
c = m || "xyz";  
show(c);  
//result is "mnoxyz"
```

Concat Items() converts a list of string expressions into a single string, with each item separated by a delimiter. If unspecified, the delimiter is a blank. Its syntax is

```plaintext
resultString = Concat Items ({list of strings}, <"delimiter string">);
```

For example,

```plaintext
a = {"ABC", "DEF", "HIJ"};  
result = Concat Items(a, "/");
```
returns

"ABC/DEF/HIJ"

Alternatively,

```plaintext
result = Concat Items(a);
```
returns

"ABC DEF HIJ"

The Contains function searches character strings (as well as lists and matrices).

Contains (string, item)  
Contains(matrix, item)  
Contains(list, item)

For example,

```plaintext
nameList = {"Katie", "Louise", "Jane", "Jaclyn"};  
r = Contains(nameList, "Katie");
```
returns a 1.

**Munger**

*Munger* works many different ways, depending on what you specify for its arguments:

*Munger*(string, offset, *find* | *length*, <replace>);

<table>
<thead>
<tr>
<th><em>Find, length, and replace arguments</em></th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>If you specify a string as the <em>find</em> and specify no <em>replace</em> string, <em>Munger</em> returns the position (after <em>offset</em>) of the first occurrence <em>find</em> string.</td>
<td><em>Munger</em>(&quot;the quick brown fox&quot;, 1, &quot;quick&quot;); 5</td>
</tr>
<tr>
<td>If you specify a positive integer as the <em>length</em> and specify no <em>replace</em> string, <em>Munger</em> returns the characters from <em>offset</em> to <em>offset</em> + <em>length</em>.</td>
<td><em>Munger</em>(&quot;the quick brown fox&quot;,1,5); &quot;the q&quot;</td>
</tr>
<tr>
<td>If you specify a string as the <em>find</em> and specify a <em>replace</em> string, <em>Munger</em> replaces the first occurrence after <em>offset</em> of <em>text</em> with <em>replace</em>.</td>
<td><em>Munger</em>(&quot;the quick brown fox&quot;, 1, &quot;quick&quot;, &quot;fast&quot;); &quot;the fast brown fox&quot;</td>
</tr>
<tr>
<td>If you specify a positive integer as the <em>length</em> and specify a <em>replace</em> string, <em>Munger</em> replaces the characters from <em>offset</em> to <em>offset</em> + <em>length</em> with <em>replace</em>.</td>
<td><em>Munger</em>(&quot;the quick brown fox&quot;, 1, 5, &quot;fast&quot;); &quot;fastuick brown fox&quot;</td>
</tr>
<tr>
<td>If you specify a positive integer as the <em>length</em> and <em>offset</em> + <em>length</em> exceeds the length of <em>text</em>, <em>Munger</em> either returns <em>text</em> from <em>offset</em> to the end or replaces that portion of <em>text</em> with the <em>replace</em> string, if it exists.</td>
<td><em>Munger</em>(&quot;the quick brown fox&quot;,5,25); &quot;quick brown fox&quot; <em>Munger</em>(&quot;the quick brown fox&quot;,5,25, &quot;fast&quot;); &quot;the fast&quot;</td>
</tr>
<tr>
<td>If you specify zero as the <em>length</em> and specify no <em>replace</em> string, <em>Munger</em> returns a blank string.</td>
<td><em>Munger</em>(&quot;the quick brown fox&quot;, 1, 0); &quot;&quot;</td>
</tr>
<tr>
<td>If you specify zero as the <em>length</em> and specify a <em>replace</em> string, the string is inserted before the <em>offset</em> position.</td>
<td><em>Munger</em>(&quot;the quick brown fox&quot;, 1, 0, &quot;see &quot;); &quot;see the quick brown fox&quot;</td>
</tr>
<tr>
<td>If you specify a negative integer as the <em>length</em> value and specify no <em>replace</em> string, <em>Munger</em> returns all characters from the offset to the end of the string.</td>
<td><em>Munger</em>(&quot;the quick brown fox&quot;, 5, -5); &quot;quick brown fox&quot;</td>
</tr>
<tr>
<td>If you specify a negative integer for <em>length</em> and specify a <em>replace</em> string, <em>Munger</em> replaces all characters from the offset to the end with the <em>replace</em> string.</td>
<td><em>Munger</em>(&quot;the quick brown fox&quot;, 5, -5, &quot;fast&quot;); &quot;the fast&quot;</td>
</tr>
</tbody>
</table>

**Repeat**

The **Repeat** function makes copies of its first argument into a result. The second (and sometimes a third) argument is the number of repeats, where 1 means a single copy.

If the first argument evaluates to a character value or list, the result is that many copies.

    repeat("abc",2)
    "abcabc"
repeat("A",2) 
{"A","A"}
repeat([1,2,3],2) 
{1,2,3,1,2,3}

If the first argument evaluates to a number or matrix, the result is a matrix. The second argument is the number of row repeats, and a third argument can specify the number of column repeats. If only two arguments are specified, the number of column repeats is 1.

repeat([1 2 3 4],2,3) 
[ 1 2 1 2 1 2, 
  3 4 3 4 3 4, 
  1 2 1 2 1 2, 
  3 4 3 4 3 4] 
repeat(9,2,3) 
[ 9 ,9 ,9 ,9 ,9 ]

The repeat function is compatible with the function of the same name in the SAS/IML language, but is incompatible with the SAS character data step function, which repeats one more time than this function.

Sequence

Sequence() corresponds to the Sequence function in the Formula Editor and is used to fill the cells in a data table column. It takes four arguments and the last two are optional:

Sequence(from, to, stepsize, repeat)

From and to are not optional. They specify the range of values to place into the cells. If from = 4 and to = 8, the cells will be filled with the values 4, 5, 6, 7, 8, 4, ...

Stepsize is optional. If you do not specify a stepsize, the default value is 1. Stepsize increments the values in the range. If stepsize = 2 with the above from and to values, the cells will be filled with the values 4, 6, 8, 4, 6, ...

Repeat is optional. If you do not specify a Repeat, the default value is 1. Repeat specifies how many times each value is repeated before incrementing to the next value. If repeat = 3 with the above from, to, and stepsize values, the cells will be filled with the values 4, 4, 4, 6, 6, 6, 8, 8, 8, 4, ... If you specify a Repeat value, you must also specify a Stepsize value.

The sequence is always repeated until each cell in the column is filled.

Example:

// Create a new data table
dt = New Table("Sequence Example");

// Add 2 columns and 50 rows
dt << New Column("Count to Five");
dt << New Column("Count to Seventeen by Fours"); dt << Add Rows (50);

/* Fill the first column with the data sequence 1, 2, 3, 4, 5, ... 
Fill the second column with the data sequence 1, 1, 5, 5, 9, 9, 13, 13, 17, 17, ... */

for each row (
column(1)[ ] = Sequence(1,5);
column(2)[ ] = Sequence(1,17, 4, 2);
); 

Since Sequence() is a formula function, you can also set a column’s formula to use Sequence to fill the column. This example creates a new column named “Formula Sequence” and adds a formula to it. The formula is a sequence that fills the column with values between 25 and 29, incremented by 1, and repeated twice (25, 25, 26, 26, 27, 27, 28, 28, 29, 29, 25, ...).

dt << new column("Formula Sequence", formula(Sequence(25, 29, 1, 2)));

Examples of Sequence results:
Sequence(1,5) produces 1,2,3,4,5,1,2,3,4,5,1, ...
Sequence(1,5,1,2) produces 1,1,2,2,3,3,4,4,5,5,1,1, ...
Sequence(10,50,10) produces 10,20,30,40,50,10, ...
10*Sequence(1,5,1) also produces 10,20,30,40,50,10, ...
Sequence(1,6,2) produces: 1,3,5,1,3,5, ... The limit is never reached exactly.

Note: If you want a matrix of values, then use the Index function, not Sequence.

Comparison and Logical Operators

The comparison operators (<, <=, >, >=) work for numbers, strings, and matrices. For matrices they produce a matrix of results. If you compare mixed arguments, such as strings with numbers or matrices, the result is a missing value. Comparisons involving lists are not allowed and also return missing values.

The equality operators (== and !=) work for numbers, strings, matrices, and lists. For matrices, they produce a matrix of results; for lists, they produce a single result. If you test equality of mixed results, e.g. strings with numbers or matrices, the result is 0 or unequal.

Range check operators let you check whether something falls between two specified values:

a=1;show(1<=a<3);
b=2;show(2<b<=3);
  1<=a<3:1
  2<b<=3:0

JSL also has a few extra operators for handling missing values in special ways.

Comparison operators

a==b
Equal(a, b)

Returns 1 if all arguments evaluate to the same value, otherwise 0. Missing values propagate.
Case-sensitive for string comparisons.
Comparison and Logical Operators

- **Not Equal** \((a, b)\)
  
  Returns 1 if all arguments do not evaluate to the same value, otherwise 0. Missing values propagate.

- **Less** \((a, b)\); **Less** \((a, b, c)\)
  
  Returns 1 if \(a\) evaluates strictly less than \(b\) (and \(b\) evaluates strictly less than \(c\)), otherwise 0. Missing values propagate.

- **LessEqual** \((a, b)\);
  
  Returns 1 if \(a\) evaluates less than or equal to \(b\) (and \(b\) evaluates less than or equal to \(c\)), otherwise 0. Missing values propagate.

- **Greater** \((a, b)\);
  
  Returns 1 if \(a\) evaluates strictly greater than \(b\) (and \(b\) evaluates strictly greater than \(c\)), otherwise 0. Missing values propagate.

- **GreaterEqual** \((a, b)\);
  
  Returns 1 if \(a\) evaluates greater than or equal to \(b\) (and \(b\) evaluates greater than or equal to \(c\)), otherwise 0. Missing values propagate.

- **Is Missing** \((expr)\)
  
  Returns 1 if the expression yields a missing value and 0 otherwise.

- **Zero or Missing** \((expr)\)
  
  Returns 1 if the expression yields a missing value or zero, and 0 otherwise.

- **LessEqual** \((a, b)\)
  
  Range check, exclusive below, inclusive above. Useful for flagging values that could have been mis-entered.

- **Less** \((a, b)\)
  
  Range check, inclusive below, exclusive above. Useful for flagging values that could have been mis-entered.

Expressions with comparison operators are evaluated all at once, not in sequence

All the comparison operators are *eliding operators*. That means JMP treats arguments joined by comparison operators as one big clause, as opposed to the way most expressions are evaluated one operator at a time. Evaluating as a single clause produces different results than the more usual method of evaluating in pieces. For example, the following two statements are different:
12<a<13;
(12<a)<13;

The first statement checks whether \( a \) is between 12 and 13, because all three arguments and both operators are read and evaluated together. The second statement uses parentheses to regroup the operations explicitly to evaluate from left to right, which would be the normal way to evaluate most expressions. Thus it first checks whether 12 is less than \( a \), returning 1 if true or 0 if false. Then it checks whether the result is less than 13, which of course will always be true.

All the comparison operators are elided when they are used in matched pairs or in the unmatched pairs \(<...\leq...\leq...\leq...<\). What this means is that if you want a comparison statement to be evaluated one comparison operator at a time, you should use parentheses ( ) to control the order of operations explicitly.

**Logical operators**

JSL provides the usual logical operators—And, Or, and Not. These are most often used for compound comparisons, e.g.

```julia
if( sex=="M" & operation=="hysterectomy", print(row() ) );
```

\( a \& b \)

Logical And. Returns 1 if both are true, missing if any are missing, and 0 otherwise.

\( a \mid b \)

Logical Or. Returns 1 if either or both are true, missing if any are missing, and 0 otherwise.

\( !a \)

Logical Not. Maps nonzero values to 0, maps zero values to 1, and leaves missing values missing.

**Special cases**

Here are some examples to show what you should expect for comparisons involving missing values, mismatched types, and matrices.

**Table 3.2 Some special-case comparison tests**

<table>
<thead>
<tr>
<th>Test</th>
<th>Result</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>m=.; m==1</td>
<td></td>
<td>Equality test with missing returns missing.</td>
</tr>
<tr>
<td>m=.; m!=1</td>
<td></td>
<td>Inequality test with missing returns missing.</td>
</tr>
<tr>
<td>m=.; m&lt;1; m&gt;1; etc</td>
<td></td>
<td>Any comparison with missing returns missing (unless it couldn't possibly be true, see next).</td>
</tr>
<tr>
<td>m=.; 1&lt;m&lt;0</td>
<td>0</td>
<td>Comparison involving missing value that couldn't possibly be true returns false, because false takes precedence over missing for comparisons with more than two arguments (as with logical operators).</td>
</tr>
</tbody>
</table>
Comparison and Logical Operators

Missing values

Missing values propagate missing values for most comparisons and logical operations. Two exceptions: if one value is true and another missing, Or returns true; if one value is false and the other missing, And returns false.

Here’s an easy way to remember how missing values work: Suppose you have survey data and some respondents didn’t specify gender, male or female, so you have missing values. Now you compare two missing values. Are they the same gender? They might be, but you don’t know for sure, so the result is missing. Are they not the same gender? Again, you don’t know, so the result is missing. Likewise, you cannot know whether some value (including a missing value) is greater than or equal to a missing value, so the result is missing.

What about when you compare a missing value with a known value? Sometimes it is possible to determine the result. For example, False AND Missing evaluates to False, because for an AND test to be true, both things must be true. Since one thing is False, it doesn’t matter what the other is—the result will be false. Similarly True OR Missing evaluates to True, because only one thing in an OR test needs to be true for the result to be true.

### Table 3.2  Some special-case comparison tests (Continued)

<table>
<thead>
<tr>
<th>Test</th>
<th>Result</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{a, b}</code> == <code>{a, b}</code></td>
<td>1</td>
<td>Equality test of list arguments returns a single result.</td>
</tr>
<tr>
<td><code>{a, b}</code> &lt; <code>{a, c}</code></td>
<td>.</td>
<td>Comparison test of list arguments is not allowed.</td>
</tr>
<tr>
<td><code>1 == &quot;abc&quot;</code></td>
<td>0</td>
<td>Equality test with mixed types returns false.</td>
</tr>
<tr>
<td><code>1 &lt;= &quot;abc&quot;</code></td>
<td>.</td>
<td>Comparison with mixed types returns missing.</td>
</tr>
<tr>
<td><code>IsMissing(m)</code></td>
<td>1</td>
<td>Returns 1 if missing, 0 otherwise.</td>
</tr>
<tr>
<td><code>{1 2 3} == [2 2 5]</code></td>
<td>[0 1 0]</td>
<td>Equality test of matrices returns matrix of elementwise results. When a matrix is compared to a matrix, comparison is done element-by-element and returns a matrix of 1s and 0s.</td>
</tr>
<tr>
<td><code>{1 2 3} == 2</code></td>
<td>[0 1 0]</td>
<td>Equality test of matrix and a matrix filled with 2s. If a matrix is compared to a number, the number is treated as a matrix filled with that number.</td>
</tr>
<tr>
<td><code>{1 2 3} &lt; [2 2 5]</code></td>
<td>[1 0 1]</td>
<td>Comparison of matrices returns matrix of elementwise results.</td>
</tr>
<tr>
<td><code>{1 2 3} &lt; 2</code></td>
<td>[1 0 0]</td>
<td>Comparison of matrix and a matrix filled with 2s.</td>
</tr>
<tr>
<td>Any([2 2] == {1 2})</td>
<td>0</td>
<td>All summarizes elementwise comparisons; it returns 1 only if all comparisons are true and returns 0 otherwise.</td>
</tr>
<tr>
<td>Any([2 2] == [1 2])</td>
<td>1</td>
<td>Any summarizes elementwise comparisons; it returns 1 if any comparison is true and returns 0 otherwise.</td>
</tr>
</tbody>
</table>

Confusion alert! Logical And and Or now handle missing values differently than in JMP version 3.x (as do If and Match). Previously missing values returned false results; now they propagate missing values, as discussed here.

For the v. 3 behavior, you can use AndMZ and OrMZ (and IfMZ and MatchMZ). These -MZ operators are used automatically when converting formulas from v. 3 data tables.
JSL Operators
Comparison and Logical Operators

If you prefer to see truth tables, submit this script:

```julia
T= [0 0 0 1 1 . . .];
U= [0 1 . 0 1 . 0 1 .];
```

- `T == U`: [1, 0, ., 0, 1, ., ., ., .]
- `T != U`: [0, 1, ., 1, 0, ., ., ., .]
- `T < U`: [0, 1, ., 0, 0, ., ., ., .]
- `T <= U`: [1, 1, ., 0, 1, ., ., ., .]
- `T > U`: [0, 0, ., 1, 0, ., ., ., .]
- `T >= U`: [1, 0, ., 1, 1, ., ., ., .]
- `-T`: [0, 0, 0, -1, -1, -1, ., ., .]
- `!T`: [1, 1, 1, 0, 0, 0, ., ., .]
- `T & U`: [0, 0, 0, 0, 1, ., 0, ., .]
- `T | U`: [0, 1, ., 1, 1, 1, ., 1, .]
- `T + U`: [0, 1, ., 1, 2, ., ., ., .]

You cannot make comparisons directly with explicit missing values—only with variables, matrices, or other things, which could contain missing values. For direct comparisons, use `Is Missing` or `Zero Or Missing`:

- `a==.; a<=.;` // produce errors - you should use `IsMissing(a)`
- `IsMissing(a);` // returns 1 when a is missing
- `b=.;a=b;` // does the comparisons
- `ZeroOrMissing(a);` // returns 1 when a is missing or a is zero

### Missing character values

Only numeric missing values are considered to be missing values. Missing character values are considered to be strings of zero length, not missing values, so you would use a test such as `a==""` rather than `Is Missing(a)`.

### Short-circuiting behavior

JMP’s comparison and logical operators are short-circuiting operators—they stop working as soon as they can return a result. Usually this produces the results you expect, but you should be careful in the following circumstances.

#### When a later argument depends on an earlier argument

For example, in this expression, `List[50]` will not even be examined unless the list actually has at least 50 elements:

```julia
n items(List)>=50 & List[50] == 10;
```

#### When lengthy calculations pose performance concerns

Suppose you have an accurate measurement function which is slow, but an approximate version which is fast. You might consider doing a quick, rough comparison first and then running the lengthy test only when the quick test’s result crosses a certain threshold:

```julia
est(x)<.05 & acc(x)<0.01;
```
When you depend upon later arguments being executed
Be careful about placing assignment statements, for example, in a part of an expression that might never be executed.

Date/Time Operators

Date/time data are handled internally as numbers of seconds since midnight, January 1, 1904.
The expression \texttt{x=01Jan1904} will set \texttt{x} to zero, since the indicated date is the base date or "zero date" in JMP. If you examine the values of dates, they should be appropriately large numbers. For example 5oct98 happens to be 2990390400.

Constructing dates

You can construct dates with the following operators. If it were the stroke of midnight, 1 December 2003, all of the following statements would be equivalent. Each would return 3153081600, and you could pass any of them to a date notation operator such as \texttt{Long Date} to get a recognizable format:

\begin{verbatim}
Date DMY(1,12,2003);
Date MDY(12,1,2003);
Today();
3153081600
long date(today());
"Monday, December 1, 2003"
\end{verbatim}

These operators are handy if you are setting up a date column to label rows, or any other case where you can describe a series of date values generally:

\begin{verbatim}
new column("Entry date", format("m/d/y"),formula(Date DMY(row(),3,2001))); // be sure the format you pick is available on your machine
\end{verbatim}

Extracting parts of dates

You can extract parts of date values using the operators \texttt{Month}, \texttt{Day}, \texttt{Year}, \texttt{Day Of Week}, \texttt{Day Of Year}, \texttt{Week Of Year}, and \texttt{Time Of Day}, which all return integers. For example:

\begin{verbatim}
day of year(today());
335
\end{verbatim}
Arithmetic on dates

You can perform the usual arithmetic operations with date/time data as with any other numeric data. For example, you could find the days elapsed between entries:

```julia
new column("Days elapsed",
    formula(
        if(row()==1,., //to avoid error for row 1
        //else for rows after 1:
        (:Entry date[row()-1]-:Entry date[row()])/in days())));
```

Converting date/time units

The `In Minutes`, `In Hours`, `In Days`, `In Weeks`, and `In Years` operators are useful for re-expressing a time interval in units other than seconds. For example, this returns the number of fortnights between now and the next Leap Day:

```julia
(Date DMY(29,2,2004)-Today())/InWeeks(2);
```

Usually `n` is 1, but you could use `In Years(10)`, for example, to re-express an interval in decades.

Y2K-ready date handling

JMP uses its own Y2K-ready algorithms for interpreting and displaying date/time strings rather than supporting operating system-specific date/time formats. JMP respects the date/time separator-characters chosen in the Regional Settings control panel (Windows) or the Date&Time control panel (Macintosh) for interpreting and displaying dates.

JMP always displays four-digit years regardless of Regional Settings. If for some reason it should be necessary to show two-digit years, use JMP's operators for manipulating character strings.

Two-digit years are interpreted according to the current system clock year, according to the rules below. To avoid ambiguity or override the default interpretation, type a four-digit year value.

---

Confusion alert! Because Sunday is the first day of the week, the first Sunday of each year always starts week 2. Days before the first Sunday (if there are any) will be in week 1. This means that in a year in which January 1 is on Sunday, the first week is counted as the second week.

Example:

```
Week of Year(Date DMY(1,1,2006));
1
```

In years in which Sunday is the first day of the year, subtract 1 from `Week Of Year()` to obtain the correct week number.
Date/Time Operators

Several operators convert numbers directly to common formats.

- short date(2998080000);
  “01/02/1999”
- long date(2998080000);
  “Saturday, January 2, 1999”
- abbrev date(2998080000);
  “Jan 2, 1999”
- mdyhms(2998080000);
  “01/02/1999 12:00:00 AM”

You can also use the Format and InFormat operators for displaying and entering date/time values in one of many notation formats.

- Format(3096558900,":"day:hr:m");
  “:35839:19:15”
- InFormat(“02Jan1999 13:01:55","ddMonyyyy h:m:s”);
  02Jan1999:13:01:55

Dates that are the results of expressions that are to be displayed in a text window (like the log window) are given as a date attribute using As Date. For example,

```julia
x = As Date(8Dec2000 + inDays(2));
```

shows as

```
10Dec2000
```

The formats JMP currently supports are shown in Table 3.4. You can also use them for the format argument to a Format message to a data column, as discussed in “Display formats,” p. 149 in the “Data Tables” chapter. Note that the date-separator character on your machine might differ from the / character shown here.

You may enter time values in 24-hour format (military time) or with AM/PM designators.
### Table 3.4 Date/time formats supported in JMP

<table>
<thead>
<tr>
<th>Type</th>
<th>Format string</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Date only</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Per day</td>
<td>&quot;m/d/y&quot;</td>
<td>&quot;01/02/1999&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;mmddyyyy&quot;</td>
<td>&quot;01021999&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;m/y&quot;</td>
<td>&quot;01/1999&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;d/m/y&quot;</td>
<td>&quot;02/01/1999&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;ddmmyyyy&quot;</td>
<td>&quot;02011999&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;ddMonyyyy&quot;</td>
<td>&quot;02Jan1999&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;Monddyyyy&quot;</td>
<td>&quot;Jan021999&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;y/m/d&quot;</td>
<td>&quot;1999/01/02&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;yyyymmdd&quot;</td>
<td>&quot;19990102&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;yyyy-mm-dd&quot;</td>
<td>&quot;1999-01-02&quot;</td>
</tr>
<tr>
<td><strong>Date and time</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Per day</td>
<td>&quot;m/d/y h:m&quot;</td>
<td>&quot;01/02/1999 13:01&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;01/02/1999 1:01 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;m/d/y h:m:s&quot;</td>
<td>&quot;01/02/1999 13:01:55&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;01/02/1999 1:01:55 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;d/m/y h:m&quot;</td>
<td>&quot;02/01/1999 13:01&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;02/01/1999 1:01 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;d/m/y h:m:s&quot;</td>
<td>&quot;02/01/1999 13:01:55&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;02/01/1999 1:01:55 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;y/m/d h:m&quot;</td>
<td>&quot;1999/01/02 13:01&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;1999/01/02 1:01 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;y/m/d h:m:s&quot;</td>
<td>&quot;1999/01/02 13:01:02&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;1999/01/02 1:01:02 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;ddMonyyyy h:m&quot;</td>
<td>&quot;02Jan1999 13:01&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;02Jan1999 1:01 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;ddMonyyyy h:m:s&quot;</td>
<td>&quot;02Jan1999 13:01:02&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;02Jan1999 1:01:02 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;ddMonyyyy:hm&quot;</td>
<td>&quot;02Jan1999:13:01&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;02Jan1999:1:01 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;ddMonyyyy:hm:s&quot;</td>
<td>&quot;02Jan1999:13:01:02&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;02Jan1999:1:01:02 PM&quot;</td>
</tr>
<tr>
<td><strong>Day number and time</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;:day:hr:m&quot;</td>
<td>&quot;34700:13:01&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;33:00:1:01 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;:day:hr:m:s&quot;</td>
<td>&quot;34700:13:01:02&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;33:00:1:01:02 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;hr:m:s&quot;</td>
<td>&quot;13:01:02&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;01:01:02 PM&quot;</td>
</tr>
</tbody>
</table>
Note that the last five date/time formats are not available for date input.

<table>
<thead>
<tr>
<th>Table 3.4 Date/time formats supported in JMP (Continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>Duration</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Long date</td>
</tr>
<tr>
<td>Abbreviated date</td>
</tr>
<tr>
<td>Locale Date</td>
</tr>
<tr>
<td>Locale Date Time</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Note that the last five date/time formats are not available for date input.

<table>
<thead>
<tr>
<th>Table 3.5 Date/time operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operator</strong></td>
</tr>
<tr>
<td>In Minutes(n)</td>
</tr>
<tr>
<td>In Hours(n)</td>
</tr>
<tr>
<td>In Days(n)</td>
</tr>
<tr>
<td>In Weeks(n)</td>
</tr>
<tr>
<td>In Years(n)</td>
</tr>
<tr>
<td>Date DMY(day, month, year)</td>
</tr>
<tr>
<td>Date MDY(month, day, year)</td>
</tr>
<tr>
<td>Today()</td>
</tr>
</tbody>
</table>
JSL Operators
Chapter 3
Date/Time Operators

- **Second**(*datetime*)
  Returns an integer representation for the second part of the date-time value supplied.

- **Minute**(*datetime*)
  Returns an integer representation for the minute part of the date-time value supplied.

- **Hour**(*datetime*)
  Returns an integer representation for the hour part of the date-time value supplied.

- **Day**(*date*)
  Returns an integer representation for the day of the month of the date supplied.

- **Month**(*date*)
  Returns an integer representation for the month of the date supplied.

- **Year**(*date*)
  Returns an integer representation for the year of the date supplied.

- **Day Of Week**(*date*)
  Returns an integer representation for the day of the week of the date supplied. Weeks are Sunday–Saturday.

- **Day Of Year**(*date*)
  Returns an integer representation for the day of the year of the date supplied.

- **Week Of Year**(*date*)
  Returns an integer representation for the week of the year of the date supplied. Weeks are Sunday–Saturday. Note that the first Sunday of the year begins week 2.

- **Time Of Day**(*date*)
  Returns an integer representation for the time of day of the date/time supplied.

- **Short Date**(*date*)
  Returns a string representation for the date supplied, in the format mm/dd/yy, e.g. "02/29/04".

- **Long Date**(*date*)
  Returns a string representation for the date supplied, formatted like "Sunday, February 29, 2004".

- **Abbrev Date**(*date*)
  Returns a string representation for the date supplied, formatted like "Feb 29, 2004".

- **MDYHMS**(*date*)
  Returns a string representation for the date supplied, formatted like "2/29/04 00:02:20".

- **Format**(*date*, "format")
  Returns the value in the format you specify in the second argument. Most typically used for formatting date/time values from a number of seconds to a formatted date. Format choices are those shown in the Column Info dialog box; also see Table 3.4 “Date/time formats supported in JMP,” p. 82.

- **In Format**(string, "format")
  Parses a string of a given format and returns date/time value expressed as if surrounded by As Date(), returning the date in ddMonyyyy format.

- **As Date**(*expression*)
  Formats a number or expression so that it shows as a date when streamed out to a text window.

**Table 3.5 Date/time operators (Continued)**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Second</strong>(<em>datetime</em>)</td>
<td>Returns an integer representation for the second part of the date-time value supplied.</td>
</tr>
<tr>
<td><strong>Minute</strong>(<em>datetime</em>)</td>
<td>Returns an integer representation for the minute part of the date-time value supplied.</td>
</tr>
<tr>
<td><strong>Hour</strong>(<em>datetime</em>)</td>
<td>Returns an integer representation for the hour part of the date-time value supplied.</td>
</tr>
<tr>
<td><strong>Day</strong>(<em>date</em>)</td>
<td>Returns an integer representation for the day of the month of the date supplied.</td>
</tr>
<tr>
<td><strong>Month</strong>(<em>date</em>)</td>
<td>Returns an integer representation for the month of the date supplied.</td>
</tr>
<tr>
<td><strong>Year</strong>(<em>date</em>)</td>
<td>Returns an integer representation for the year of the date supplied.</td>
</tr>
<tr>
<td><strong>Day Of Week</strong>(<em>date</em>)</td>
<td>Returns an integer representation for the day of the week of the date supplied. Weeks are Sunday–Saturday.</td>
</tr>
<tr>
<td><strong>Day Of Year</strong>(<em>date</em>)</td>
<td>Returns an integer representation for the day of the year of the date supplied.</td>
</tr>
<tr>
<td><strong>Week Of Year</strong>(<em>date</em>)</td>
<td>Returns an integer representation for the week of the year of the date supplied. Weeks are Sunday–Saturday. Note that the first Sunday of the year begins week 2.</td>
</tr>
<tr>
<td><strong>Time Of Day</strong>(<em>date</em>)</td>
<td>Returns an integer representation for the time of day of the date/time supplied.</td>
</tr>
<tr>
<td><strong>Short Date</strong>(<em>date</em>)</td>
<td>Returns a string representation for the date supplied, in the format mm/dd/yy, e.g. &quot;02/29/04&quot;.</td>
</tr>
<tr>
<td><strong>Long Date</strong>(<em>date</em>)</td>
<td>Returns a string representation for the date supplied, formatted like &quot;Sunday, February 29, 2004&quot;.</td>
</tr>
<tr>
<td><strong>Abbrev Date</strong>(<em>date</em>)</td>
<td>Returns a string representation for the date supplied, formatted like &quot;Feb 29, 2004&quot;.</td>
</tr>
<tr>
<td><strong>MDYHMS</strong>(<em>date</em>)</td>
<td>Returns a string representation for the date supplied, formatted like &quot;2/29/04 00:02:20&quot;.</td>
</tr>
<tr>
<td><strong>Format</strong>(<em>date</em>, &quot;format&quot;)</td>
<td>Returns the value in the format you specify in the second argument. Most typically used for formatting date/time values from a number of seconds to a formatted date. Format choices are those shown in the Column Info dialog box; also see Table 3.4 “Date/time formats supported in JMP,” p. 82.</td>
</tr>
<tr>
<td><strong>In Format</strong>(string, &quot;format&quot;)</td>
<td>Parses a string of a given format and returns date/time value expressed as if surrounded by As Date(), returning the date in ddMonyyyy format.</td>
</tr>
<tr>
<td><strong>As Date</strong>(<em>expression</em>)</td>
<td>Formats a number or expression so that it shows as a date when streamed out to a text window.</td>
</tr>
</tbody>
</table>
JMP supports the use of currency symbols. To set a specific currency, use the `Format()` function. The syntax is:

\[
\text{Format}(x, "format", \langle"currency code\rangle, \langle\text{decimal}\rangle)
\]

where \(x\) is a column, or a number. (For datetime formats, see “Date/Time Operators,” p. 79).

To designate a number or a column as a currency, the format argument must be set to “Currency”. Then add the ISO 4217 code for a specific currency as a quote string. This third argument is invalid unless the format is set to “Currency”. For example:

\[
\text{Format}(12345.6, "Currency", "GBP", 3)
\]

“£12,345.600”

**Inquiry Functions**

How can you tell what kind of data element you have? JSL has inquiry operators to test whether something is or isn’t a specific type of data element.

JSL also has an inquiry operator to test whether a global variable, a data table, or a data column has a value assigned or not: `Is Empty`. You can get errors if you try to refer to something that hasn’t been created or assigned a value yet. Programmers call this an “uninitialized variable.” To check whether a global variable, data table, or data column has a value yet, use `Is Empty` with the name of the item.

```
Is Empty(myGlobal);
Is Empty(dt);
Is Empty(col);
```

In addition, JSL provides a general `Type` function that returns a string naming the type of the resulting value. The possible type names are `Unknown`, `List`, `DisplayBox`, `Picture`, `Column`, `TableVar`, `Table`, `Empty`, `Pattern`, `Date`, `Integer`, `Number`, `String`, `Name`, `Matrix`, `RowState`, `Expression`, `Associative Array`, or `Blob`.

For example,

```
Show(Type(1), Type("hi"), Type(\{"a",2\}));
```

results in

```
Type(1):"Integer"
Type("hi"):"String"
Type(\{"a", 2\}):"List"
```

`Is Scriptable()` is especially useful when parts of a script depend on previous-generated results. Test that the previous results are scriptable and only continue with a script after `IsScriptable()` returns `true`.

**Table 3.6 Inquiry functions to identify object types**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Is Number(x)</code></td>
<td>Returns 1 if the evaluated argument is a number or missing numeric value, or 0 otherwise.</td>
</tr>
</tbody>
</table>
86

JSL Operators
Functions that communicate with users

Chapter 3

Table 3.6 Inquiry functions to identify object types (Continued)
Syntax
Is String(x)

Explanation

Returns 1 if the evaluated argument is a string, or 0 otherwise.
Is Matrix(x)
Returns 1 if the evaluated argument is a matrix, or 0 otherwise.
Is Expr(x)
Returns 1 if the evaluated argument is an expression, or 0 otherwise.
Is Name(x)
Returns 1 if the evaluated argument is a name-only expression, or 0 otherwise.
Is List(x)
Returns 1 if the evaluated argument is a list, or 0 otherwise.
Is Associative Array(x) Returns 1 if the evaluated argument is an associative array, or 0 otherwise.
Is Scriptable(x)
Returns 1 if the evaluated argument is a scriptable object, or 0 otherwise.
Is Empty(global)
Returns 1 if the global variable, data table, or data column does not have a value
(is uninitialized), or 0 otherwise.
Is Empty(dt)
Is Empty(col)
Type(x)

Returns a string naming the type of X.

Version and Host information
Host Is is an inquiry operator. Host Is returns a boolean result, either 1 if true or 0 if false. This is

useful for identifying the current operating system in scripts intended for use on Windows, Linux and
Macintosh platforms. Although now not necessary, Host Is was frequently used to open data tables.
For example:
dt=if(
host is(Windows),
open("..\sample data\Big Class.JMP"),
host is(Linux),
open(“/usr/local/jmp/Sample Data/Big Class.jmp”),
host is(Macintosh),
open("::sample data:big class"));

Another use is to specify different text sizes in reports for different operating systems. If you commonly
write your scripts on Windows and share them with people who use them on Macintosh or Linux, the
results can look different from what you intended. For example, this line sets the text to a larger size on
Macintosh and a smaller size on Windows and Linux:
txtsize = if(host is(Mac),12,10);

The JMP Version() function returns the JMP version as a string. The function puts a leading blank
on the version if it is less than 10 to make it easier to compare versions 7, 8, or 9 with version 10 in the
future. This function is not available in releases prior to 6.0.
JMP Version();
" 8.0"

Functions that communicate with users
Show, Print, and Write put messages in the log window. Speak, Caption, Beep, and StatusMsg
provide ways to say something to a viewer. Mail can send an email alert to a process operator.


JMP's scripting language has methods for constructing dialog boxes to ask for data column choices and other types of information. See “Modal Dialogs,” p. 266 in the “Display Trees” chapter.

**Writing to the log**

**Show**

Show displays the items you specify in the log. Notice that when you show variables, the resulting message is the variable name, a colon :, and its current value.

```julia
X=1;A="Hello, World"
show(X,A,"foo")
```

```
x:1
a:"Hello, World"
"foo"
```

**Print**

Print sends the message you specify to the log. Print is the same as Show except that it prints only the value of each variable without the variable name and colon.

```julia
X=1;A="Hello, World"
print(X,A,"foo")
```

```
1
"Hello, World"
"foo"
```

**Write**

Write sends the message you specify to the log. Write is the same as Print except that it suppresses the quotation marks around the text string, and it doesn’t start on a new line unless you include a return character yourself with the \!N escape sequence.

```julia
write("Here is a message.");
```

```julia
write(myText||" Don't forget to buy milk."); // use || to concatenate
```

```julia
write("\!NAnd bread."); // use \!N for return
```

```julia
Here is a message. Don't forget to buy milk.
And bread.
```

The sequence \!N inserts the line breaking characters that are appropriate for the host environment. For an explanation of the three line breaking escape sequences, see "Quoted Strings," p. 34 in the “JSL Building Blocks” chapter.

**Send information to the user**

**Beep**

Beep causes the user’s computer to make an alert sound.
Speak

Speak reads text aloud. On Macintosh, Speak has one boolean option, Wait, to specify whether JMP should wait for speaking to finish before proceeding with the next step. The default is not to wait, and you need to issue Wait(1) each time. For example, here's a script certain to drive anybody crazy. With Wait(1), you'll probably want to interrupt execution before long. If you change it to Wait(0), the iterations proceed faster than the speaking possibly can and the result sounds strange. On Windows, you can use a Wait(n) command to accomplish the same effect.

for(i=99,i>0,i--,
    speak(wait(1),char(i)||" bottles of beer on the wall, "
    ||char(i)||" bottles of beer; "
    ||"if one of those bottles should happen to fall, "
    ||char(i-1)||" bottles of beer on the wall. ")

A more practical example has JMP announce the time every sixty seconds:

// Time Announcer
script = expr(
    tod = mod(today(),indays(1));
    hr  = floor(tod/inHours(1));
    min = floor(mod(tod,inHours(1))/60);
    timeText = "time, " || char(hr) || " " || char(min);
    text = Long Date(today()) || ", " ||timeText;
    speak(text);
    show(text);
    schedule(60,script);    // seconds before next script
);
script;

You might use a similar technique to have JMP alert an operator that a process has gone out of control.

Note: For Speak to work on a Linux machine, the text-to-speech synthesizer program Festival must be installed and in the user's path. If Festival is not found, the an error is reported to the lost that it was unable to find the speech synthesizer program.

Caption

Caption brings up a small window with a message to the viewer. Captions are a way to annotate demonstrations without adding superfluous objects to results windows. The first argument is an optional \{x,y\} screen location given in pixels from the upper left; the second argument is the text for the window. If the location argument is omitted, windows appear in the upper left corner.

The Spoken option causes captions to be read aloud by the operating system's speech system (if available). Spoken takes a boolean argument, and the current setting (on or off) remains in effect until switched by another Caption statement that includes a Spoken setting.

You can include pauses in the playback by including the named argument Delayed and a time in seconds. Such a setting causes that caption and all subsequent caption windows to be delayed by that number of seconds, until a different Delay setting is issued in a Caption statement. Use Delay(0) to stop delaying altogether.

This script turns speaking on and leaves it on until the last caption.
caption(\{10,30\},"A tour of the JMP Analyses", spoken(1), Delayed(5));
caption("Open a data Table");
    bigClass=open("$SAMPLE_DATA/Big Class.JMP");
caption("A JMP Data Table consists of rows and columns of data");
caption("The rows are numbered and the columns are named");
caption(\{250,50\},"The data itself is in the grid on the right");
caption(\{5,30\},spoken(0),"A panel along the left side shows columns and other attributes");

Each new Caption hides the previous one, i.e. there is only one caption window available at a time. To close a caption without displaying a new one, use the named argument Remove.

caption(remove);

StatusMsg

This command sends a message to the status bar.

    StatusMsg("string")

Mail

(Windows only) Mail sends an email message to alert a user about a condition in JMP. For example, a process control manager might include a test alert script in a control chart to trigger an email warning to her pager:

    mail("JaneDoe@company.com", "out of control", "Process 12A out of control at \|\|Format(today(), "d/m/y h:m:s\")");

Mail can also send an attachment with the email. An optional fourth argument specifies the attachment. The attachment is transferred in binary format after its existence on the disk is verified. For example, to attach the Big Class.jmp data table, submit

    mail("JohnDoe@company.com", "Interesting Data Set", "Have a look at this class data.", "C:\myJMPData\Big Class.jmp");
Sometimes you may need to program repetitive operations, or perhaps implement some statistical test that JMP doesn’t have built in. Then you will need to learn some of the functions that control flow.

The following functions take on the roles that in other programming languages have statements with special syntactic structure. They are just functions in JSL, like any other functions, with arguments separated by commas.

This chapter covers basic programming functions and variables. The chapter “Advanced Concepts,” p. 401, covers more advanced techniques, such as throwing and catching exceptions and working with expressions.
## Contents

Programming Example ......................................................... 93
Gluing expressions together ...................................................... 93
  Glue .................................................................................. 93
  First ................................................................................. 94
  Example .......................................................................... 94
Iterating .................................................................................. 94
  For ..................................................................................... 94
  While ............................................................................... 96
  Summation ....................................................................... 96
  Product ............................................................................ 96
  Break and Continue ............................................................. 96
Conditional functions ............................................................... 97
  If ....................................................................................... 97
  Match ............................................................................. 98
  Choose ........................................................................... 98
  Interpolate ....................................................................... 99
  Step ............................................................................... 100
Controlling script execution ...................................................... 100
  Wait ............................................................................... 101
  Schedule ......................................................................... 101
Variables ................................................................................ 101
  Global Variables .............................................................. 101
  Local variables ............................................................... 102
JSL Data Structures ................................................................. 103
  Lists ............................................................................... 103
  Matrices ......................................................................... 107
  Associative Arrays ............................................................. 107
Chapter 4

Programming Functions

Programming Example

Here is an example script using JSL to implement a statistical method, a type of runs test. Many of the functions used in this script are already familiar from the "JSL Operators" chapter. A flow-control function, Summation, is also used to iterate computations over many rows at a time.

Open("$SAMPLE_DATA/Big Class.jmp");
Summarize(mean=mean(Height),n=count(Height));
nr = nrow();
n1 = summation(i=1, nr, Height[i]>mean);
n2 = n-n1;
r = summation(i=2, nr, (Height[i]>mean)!=(Height[i-1]>mean));
muR = 2*n1*n2/(n1+n2);
sigmaR = sqrt(2*n1*n2*(2*n1*n2-n1-n2)/(n^2*(n-1)));
zTest = (r-muR)/sigmaR;
pValue = (1-normal distribution(abs(zTest)))*2;
show(r, muR, sigmaR, zTest, p Value);

The first statement opens a JMP data table. The Summarize function obtains the mean and count for the column height. NRow gets the number of rows, which will be the same as the count n, since there are no missing values. Then it finds n1, the number of heights greater than the mean; n2, the remaining number; and r, the number of adjacent heights that are not both above or both below the mean. The muR and sigmaR are the mean and standard deviation of the test statistic under the hypothesis that there is no run pattern, zTest is the normal test statistic, and p Value is the significance level of the test. If there is a pattern across time, then neighboring values would be more likely to be alike in whether they were above or below the mean.

Gluing expressions together

Glue

The semicolon operator just evaluates each argument in succession, returning the result of the last one. For example:

i=0; j=2

The semicolon in other languages is used as a statement terminator character. In JSL it is just a binary operator for gluing together expressions. The result returned is the result of the right-most expression. The function version of the semicolon is Glue, which makes the following two statements the same:

Glue(a=2,b=3)
a=2; b=3

You can also add a semicolon at the end of an expression as a terminator character, but it is not required. The semicolon is ignored if nothing follows it. Trailing semicolons are allowed at the end of a script stream, and before a closing parenthesis or closing brace: ) or }. It is useful to get in the habit of using a final semicolon to avoid errors when copying and pasting small scripts into larger ones.
First

The First function works just like Glue() except that it returns the first argument’s result instead of the last’s. For example, this script increments a counter global \( i \) after getting the \( i \)th value of a variable, but enclosing the two steps inside First( ) enables the script to return the datum, not the new value of the counter.

\[
i=1; \text{First}(\text{height}[i], i++);
\]

Example

What are the results of this script?

\[
a=\text{root}(b=9;4);
\]

Semicolon has weak precedence, so when JMP evaluates this statement, it first attempts to evaluate the quantity inside the parentheses. JMP simply executes two separate statements:

\[
b=9;
4;
\]

Thus it assigns 9 to \( b \), then it evaluates the last statement, 4. Several statements glued together return the return value of the last statement, so 4 is returned for the expression inside the parentheses. JMP then takes the square root of 4 and assigns the result (2) to \( a \). Assignment always returns the assigned value, so the ultimate return value for the whole script is 2. So, this script results in two assignments, \( a=2 \) and \( b=9 \), and returns 2.

Example

What are the results of this script?

\[
a=\text{root}(\text{First}(b=9,4));
\]

First executes both its statements but returns the result of the first, not the last. So the statement 4 accomplishes nothing, 9 is assigned to \( b \), and the square root of 9, 3, is assigned to \( a \).

Iterating

JSL provides For, While, Summation, and Product to iterate actions according to the conditions you specify.

To iterate actions over rows of a data table, use For Each Row, which is discussed in the “Data Tables” chapter.

For

The For function first evaluates init expression, but just one time. Then it tests the while expression, breaking if not true. As long as the while expression is true, it keeps executing the body and iter expressions.

\[
\text{For}(\text{init}, \text{while}, \text{iter}, \text{body});
\]
For loops just like it does in the C (and C++) programming language, although the punctuation is different.

Confusion alert! If you know C, watch out for the difference between JSL and C in the use of semicolons and commas. In JSL For is a function where commas separate arguments and semicolons join them; in C for is a special clause where semicolons separate arguments and commas join them.

For example, you can do a Distribution on the first three columns of the current data table:

```javascript
for(i=0, i<3, i++, Distribution(column(i+1)));
```

This sets \( i \) to 0, evaluates \( 0<3 \) as true, launches the Distribution platform for the first column, increments \( i \) to 1, evaluates \( 1<3 \) as true, does Distribution for the second column, increments \( i \) to 2, evaluates \( 2<3 \) as true, does Distribution for the third column, increments \( i \) to 3, evaluates \( 3<3 \) as false, and finally breaks out of the loop.

Of course, it is not hard to write scripts that get into infinite loops. When this happens, you can use the keyboard cancel sequence to get out of the loop (press Escape on Windows, or type \(-\) on Macintosh).

The following expression sums the numbers from 0 to 20:

```javascript
s=0; For(i=0, i<=20, i++, s=s+i);
```

The For loop works like this. It expects four arguments separated with commas ( , ). The first three arguments are rules for how many times to repeat the loop, and the fourth is what to do repeatedly—this is where the action takes place.

1. First it sets an initial condition or state. In this case, it assigns the value 1 to a variable \( i \). The first argument of a for loop is always a starting condition like this one, but the variable and the value could be anything.

2. Next it tests whether the second argument, \( i<=20 \), is true. If it is true, then it does the action script. If false, it ends the for loop and continues with the rest of the script (if there’s more waiting after the last parenthesis ) of the for loop). The second argument is always a condition, like a little If test. Usually the condition involves the same variable initialized in the first argument, but that’s not always the case. Be careful not to create an infinite loop. (If you make a mistake, you can press Esc on Windows or -Period on Macintosh to stop the looping.)

3. The third argument is what to do after completing the actions; typically it increments \( i \) (adds one to its current value). The actions are everything that comes next inside the for’s parentheses. You’ll come back to this in step 5.

4. The fourth argument does the work of the for loop. In this case, it replaces \( s \) with \( s+i \). Only one command statement is expected in the fourth argument. If you want to do several things on each step, glue the statements together with semicolons ( ; ). Semicolons let numerous things stand in place of one thing.

5. Now the third argument kicks in. After the action of the fourth argument is done, the counter variable \( i \) is incremented, and the execution starts over at the second argument, testing again whether \( i<=20 \) is still true.
While

A related function is While, which repeatedly tests the condition and executes its body script as long as the condition is true.

While(condition, body);

For example, here are two different programs to find the least power of 2 that is greater than or equal to x:

```plaintext
x=287;
y=1; while(y<x,y*=2); show(y);
k=0; while(2^k<x, k++); show(2^k);
```

Summation

The Summation function adds up the body results over all i values. For example:

```plaintext
s = Summation(i=0,10,i); // which is 0+1+2+...+9+10
```

This is the same idea as Σ in the calculator except that with Summation you needn't restrict yourself to creating formula variables.

```plaintext
Summation(i=1, NRow(), x^2);
```

This is the JSL equivalent in the formula calculator:

```plaintext
\sum_{i=1}^{NRow} x^2
```

Product

The Product function works just like Summation, except that it multiplies the body results rather than adding them.

```plaintext
p = Product(i=1,5,i); // which is 1*2*3*4*5
```

This is the JSL equivalent of this in the formula calculator:

```plaintext
\prod_{i=1}^{5} i
```

Break and Continue

Two functions give you further control over looping: Break() and Continue(). These two functions work with For and While loops, and also with For Each Row.

Break

Break immediately stops the loop and proceeds to the next command after the loop. This is usually used inside a conditional statement. For example:

```plaintext
For( i = 1, i <= 10, i++,
    If( i == 5, Break();
    Print( "i=" || Char( i ) );
);```
Conditional functions

JSL provides five functions to execute script conditionally: If, Match, Choose, Interpolate, and Step.

If

The If function returns the result statement(s) when condition evaluates as true (nonzero and non-missing). Otherwise it skips ahead and returns the next result when that condition evaluates as true. The final result is the result if none of the preceding conditions are true—it is the “else” result.

If ( condition1, result1, condition2, result2, ...., resultLast )

If at least one condition evaluates to missing, the function keeps on evaluating and only returns missing (rather than evaluating the else clause) if none of the clauses evaluated to true and at least one conditional returned missing. The way to get a clause to always return despite missing conditionals is to add a conditional at the end that is always true. For example,

If ( age<12,"Young", a>12,"Old", 1,"Ageless" );

For example, the following code sets X to Y if Y is less than 20; otherwise, it sets X to 20.

X = If(Y<20,Y,20);

You can use this to recode ranges, for example:

row()=1; Age Group = If (Age<12,"Child",Age<18,"Teen","Adult");
You can put actions and assignments in the result clauses, and you don’t have to return a result. For example the following two are equivalent:

\[
X = \text{If}(y<20, y, 20);
\]

\[
\text{If}(y<20, x=y, x=20);
\]

Similar conditionals are \text{Choose} and \text{Match}.

Be careful to use \texttt{==} for equality tests, not \texttt{=}. An \texttt{If} with an argument like \texttt{name=value} would assign the value rather than test it.

**Match**

The \texttt{Match} function allows you to make a sequence of equality comparisons without needing to rewrite the value to be compared. The result of the first operand is compared to each of the succeeding even operands, except the last, and if they are equal, the value of the operand immediately after is returned. If nothing matches, then the last operand is returned. For example:

\[
\text{SLabel} = \text{Match}(\text{SCode},
1, "Male",
2, "Female",
"unknown")
\]

This would be equivalent to the following \texttt{If} expression:

\[
\text{SLabel} = \text{If}(\text{SCode}==1, "Male",
\text{SCode}==2, "Female",
"unknown")
\]

With more groups, the value of \texttt{Match} becomes more apparent:

\[
dt=\text{open}("\$SAMPLE\_DATA/Big\ Class.JMP");
\quad\text{for each row(aword=}
\quad\text{match(age, 12, "twelve", 13, "thirteen", 14, "fourteen", 15, "fifteen",
16, "sixteen", "seventeen"))};
\quad\text{show(age, aword));}
\]

If \texttt{a} has integer values 1, 2, 3, ..., you can save even more typing by using \texttt{Choose}.

---

**Confusion alert!** If and \texttt{Match} now handle missing values differently than in JMP version 3.x (as do \texttt{And} and \texttt{Or}). Previously missing values returned false (the “else” result for \texttt{Match}); now they return missing values, as discussed under “Missing values,” p. 77 in the “JSL Operators” chapter.

For the v. 3 behavior, you can use \texttt{IfMZ} and \texttt{MatchMZ} (and \texttt{AndMZ} and \texttt{OrMZ}). These \texttt{~MZ} operators are used automatically when converting formulas from v. 3 data tables.

---

**Choose**

The \texttt{Choose} function can be used to shorten certain recoding even more, provided the arguments are to be tested against integers 1, 2, 3, etc. If the first argument evaluates to 1, \texttt{Choose} returns the first replacement value; if it is 2, the second, and so on. If the first argument evaluates to a integer out of
Conditional functions

range (e.g. 7 when only 5 replacement values are listed), Choose returns the last replacement value. For example:

SLabel = Choose(SCode, "Male", "Female", "Unknown");

Thus, if group is a column with values 1, 2, and 3, the following all accomplish the same task, replacing 1s with "low," 2s with "medium," and 3s with "high."

if(group==1, "low", group==2, "medium", group==3, "high");
g2=match(group, 1, "low", 2, "medium", 3, "high");
g3=choose(group, "low", "medium", "high");

Interpolate

Interpolate linearly interpolates the $y$-value corresponding to a given $x$-value between two points \((x_1, y_1)\) and \((x_2, y_2)\).

The points can be specified as a list of points

Interpolate(x, x1, y1, x2, y2, ...)

or as matrices containing the $x$- and $y$-values

Interpolate(x, xmat, ymat)

Suppose we wanted to approximate the exponential function by linear interpolation between integers, 0 to 5. The following would create the two vectors of values:

\[
xgrid = (0::5);
yvalues = \exp(xgrid);
show(xgrid, yvalues);
\]

This produces the output

\[
xgrid: [0 1 2 3 4 5]
yvalues: [1 2.718281828459045 7.38905609893065 20.08553692318767 54.59815003314424 148.4131591025766]
\]

Following is a plot of what the interpolation function would look like (dashed line), compared with what the continuous version (solid line), which is what is being approximated.
Programming Functions

Controlling script execution

Figure 4.1 Interpolate Exp()

The points must be arranged in ascending order. For example, \texttt{Interpolate(2,1,1,3,3)} returns 2. However, \texttt{Interpolate(2,3,3,1,1)} returns a missing value (\texttt{.}).

\textbf{Step}

\texttt{Step} is like \texttt{Interpolate} except that it finds the corresponding \texttt{y} for a given \texttt{x} from a step-function fit rather than a linear fit between points. That is, it returns the \texttt{y}-value corresponding to the greatest \texttt{x}-value that is less than or equal to the \texttt{x} specified as the first argument.

\texttt{Step(x, x1, y1, x2, y2, ...)}

For example:

\begin{verbatim}
for(i=1,i<7,i++,print(step(i, 1,10, 3,30, 4.5,45, 7,70)));
10
10
30
30
45
45
\end{verbatim}

As with \texttt{Interpolate}, the points must be in ascending order.

\textbf{Controlling script execution}

When you run a script, the \texttt{Run Script} command in the \texttt{Edit} menu changes to \texttt{Stop Script}. While a script is executing, you can select this item (or type Esc on Windows or \texttt{⌘}-.Period on Macintosh) to stop it. Many scripts execute more quickly than you can stop them.
Wait

Stopping a script manually stops the entire script completely. In contrast, the `Wait` command allows you to pause script execution for a time and then continue running the script. This approach is sometimes necessary for scripts intended for interactive use or demonstrations.

`Wait` allows you to slow down scripting playback by pausing the specified number of seconds before continuing with the script. For example, this is a 15-second timer:

```
Wait(15);
```

`Wait` also yields to the system, so that system tasks can be run while waiting. This includes updating the contents of the screen. Issuing a wait time of 0—`Wait(0)`—allows JMP to do other pending events, but not wait any longer than needed to do these events. Waiting zero seconds is useful for allowing results to be displayed right away; otherwise JMP waits for the script to end before yielding to the system to update windows.

Schedule

Schedule runs commands or entire scripts at the times you specify. See the “Production Environments” chapter.

Variables

Assigning a variable this way:

```
x=2;
```

makes a global variable. JMP has a single global variable space that all scripts use. You can also create local variables.

Global Variables

Following are a few commands that can help you manage global variables.

Watch

`Watch` lets you keep track of the value in a global variable.

`Watch` draws a window to report the values of global variables you list. Use the named argument All to watch the values for all global variables. The window updates whenever values change while the script is executing, or you can manually change values yourself by simply editing the value shown in the `Watch` window. This is a useful debugging tool.

```
Watch(a, b);
Watch(all);
```
Show Globals and Clear Globals

Show Globals lists all global variables defined, along with their current values. Clear Globals erases the values set for all global variables defined.

**Note:** This is not the same as erasing the global variables themselves. The only way to erase a global variable completely is by quitting JMP.

```plaintext
show globals();
//Globals
  a = "Hello, World";
  b = 0;
  dt = DataTable("Big Class");
  myRoot = Function({x}, If(x>0, Sqrt(x), 0));
  x = 1;
// 5 Globals
clear globals();
show globals();
//Globals
  a = empty();
  b = empty();
  dt = empty();
  myRoot = empty();
  x = empty();
// 5 Globals
```

Locking and Unlocking Global Variables

If you want to lock a global symbol to prevent it from being changed, use Lock Globals.

**Lock Globals** (name1, name2, ...)

To release the lock and enable the global to be changed, use Unlock Globals.

**Unlock Globals** (name1, name2, ...)

The primary use of these two commands is to prevent scripts from interfering with each other. For example, locking prevents a Clear Globals() command from inadvertently clearing a global that is being used by another script.

Another way of preventing two or more scripts from clobbering each other's variables is to use local variables.

Local variables

Local lets you deal with local variables without affecting the name space around you. For example, suppose you want to call something \( x \), but there could already be an \( x \) global in another script that is calling your script. Just put \( x \) inside a Local block.

```plaintext
y=Local({x},
  x = sqrt(b^2-4*a*c);
  {(-b+x)/(2*a), (-b-x)/(2*a)});
```

The general form is:

```plaintext
Local( {list_of_variables_or_assignments}, body );
```
All occurrences of names in the **body** expression that match names listed for the first argument are resolved locally. **Locals** can be nested in **Functions** and other **Locals**.

Another example: in the following expression, \( mn \) and \( mx \) are never seen outside the scope of the **Local** function:

\[
y = \text{local}( \{ mn=\min(x,y), mx=\max(x,y) \}, mx*(mx-1)+mn );
\]

This example manipulates an expression temporarily without changing the permanent contents of the expression:

\[
\text{polynomial}=\text{expr}(a*x^2 + b*x + c);
\]

\[
\text{show(polynomial)};
\]

\[
\text{polynomial}:a*x^2+b*x+c
\]

\[
\text{local}((\text{polynomial}=\text{insert}(\text{expr}(a*x^2 + b*x + c), \text{expr}(d*x^3),1)), \text{show(polynomial)});
\]

\[
\text{polynomial}:d*x^3+a*x^2+b*x+c
\]

\[
\text{show(polynomial)};
\]

\[
\text{polynomial}:a*x^2+b*x+c
\]

### JSL Data Structures

Besides numbers and strings, JSL provides several other data types variables can hold:

- Lists
- Matrices
- Associative Arrays

#### Lists

Lists are containers to store numerous items—potentially items of different type: numbers, text strings, expressions, matrices, and even other lists. A list is just the function **List** with arguments. The shorthand for **List** is to use curly braces. Thus the following two statements are equivalent:

\[
A = \text{List}(1,2,B,\sqrt{3});
A = \{1,2,B,\sqrt{3}\};
\]

When you define or evaluate a list, it just returns a copy of itself. It does not evaluate items inside the list. For instance, this step reveals that the list contains the variable \( B \), not its current value:

\[
B=7;\text{show(A)};
A:\{1,2,B,\sqrt{3}\}
\]

Use **Eval List** if you want a list with all its items evaluated:

\[
C = \text{Eval List(A)};
\]

\[
\{1, 2, 7, 1.732050807568877\}
\]

Lists can also hold assignments—\{a=1, b=2\}—, or function calls—\{a(1), b(2)\}. Remember, statements such as these just return lists; if you want to evaluate the contents, use **Eval List**.
assignList={a=1, b=3, c=5};
  {a=1, b=3, c=5}
show(assignList);
  assignList:{a = 1, b = 3, c = 5}
evallist(assignList);
  {1, 3, 5}
show(a,b,c);
  a:1
  b:3
  c:5
show(assignList);
  assignList:{a = 1, b = 3, c = 5}

h=function({x},x+2); i=function({x},x+3); k=function({x},x+4);
fnList={h(1), i(3), k(5)};
  fnList:{h(1), i(3), k(5)}
show(fnList);
  fnList:{h(1), i(3), k(5)}
evallist(fnList);
  {3, 6, 9}
show(h,i,k);
  h:Function({x},x + 2)
  i:Function({x},x + 3)
  k:Function({x},x + 4)
show(fnList);
  fnList:{h(1), i(3), k(5)}

Assignment

List assignment is powerful.

{a,b,c} = {1,2,3}; //assigns 1 to a, 2 to b, 3 to c
{a,b,c}+=10; //adds 10 to each variable a, b, and c.
{a},{b,c}++; //increments a, b, and c by 1
{a,b,c}--; //decrements a, b, and c
mylist={1, log(2), e()^pi(), height[40]};
  // stores the expressions in a list

Note that any structure of a list is supported, as long as the arguments are conformable and ultimately numeric. One-element lists are treated like scalars for conformability. For example,

a = {{{{1, 2}, {3, 4}}, {5, {6, 7}}}, 8}, {9, 10}};
b = {{{{10, 20}, {30, 40}}, {50, {60, 70}}}, 80}, {90, 100}};
c = a + b; show(c);
d = a + 100; show(d);
e = sqrt(a); show(e);

results in the output

c:{{{{11, 22}, {33, 44}}, {55, {66, 77}}}, 88}, {99, 110}}
d:{{{{101, 102}, {103, 104}}, {105, {106, 107}}}, 108}, {109, 110}}
e:{{{{1, 1.414213562373095}, {1.732050807568877, 2}}, {2.23606797749979, {2.449489742783178, 2.645751311064591}}, 2.82842712474619}, {3, 3.16227766016838}}}
How many items?

To determine the number of items in a list, use the `NItems` function:

\[ N = N \text{Items}(A); \]

Subscripts

Subscripts for lists extract the specified item(s) from a list. Subscripts can in turn be lists. Note that JSL starts counting from 1, so the first element in a list is [1], not [0] as in some other languages.

\[
\begin{align*}
\text{a} &= \{ \text{"bob"}, 4, [1,2,3], \{x,y,z\} \}; \\
\text{show(a[1])}; & \quad // \text{returns} \text{ "bob"} \\
\text{show(a[\{1,3\}])}; & \quad // \text{returns} \ \{ \text{"bob"}, [1,2,3] \} \\
\text{a[2]} &= 5; & \quad // \text{changes} \ 4 \ \text{to} \ 5
\end{align*}
\]

You can also use subscripts to select or change items in a list.

\[
\begin{align*}
\text{c} &= \{1,2,3\}; \\
\text{c}\{1,2\} &= \{4,4\}; & \quad // \text{changes} \ c \ \text{to} \ \{4,4,3\} \\
\text{c}\{1,2\} &= 4; & \quad // \text{equivalent}
\end{align*}
\]

When you have assignment lists or function lists, you can use a quoted name for the subscript:

\[
\begin{align*}
\text{X} &= \{ \text{a}(1), \text{b}(3), \text{c}(5) \}; \\
\text{XX} &= \{ \text{a}=1, \text{b}=3, \text{c}=5 \}; \\
\text{show(X[\"a\"], XX[\"a\"])}; \\
\text{X[\"a\"]}:1 & \quad \text{X[\"a\"]}:1
\end{align*}
\]

You need to put the name in quotation marks, because otherwise JMP would try to evaluate it and use its value, like this:

\[
\begin{align*}
\text{a} &= 2; \text{show(X[a], XX[a])}; \\
\text{X[a]}:b(3) & \quad \text{XX[a]}:b=3
\end{align*}
\]

Multiple left-side subscripts (e.g. `a[i][j] = value` where `a` contains a list of things that are subscriptable) are allowed in certain cases.

1. Each level except the outermost must be a list. So, in the above example, `a` must be a list but `a[i]` can be anything subscriptable.
2. Each subscript except the outermost must be a number. So, in the above example, `i` must be a number, but `j` could be a matrix or list of indices.

The subscripting can be done to any level of nesting. e.g.

\[ a[i][j][k][l][m][n] = 99; \]

Locating items in a list

To find values in a list, use the `Loc` function.

\[
\text{Loc(list, value)}
\]

This returns a matrix of indices to the values in the list that are equal to `value`, where `value` and `list` can be strings or numbers. If `value` is not found, the resulting matrix has zero rows and zero columns. Therefore, `NRow(Loc(list, value))>0` is equivalent to `Contains(list, value)>0`. 

Note: Complete details on matrix manipulation are found in the “Matrices” chapter, including the description of the equivalent Loc command for matrices (which requires only one argument).

For example, given these lists:

```plaintext
nameList = {“Katie”, “Louise”, “Jane”, “Jane”};
numList = {2, 4, 6, 8, 8}
```

Loc(nameList, “Katie”) returns [1]
Loc(nameList, “Erin”) returns [], an empty matrix, since there are no matches.
Loc(nameList, “Jane”) returns [3, 4]
Loc(nameList, 1) returns []. Note that the type mismatch is tolerated.
Loc(numList, 1) returns []
Loc(numList, 6) returns [3]
Loc(numList, 8) returns [4, 5]

Here’s a summary of the list operators:

**Table 4.1** Operators for working with lists

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>List(a, b, c)</td>
<td>Constructs a list from a set of items. An item can be any expression, including other lists. Items must be separated by commas. Text strings should either be enclosed in double quotation marks (“”) or stored in a variable and called as that variable.</td>
</tr>
<tr>
<td>N Items</td>
<td>N Items(list)</td>
<td>Returns the number of elements in the list specified. Can be assigned to a variable.</td>
</tr>
<tr>
<td>Is List</td>
<td>Is List(arg)</td>
<td>Returns true (1) if arg is a classical list, i.e. one that would result from the construction by List(items) or {items}, and returns false (0) otherwise. An empty list is still a list, so IsList({ }) returns true. If miss=., then IsList(miss) returns false, not missing.</td>
</tr>
<tr>
<td>[]</td>
<td>Subscript(list[i])</td>
<td>Subscripts for lists extract the i&lt;sup&gt;th&lt;/sup&gt; item from the list. Subscripts can in turn be lists or matrices.</td>
</tr>
<tr>
<td>=</td>
<td>Assign(list) = (list)</td>
<td>If the target of an assignment operator is a list and the value to be assigned is a list, then it will assign item by item. The ultimate values in the left list must be L-values, i.e. names capable of being assigned values. Note: If you want to test equality of lists, use ===, not =.</td>
</tr>
<tr>
<td>+=</td>
<td>Add To(list) += (list)</td>
<td></td>
</tr>
<tr>
<td>-=</td>
<td>SubtractTo(list) -= (list)</td>
<td></td>
</tr>
<tr>
<td>*=</td>
<td>MultiplyTo(list) *= (list)</td>
<td></td>
</tr>
<tr>
<td>/=</td>
<td>DivideTo(list) /= (list)</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>Post Increment (list) ++</td>
<td></td>
</tr>
<tr>
<td>--</td>
<td>Post Decrement (list) --</td>
<td></td>
</tr>
<tr>
<td>Eval List</td>
<td>Eval List(list)</td>
<td>Returns a list of the evaluated expressions inside list; see “Lists,” p. 101.</td>
</tr>
</tbody>
</table>
Matrices

Matrices can be used to represent values in a data table as well as matrix algebra. See the “Matrices” chapter for details.

Associative Arrays

An associative array associates unique keys with values, which may not be unique. Though associative arrays are generally not ordered, JMP keys are returned in lexicographical order for the purpose of iteration and serialization.

To create an empty associative array, use one of the following:

```javascript
  cary = [ => ];
  cary = Associative Array();
```

The keys and values can be any JSL objects. Items can be added and changed with subscripting:

```javascript
  cary["state"] = "NC";
  cary["population"] = 116234;
  cary["weather"] = "cloudy";
  cary["population"] += 10;
  cary["weather"] = "sunny";
  cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};
```

Note that L-values in an associative array are just like any other L-value.

The default value determines the value of `map[key]` when key is not in the map. Initially, maps have no default value and `map[key]` will cause an error much like `list[-1]` does. If a default value has been set, then that value will be returned. Besides the `Set Default Value` message, a default value can be set in the literal constructor using `=>value` without a key. For example:

```javascript
  counts = ["a" => 10, "b" => 3, => 0]; // default value of 0
  counts["c"] += 1; // ["a" => 10, "b" => 3, "c" => 1, => 0]
```

Associative Array Constructors

```javascript
  map = [ => ]; // empty associative array
  map = [ => 0 ]; // with default value
  map = ["yes" => 0, "no" => 1 ]; // literal associative array
  map = ["yes" => 0, "no" => 1, => 2 ]; // with default value
  map = Associative Array(); // empty associative array
  map = Associative Array(0); // with default value
  map = Associative Array({"yes", 0}, {"no", 1}); // list containing 2-item lists
  map = Associative Array({"yes", 0}, {"no", 1}, 2); // with default value
  map = Associative Array("yes", "no"). {0, 1}; // two lists, keys and values
  map = Associative Array("yes", "no"). {0, 1}, 2); // with default value
  map = Associative Array(:name, :height); // two column refs
  map = Associative Array(:name, :height, .); // with default value
```
set = Associative Array({"yes", "no"}); // one list creates a set with
default value 0
set = Associative Array(:name); // one column ref creates a set with
default value 0

Working with associative arrays

To determine the number of keys that an associative array contains, use N Items. Using the cary associative array from earlier:

N Items(cary);
4

There are several functions that can be used for associative arrays in addition to lists, strings, and expressions.

Adding and deleting keys and values

Insert, Insert Into, Remove, Remove From, Reverse, and Reverse Into can all be used to add key-value pairs or remove them from the associative array entirely. Insert and Remove return copies of the given associative array with the key-value pair added or removed. Insert Into and Remove From add or remove the key-value pairs directly from the given associative array.

Insert and Insert Into take three arguments: the associative array, a key, and a value.

newcary = Insert(cary, "time zone", "Eastern");
show(cary, newcary);
cary:Associative Array({
  "high schools": {"Cary", "Green Hope", "Panther Creek"},
  "population", 116244,
  "state", "NC"},
  "weather", "sunny"}
})
newcary:Associative Array({
  "high schools": {"Cary", "Green Hope", "Panther Creek"},
  "population", 116244,
  "state", "NC"},
  "time zone", "Eastern"},
  "weather", "sunny"}
})

Insert Into(cary, "county", "Wake");
show(cary);
cary:Associative Array({
  "county", "Wake"},
  "high schools", {"Cary", "Green Hope", "Panther Creek"},
  "population", 116244,
  "state", "NC"},
  "weather", "sunny"}
})

Remove and Remove From take two arguments: the associative array, and a key.

newcary = Remove(cary, "high schools")
show(cary, newcary);
cary:Associative Array({
  "county", "Wake"},}
newcary:
{
  "county" => "Wake",
  "population" => 116244,
  "state" => "NC",
  "weather" => "sunny"
}
)

Remove From(cary, "weather");
show(cary);

cary:Associative Array(
  {
    "county", "Wake",
    "high schools", {
      "Cary", "Green Hope", "Panther Creek"
    },
    "population", 116244,
    "state", "NC"
  }
)

Insert Into and Remove From also have message equivalents. These two statements are equivalent.

cary << Insert("county", "Wake");
Insert Into(cary, "county", "Wake");

Likewise, these two statements are equivalent:

cary << Remove("weather")
Remove From(cary, "weather")

Another message lets you set the default value:

cary << Set Default Value("Cary, NC")

Use <<Get Default Value to determine if there is a default value set for an associative array, and if there is, what its value is. If there is no default value set, Empty() is returned.

Obtaining information about an associative array’s contents

Contains can be used to determine if a certain key is contained inside an associative array:

Contains(cary, "high schools")
1
Contains(cary, "lakes")
0

Several messages provide information about the associative array’s contents.

Use << Get Keys to obtain a list of all keys contained in an associative array.

cary << Get Keys;
  {
    "county", "high schools", "population", "state"
  }

Likewise, <<Get Values returns a list of all values:

cary << Get Values;
  {
    "Wake", {"Cary", "Green Hope", "Panther Creek"}, 116244, "NC"
  }

If you want only the values for certain keys, you can specify them as arguments:

cary << Get Values({"state", "county"});
  {
    "NC", "Wake"
  }

Note that the keys must be given as a list.

Similar to <<Get Values is <<Get Value, which returns the value for a single key:
cary <<Get Value("population");
  116244

Note that only one key can be specified, and it must not be in a list.

To obtain a list of all key-value pairs in an associative array, use <<Get Contents:

cary <<Get Contents;
  {"county": "Wake"},
 {"high schools": {"Cary", "Green Hope", "Panther Creek"}},
  {"population": 116244},
  {"state": "NC"}

Iterating through an associative array

To iterate through an associative array, use <<First and <<Next. The following removes all key-value pairs from the associative array cary, leaving an empty associative array.

currentkey = cary <<First;
total = N Items(cary);
for (i = 1, i <= total, i++,
    nextkey = cary<<Next(currentkey);
    Remove From (cary, currentkey);
    currentkey = nextkey;
);

Associative arrays

Sets

A set is an application of an associative array in which all the keys have a value of 1 and non-keys have an implicit value of 0.

To facilitate set usage, 1 is the default value for inserted keys if no value is provided:

map << insert("sample");
map["sample"]; 1

If you construct an associative array from a list, then the default value is set to 0 for you. Otherwise, you need to set the default value to 0 explicitly.

Graph Theory

Associative arrays can be used for graph theory data structures, such as the following directed graph example:

g = Associative Array();
g[1] = Associative Array({1, 2, 4});
g[2] = Associative Array({1, 3});
g[3] = Associative Array({4, 5});
g[4] = Associative Array({4, 5});
g[5] = Associative Array({1, 2});

The following depth-first search JSL function may be used to traverse this map, or any other directed graph expressed as a map:

dfs = Function( {ref, node, visited},
    Local( {chnode, tmp},
    ... )
)
Write( "Node: " || Char( node ) || ", " || Char( ref[node] << get contents ) || "\\N" );
visited[node] = 1;
tmp = ref[node];
chnode = tmp << first;
While( !Is Missing( chnode ),
    If( !visited[chnode],
        visited = Recurse( ref, chnode, visited )
    );
    chnode = tmp << next( chnode );
    visited
);

The first parameter is a reference to the map, the second is the node you want to use as the starting point, and the third is simply a vector that the function uses to keep track of nodes visited. To see how this function works try the following:

dfs( g, 2, J( N Items( g << get keys ), 1, 0 ) );
JMP has many kinds of objects that can respond to messages. This chapter shows how to create and manage data objects: data tables, data columns, and rows. JSL's messages for working with data correspond to the items in the Tables, Rows, and Cols menus in the graphical user interface, and in turn to items in the various panes of a data table window.

This chapter starts with an introduction to data objects and how to send messages to them and continues with the JSL for working with each of the objects.
## Contents

Data table basics ................................................................. 115
  Getting a data table object ............................................. 115
  Objects and messages ..................................................... 118
Messages for data tables ..................................................... 122
  Naming, saving, and printing .......................................... 122
Journal and Layout ............................................................ 123
Creating and deleting columns ............................................ 124
Accessing values ............................................................... 125
Summarize ........................................................................ 129
Working with metadata ....................................................... 131
Commands from the Rows menu .......................................... 136
Commands from the Tables menu ........................................ 140
Manipulating columns ........................................................ 145
  Setting and getting attributes ....................................... 148
Row State operators ........................................................... 154
  What are row states? ....................................................... 155
  An overview of row state operators ................................ 159
  Each of the row state operators in detail ......................... 163
  Optional: the numbers behind row states ......................... 169
Calculations ................................................................. 170
  Pre-evaluated statistics ................................................ 170
  Calculator formulas ..................................................... 172
Data table basics

JSL has full access to the values and properties of a data table. This section examines how to open and close data tables, view their properties, and interact with their data.

Once you’ve gotten a data table open, you accomplish most other data tasks by sending messages to a reference to the data table object. Objects that can act on messages will be a common theme in this book, so as soon as you’ve seen how to create or open a data table object, you’ll step back and look at how objects and messages work.

Getting a data table object

Any of the following methods of beginning to work with a data table will produce a data table object.

Opening a data table

Opening a data table returns a reference to the data table. Usually you will assign the reference to a global for later use. The usual argument for `Open` is a path (either relative or absolute) to the data table. For information on using POSIX filenames and the six built-in path variables, see “File Paths,” p. 36 in the “JSL Building Blocks” chapter.

```javascript
Open("$SAMPLE_DATA/big class.jmp"); // just open the data table
dt=Open("$SAMPLE_DATA/big class.jmp"); // open and assign a reference
```

The path can be a literal path (absolute or relative) in quotation marks or an expression yielding a path name. Relative paths are interpreted relative to the location of the .JSL file (if the script has been saved) or the JMP application (for an unsaved script); use `../` to move up one folder level. Example scripts typically use a relative path to open files from the installed Sample Data folder and assume the script is run from a folder inside JMP’s folder, such as the installed Scripts folder. We now also support a `Set Current Directory` command.

If no arguments are given, `Open` presents a dialog prompting the user to navigate to a file.

If a data table with that path is already open, then JMP does not read it again but instead brings that window in the front. If you have made changes, then the table will not be the same as the one on disk you may expect. Since JMP always keeps the whole table in memory, opening is equivalent to reading the whole file.

Importing data from a text file

If you want to import data from a text file, you need to include some additional arguments.

```javascript
Number of Columns(number)
Columns(colName=colType(colWidth),...)
// colType is Character|Numeric
// colWidth is an integer specifying the width of the column
End Of Field (Tab|Space|Comma|Semicolon|Other|None)
EOFOther ("char")
End Of Line (CRLF|CR|LF|Semicolon|Other)
EOLOther ("char")
Strip Quotes|Strip Enclosing Quotes (boolean)
```
Labels|Table Contains Column Headers (boolean)
Year Rule|Two digit year rule ("10-90"| "19xx"| "20xx"| Custom)
Column Names Start|Column Names are on line (number)
Data Starts|Data starts on line (number)
Flags (number)
Custom Rule (name)
Lines to Read
Use Apostrophe as Quotation Mark

For example,
open("c:\test.txt", End of Field(comma), Labels(0),
   columns(studentName=character(12), age=numeric(14)));
or:
open("c:\test2.txt", End of Field(Other), EOFOther ("|"), Labels(1),
   columns(studentName=character(12), age=numeric(14)));

The options for Open:

**Column names and lengths**  You can specify column names, types, and field widths with a
Columns argument, e.g. columns(studentName=character(12), age=numeric(14)) in
the example above.

Note that if you specify settings for a column other than the first column in the file, you must
also specify settings for all the columns that precede it. For example, You want to open a text file
that has four columns, Name, Sex, Age, and ID, in that order, and you want to set Age to
numeric(2). You must also set Name and Sex, and you must list them in the same order:
columns(Name=character(15), Sex=character(1), Age=numeric(2))

You are not required to specify the settings for any columns that follow the one you want to set.

**Strip Quotes**  Boolean. Specifies whether to remove quotation marks from string values.

When saving spreadsheet-style data in text format, most programs, including JMP, enclose string
values inside double quotation marks (" "). This ensures that when another program imports the
text file, any spaces, tabs, or commas that are meant to be part of a string value are not misinter-
preted as field delimiters. For example, a string value such as John Doe would be interpreted as
two separate strings, John and Doe, if space were the field delimiter. But "John Doe" inside
quotation marks would be interpreted as a single string, because when most programs including
JMP read a quotation mark, they disregard any delimiter characters and keep reading until they
encounter the second quotation mark.

However, since you don't usually want those double quotation marks to be part of the value (e.g.,
you want John Doe, not "John Doe", in the data table), JMP provides an option Strip
Quotes. If you choose this option, JMP respects the quotation mark rules but removes the marks
from the values.

Warning: text import in JMP 5 and later is different from JMP 3 and earlier. In version 3, delimiter
characters within quoted string were only ignored if they were not the chosen delimiter char-
acter. For example, if you chose "space" as the delimiter, you could safely use tabs, commas, and
semicolons inside quoted strings, but you could not use a space inside a quoted string. In version
4 and above, text import was improved to ignore even the chosen delimiter, as long as the value is inside double quotation marks.

Another warning: many word processors have a “smart quotes” feature that automatically converts double quotation mark ” characters into left and right curled quotation marks “ ”. These special characters will cause problems with text import and also for JSL.

**End of Line** (CR | LF | CRLF | Semicolon | Other) Specifies the character(s) used to separate records. The choices are CR for carriage returns (typical for Macintosh text files), LF for linefeeds (typical for UNIX text files), or CRLF for both carriage returns and linefeeds (typical for Windows text files). The default is to interpret all three as line delimiters. Other adds another character of your choice to be interpreted as a record separator. Specify the character with EOLOther. Note that specifying Other does not block the default action, but adds to it.

**End of Field** (Tab | Space | Spaces | Comma | Semicolon | Other | None) Specifies the character(s) used to separate fields. The default field delimiter is Tab. Other adds another character of your choice to be interpreted as a record separator. Specify the character with EOFOther. Note: Space uses a single space as a delimiter, while Spaces uses two or more.

**EOFOther, ELOOther** specifies a special character as the end-of-line or end-of-field. For example, ELOOther("*") specifies the asterisk as the end-of-line character.

**Labels** Boolean. Indicates whether the first line of the text file contains column labels or not.

**Year Rule** | **Two digit year rule** specifies the year rule. Options are 10-90, 19xx, 20xx, or Custom.

**Custom Rule** specifies JSL that handles dates in a custom way.

**Column Names Start** | **Column Names are on line** specify the starting line for column names.

**Data Starts** | **Data Starts on line** specifies the starting line for data.

**Lines to Read** Specifies the number of lines to read.

**Use Apostrophe as Quotation Mark** Declares apostrophes as quotation marks.

**Importing other file types**

JMP can also import other supported file types through the Open command.

On Windows, JMP relies on the usual three-letter filename extensions (e.g., .XPT, .XLS, etc.) to identify the type of file and how to interpret its contents. Some examples:

```plaintext
sasxpt=open("carpoll.xpt");
sassd2=open("class.sd2");
sasdbf=open("bigclass.dbf");
xls=open("bigclass.xls");
csv=open("book1.csv");
```

On Macintosh, JMP relies on the Macintosh Type and Creator codes (if present) and secondarily on three-letter filename extensions. (Type and Creator codes are invisible data that enable the Finder to display a file with the correct icon corresponding to the application that created it; files with generic icons (such as often appear on files obtained from other system) should have the filename extensions.)
Open Database

Open Database opens a database using ODBC and extracts data into a JMP data table. See the “Production Environments” chapter.

Starting a new data table

You can start a new data table, or start a new data table and store its reference in a global. In either case, you can specify a name for the table as an argument.

```
New Table();
dt = New Table("myName.jmp");
```

If you want to avoid a window appearing when you create a new data table, use the invisible keyword. For example, to create an invisible table Abc with one column of ten rows,

```
dt = newTable("Abc", invisible, newColumn("X"), addRows(10));
```

**Note:** To prevent “lost” data tables created invisibly, you can only create an invisible data table if you assign it a reference. If you do create an invisible data table without a reference, JMP immediately destroys it, so any further uses of the table result in errors.

**Tip:** Once you are finished with an invisible data table, do not forget to close it. Otherwise, it will remain in memory until you quit JMP.

More ways of opening data

You can open data tables that are accessible through a webpage:

```
open("website url", <extension>)
```

The website URL is a quoted string that exactly describes the internet location of a JMP data table. Add the extension (for example, .jmp) as an unquoted named argument if the URL for the data table does not have the .jmp extension.

Note that the URL is case sensitive.

You can import data the same way. For example, specify text for a text import using your text import preferences.

```
open("http://www.sas.com/", text);
```

If you do not add the text argument, the web page is returned as a string.

Additional formats include csv, script, and journal.

Objects and messages

Many of JMP’s capabilities are performed by scriptable objects within JMP that know how to execute tasks relevant to themselves. Objects are entities that, once created, know how to do certain things for themselves. A JMP data table is one such object.

To work with an object from JSL, you send a message to the object, asking it to perform one of its tasks. Messages are commands that can only be understood in context by a particular type of object. For exam-
Chapter 5

Data Tables
Data table basics

119

Each object defines its own set of messages and what those messages mean. Although there might be
some messages that happen to exist for more than one kind of object, each message’s effect will be determined solely by that object. This is one way in which JSL is heavily context-dependent.
First you’ll see how to address a data table object with a reference, then how to send a message to the reference, and then learn about the messages data table objects can understand. A later section examines
another type of object, data table columns. Subsequent chapters explore analysis and graph platform
objects, display objects, and others.
Storing a reference to a data table
To review, here are the various ways to get a data table open. Notice that each one begins with an
assignment to a name. That is because each of these operators, in addition to creating or opening the
data, return a value which is a reference to the table—a reference to one of JMP’s kinds of scriptable
objects. By assigning the operator to a global variable, you capture that reference in the variable, and
from now on you can use that variable to talk to the object. (This manual uses the placeholder dt to
represent data table references.)
dt=open("..\sample data\big class.jmp"); //Windows
dt=open("::sample data:big class");
//Macintosh
dt=open("$SAMPLE_DATA/Big Class.JMP"); // all operating systems
dt=new table("myTable"); // create a new data table

You can also assign a reference to a table you already have open, in case you didn’t include an assignment in your Open or New Table statement (or if you opened it through the menus):
dt=data table("myTable"); // the open table named myTable
dt=current data table(); // the table in the topmost window
dt=data table(3);
// the third open table
Current Data Table can also take a scriptable object reference as an argument. For example, this
would make current the data table in use by the second Bivariate object:
Current Data Table(Bivariate[2]);

The section “Sending script commands to a live analysis,” p. 182 in the “Scripting Platforms”
chapter, discusses references to analysis platform objects.
Sending a message to a reference
Now, having gotten a reference to a table, you just send it a message with the << operator, also known
as the Send function:
dt=Open("$SAMPLE_DATA/Big Class.jmp");
dt<<save();
Send(dt,save());

Most of the rest of this chapter discusses various messages you can send to data tables, or rows or columns inside data tables. The pattern is always the same:
reference << message(arguments);

Usually saving a reference in a global once and then sending many messages to the globals is most convenient, but the direct route also works, and is easier if you only have one message to send:

5 Scripting Data Tables

ple, messages for data table objects include Save, New Column, Sort, and so on. Most of these
wouldn’t make any sense for, say, a platform object.


Finally, you can stack up a series of messages in one statement. The result of sending one message to an object is a usually reference to the newly-altered object, so then subsequent messages just get sent to the evolving object in turn.

```
dt << set name("myName") << new column("myColumn") << save();
close(dt);
```

### Running a script attached to a data table

The **Run Script** command, when sent to a data table, finds a named property (i.e., a script) and runs it as a JSL script.

For example, if you open `Big Class.jmp` and submit the following, the script named “Distribution” is run.

```
Current Data Table()<<Run Script ("Distribution")
```

### Closing a data table

To close a data table, use a **Close** command with the data table's reference as the argument. If there are unsaved changes, an alert dialog asks whether to save changes. Such a prompt suspends execution of a script; to avoid being prompted, either send a **Save** message first, or include a filename argument for saving in the **Close** message. To close without saving or being prompted, include **No Save** in the command.

```
dt=Open("$SAMPLE_DATA/Big Class.jmp");
close(dt);
close(dt, save("myFile"));
close(dt, No Save);
```

### Reverting to a saved data table

To revert to the most recently saved data table, send the **Revert** message to the data table. Before reverting, JMP asks whether the current data table should be saved.

### Counting open data tables

**N Table()** returns the number of open data tables.

### Getting and using a list of open data tables

If you want to do something to all the data tables that are currently open, you can use **N Table** to get a list of references to each one:

```
openDTs = List();
For( i = 1, i <= N Table(), i++,
   Insert Into( openDTs, Data Table( i ) );
);
```

`openDTs` now is a list of references to all open data tables. You can send messages to any one by using `openDTs(n)`. You can use a for loop to send messages to all of the open data tables. This loop adds a new column named My Column to each open data table.

```
For( i = 1, i <= N Items(openDTs), i++,
```
If you just want a list of table names and not references, use the Get Name message:

```javascript
For( i = 1, i <= N Table(), i++,
    Insert Into( openDTs, Data Table( i )<<Get Name());
);
```

Why are some commands sent to objects and others used directly?

`New Table` and `Open` are commands to create objects that don’t exist yet. Once created, these objects know how to manipulate themselves, so you send them messages requesting changes. To close such objects, you must use a command to its container, because the objects can’t delete themselves. `Close` is a command.

**Learning about data tables’ messages**

`Show Properties` lists the messages a data table can receive in the Log window. `Show Properties` is a command that takes any scriptable object, such as a data table, as its argument.

```javascript
dt=Open("$SAMPLE_DATA/Big Class.jmp");
show properties(current data table());
```

The message list is hierarchical, and `[Subtable]` entries show the menus in which messages are found. For example, the first category of messages is `Tables`, and these are all the messages shown in the main Tables menu. Next are `Analyze` and `Graph` menu items. After these are the items available in the `Table`, `Column`, and `Row` panes of the data table window. Finally, items available only in the scripting language are shown.

The messages labeled as `[Action]` all result in some action being taken. Some are shown to be available for `[Scripting Only]` or `[Interactive Only]`.

The `Analyze` and `Graph` items are listed here because you can send a platform name as a message to a data table to launch the platform for that data table through the usual launch dialog box. The “Scripting Platforms” chapter discusses platform-launching in more detail.

```javascript
dt<<Distribution(Y(height));
```
Data table objects contain column objects, which can handle messages of their own. The later section “Manipulating columns,” p. 145, discusses how to refer to column objects, show their properties, and send them messages.

Show Properties also works with platforms and display boxes; see “Learning the messages an object responds to,” p. 184 in the “SCRIPTING PLATFORMS” chapter and “Learning what you can do with a display box,” p. 232 in the “DISPLAY TREES” chapter. Some general tips that also apply to data table objects appear in “How to interpret the listing from Show Properties,” p. 185 in the “SCRIPTING PLATFORMS” chapter. Refer to the JSL browsers in the Index pane of the JMP Starter Window for a brief overview of any item.

Messages for data tables

This section details the messages you can send to data tables. Previous sections discussed how to start or open data tables and how to send basic messages to them. The next section discusses data table column objects and their messages.

Naming, saving, and printing

Before saving, you might want to give the table a name, or change it to a new name. The Set Name message does this, and its argument is just a filename in quotation marks, or else something that evaluates to a filename.

\[ \text{dt} \ll \text{set name}("new big class.jmp"); \]

There is a command to obtain the name:

\[ \text{name} = \text{dt} \ll \text{Get Name}; \quad \text{// to retrieve the name} \]

To save, you send a Save message. You can also specify the filename (and a complete path) as an argument to a Save message, if you don't want to do a separate Set Name message. The syntax is:

\[ \text{dt} \ll \text{Save}(<\text{"pathname"}>, <\text{"file type"}>, <\text{JMP(5)}>); \]

Some examples:

\[ \text{dt} \ll \text{save}(); \quad \text{// Save using the current name} \]
\[ \text{dt} \ll \text{save}(\text{"c:/mydata/new big class.jmp"}); \quad \text{// Save a new file} \]
\[ \text{dt} \ll \text{save}(\text{"MyTable", JMP(5)}); \quad \text{// Save as JMP 5 table} \]

On Windows, saving with a .txt extension exports according to the current Text Export preferences. On the Macintosh, append the file type as a second argument to the save function when saving to a type other than JMP. Some Macintosh examples are:

Valid file types are:

\[ \text{current data table()} \ll \text{save}(\text{"New Big Class.txt", Text}); \]
\[ \text{current data table()} \ll \text{save}(\text{"New Big Class.sas", SAS}); \]
\[ \text{current data table()} \ll \text{save}(\text{"New Big Class.jrn", Journal}); \]
\[ \text{current data table()} \ll \text{save}(\text{"New Big Class.jsl", Script}); \]

You can print the current data table by sending it a Print Window message.
Journal and Layout

Journal and Layout for a data table are accessible from JSL.

\[
\text{current data table() \text{ \langle\rangle \text{ \text{\texttt{journal}}} \text{ \text{\texttt{journal}}} \text{ \text{\texttt{layout}}}}
\]

You can also create a new journal window and immediately add to it within the `NewWindow()` command. For example:

\[
\text{NewWindow(<<Journal);} \text{\langle\rangle}
\]
creates an empty untitled journal.

\[
\text{NewWindow("title",<<Journal);}
\]
creates an empty journal with the title “title”.

\[
\text{NewWindow("Test Buttons", <<Journal, ButtonBox("Test One",Dialog("Hi there1")), ButtonBox("Test Two",Dialog("Hi there2")) \text{\langle\rangle}}
\]
creates a journal titled “Test Buttons” with two buttons in it.

There is an optional parameter in the journal and journal window commands. This parameter “freezes” the current display. That is, it converts the display tree with just a picture.

The four options for parameters are:

- \text{\langle\rangle \text{\texttt{journal (freeze all)}}}
  wraps the selected display tree in a picture node and takes its picture.

- \text{\langle\rangle \text{\texttt{journal(freeze pictures)}}}
  takes pictures of each picture node.

- \text{\langle\rangle \text{\texttt{journal(freeze frames)}}}
  takes pictures of each frame.

- \text{\langle\rangle \text{\texttt{journal(freeze frames with scripts)}}}
  takes pictures of each frame that has a script.

\text{Current Journal()}\text{\langle\rangle \text{\texttt{current Journal display window. If no journal is open, one is created. There are no arguments.}}}

You can add to the journal using the Append command

\[
\text{Current Journal()}\text{\langle\rangle \text{\texttt{append\langle\rangle display box reference\rangle}}}
\]
For example,

\[
\text{Current Journal()}\text{\langle\rangle \text{\texttt{append\langle\rangle Text Box("Hello World")\rangle}}
\]
You can also find items in an existing journal by enclosing the search specification in square brackets

\[
\text{Current Journal()\text{\langle\rangle \text{\texttt{current Journal() specification}}}}
\]
For example,
Messages for data tables

Current Journal()["Parameter Estimates"]<<Append(Text Box("Asterisks show items significant at 0.05"));

Creating and deleting columns

New columns

To add a new column to an existing data table, send a New Column message to a data table reference. The first argument (required) is the column's name, which must either be a name in quotation marks or else an expression evaluating to that. Further arguments are optional and set attributes for data type (Numeric, Character, or RowState) and analysis type (Continuous, Nominal, or Ordinal). You may also include Values, Formula, and other script messages appropriate to a column.

```
dt<<NewColumn("logHt");
```

```
tt="logHt"; dt<<New Column(tt, numeric, ordinal);
```

```
dt<<New Column("Ratio", Numeric, Continuous, Formula(Height/Weight));
```

```
dt<<New Column("Last Name", Character, Width(16), Values({"Smith", "Jones", "Anderson"}));
```

```
dt<<NewColumn("RowNumber",Numeric,Values(1::NRow()), Format("Best",5));
```

```
dt<<NewColumn("myMarkers",row states, set formula(marker state(age-12)));
```

New Column can also be used as a built-in function, i.e. without the << (send) command applied to the data table reference. When used in this way, it is applied to the current data table. The built-in function and command work identically.

```
New Column("logHt");
```

New Column returns a column reference which can be saved for later use.

```
myCol = dt<<NewColumn("logHt");
```

Further details on column references and how you can send commands to column objects are covered in “Manipulating columns,” p. 145. The commands covered there can all be used as additional arguments to New Column.

You can also add columns with Add Multiple Columns... from the Columns pop-up menu. The JSL equivalent for this is a Add Multiple Columns message. Arguments are, in order, the prefix to use in naming the columns, then how many columns, where to insert them (Before First, After Last, or After(col)), and data type (Numeric, Row State, or Character(width)).

Some examples are:

```
dt<<add multiple columns("beginning",2,before first, rowstate);
```

```
dt<<add multiple columns("middle",3,after(height),numeric);
```

```
dt<<add multiple columns("end",4,after last,character(4));
```

Deleting columns

To delete columns, send a Delete Columns message and specify which column or columns to delete. To delete more than one column, list the columns as multiple arguments or as a list. Without an argument, Delete Columns deletes the currently selected columns.

```
dt<<delete columns("weight", "age", "sex"); // specified column(s)
```

```
dt<<delete columns({"weight", "height"]);
```
To create columns, see “Creating and deleting columns,” p. 124.

Reordering columns

These messages enable you to rearrange columns in a data table:

- `dt<<Reorder By Name;` // alphanumeric order
- `dt<<Reverse Order;` // reverse current order
- `dt<<Reorder By Data Type;` // character then numeric
- `dt<<Reorder By Modeling Type;` // continuous, ordinal, nominal
- `dt<<Original Order;` // saved order

Accessing values

The typical way to access values in a data table is to set up a data table as the “current data table,” specify a row to be the “current row,” and then access the value for a particular column on that row through a column name. This section discusses each of these steps separately, puts them together with a few examples, and then also looks at other strategies.

Current data table

A data table becomes the current data table when you Open it, or when you create it with New Table. To switch to another already-open, already-created data table, you can use the Current Data Table command:

```
Current Data Table(myTable);
```

Current row

The current row is determined by the value of a pseudo-variable called Row(). You can either assign a value to Row() or else use the iterative command For Each Row().

```
Row()=3; ...
For Each Row(...);
```

“Iterating on rows of a table,” p. 127, and “What is the current row?,” p. 127 give more details.

Selected Columns

To get a list of selected columns, use the Get Selected Columns message.

```
dt<<Get Selected Columns
```

To select a column, use the Set Selected message. See more information in “Manipulating columns,” p. 145.

```
col<<Set Selected (1);
```

Selecting Rows

To select certain rows, use the Select Rows message.

```
Current Data Table()<<Select Rows({1, 3, 5, 7})
```
Setting or getting values by column name

The easiest way to refer to a column is by its name. To prevent ambiguity in case you have a global variable that also uses the same name, you can use the : prefix operator to scope it as a column name. It works without the prefix, but we recommend the prefix. To set the cell of the current row of the column name, use the assignment statement with the column name on the left (it is an L-value).

```name = "Archibald";
```

If you want to get the value, just specify the name, preferably scoped with the colon.

```show(:name);
myGlobal = :name;
```

If you want to set a specific row of the column, rather than the current row, use a subscript with the row number.

```:name[rownum] = "Archibald";
rowValue = :name[rownum];
```

By convention, an empty subscript refers to the current row, so :name[ ] is the same as :name.

Confusion alert! Be careful that you are subscripting to a row of the table that exists. The default row number is zero, so statements like :name that refer to row zero generate an Invalid Row Number error.

Putting it all together (example)

Here you first open a table to make it the current data table, then use For Each Row to use every row in turn as the current row, and refer to a column to get its values.

```dt=open("$SAMPLE_DATA/Big Class.JMP");
for each row( print(:age) );
```

Other ways

There are other ways to specify all three elements (the data table, the row, and the column). Some of these involve the indirect reference to a column through a column reference, as covered in detail in “Manipulating columns,” p. 145. All the following lines obtain the value of row 2 of column age.

```:age[2];                     // returns value of age in row 2: 12
column("age")[2];            // same but indirectly thru column reference
col=column("age"); col[2];   // same but with a stored column reference
Row()=2; col[];              // empty subscript means current row
```

Confusion alert! When referring to columns indirectly—using column references—a subscript is required to refer to the individual values. Without a subscript, the reference is to the column object as a whole.

You can specify all three items directly by using the : infix operator and a subscript:

```dt:Age[2] = 12;              // table, column, and subscript
```

If you want to target multiple rows, you can use subscripts with a list or matrix of row numbers.

```if (age<13,print(row()));
if (age[i]<13,print(i));
```
Iterating on rows of a table

In addition to the programming operators for iterating that are built into JSL (see “Iterating,” p. 94 in the “Programming Functions” chapter), JSL has operators for iterating through data table rows, groups, or conditional selections of rows. This section first introduces basic concepts and some useful operators.

Generally, script executes on the current row of the data table only. Some obvious exceptions are the expressions inside formula columns, Summarize and the pre-evaluated statistics operators, and any use of data table columns by analysis platforms. Otherwise, though, a script is performed on the current row of the data table.

What is the current row?

By default, the current row number is 0. The first row in a table is row 1, so row 0 is essentially not a row. In other words, by default, an operation is done on no rows. Unless you take action to set a current row or to specify some set of rows, you get missing values due to the lack of data. For example, a column name returns the value of that column on the current row. Use the prefix : operator to avoid ambiguity (to force the name to be interpreted as a data table column name).

:sex; //returns ""
:age; //returns .

Why? This default protects you from getting a result that might look reasonable for the whole data table but is actually based on only one row. It also protects you from accidentally overwriting data values when making assignments to ambiguous names under most circumstances. (More complete protection is had by using the prefix or infix :: operator to refer unambiguously to a data column and the prefix : operator to refer unambiguously to a global script variable. See “Stored expressions,” p. 415 in the “Advanced Concepts” chapter.)

You can use the Row() operator to get or set the current row number. Row() is an example of an L-value expression in JSL: an operator that returns its value unless you place it before an assignment operator (=, +=, etc.) to set its value.

Row(); //returns the number of the current row (0 by default)
x=Row(); //stores the current row number in x
Row()=7; //makes the 7th row current
row()=7; :age; //returns 12

Note that the current row setting only lasts for the portion of a script you select and submit. After the script executes, the current row setting resets to the default (row 0, or no row). Therefore, a script submitted all at once can produce different results from the same script submitted a few lines at a time.

How many rows and columns?

The NRow and NCol operators count the rows and columns in a data table (or matrix; see the “Matrices” chapter).

NRow(dt); // number of rows
NCol(dt); // number of columns

**For Each Row**

To iterate a script on each row of the current data table, put *For Each Row* around the script.

```plaintext
For Each Row( if(:age>15, show(age)) );
```

A typical use is for setting rowstates without creating a new formula column in the data table. The scripts below are similar except that the first one creates a rowstate column and the *For Each Row* version simply sets the rowstate without creating a column.

```plaintext
new column("My Rowstate", rowstate, formula( color state(age-9)) );
for each row(color of(rowstate())=age-9);
```

To iterate a script on each row meeting a specified condition, combine *For Each Row* and *If*.

```plaintext
for each row(marker of(rowstate())=if(sex=="F",2,6));
```

Break and Continue can be used to control execution of a *For Each Row* loop. For details on using these two functions, see "Break and Continue," p. 96 in the "Programming Functions" chapter.

**Dif and Lag**

JMP has two special operators that are useful for statistical computations, particularly when working with time series or cumulative data. *Lag* returns the value of a column $n$ rows before the current row. *Dif* returns the difference between the value in the current row and the value $n$ rows previous. The following are equivalent:

```plaintext
dt<<new column("htDelta");
for each row( :htDelta=height-lag(height,1) );
for each row( :htDelta=dif(height,1) );
```

**Comparing two data tables (example)**

Here is a little program to compare two data tables and show entries that differ:

```plaintext
dt1=open();
dt2=open();
nr = min(nrow(dt1),nrow(dt2));
nc = min(ncol(dt1),ncol(dt2));
for(i=1,i<=nr,i++,
    for(j=1,j<=nc,j++,
        v1 = Column(dt1,j)[i];
        v2 = Column(dt2,j)[i];
        if(v1!=v2,show(i,ColumnName(j),v1,v2));
    )
);
```

You could make a fancier program to compare data tables that would also check whether they have the same numbers of rows and columns, whether the columns share the same names, and whether they have the same row states in effect.
5 Scripting Data Tables

Chapter 5
Data Tables

Messages for data tables

Summarize

The Summarize command gathers summary statistics for a data table and stores them in global variables (as opposed to Summary, which gathers summary statistics and presents them in a new data table). Named arguments are Count, Sum, Mean, Min, Max, StdDev, and Quantile, and each of these take a data column argument. If a name=By(groupvar) statement is included, then a list of subgroup stats is assigned to each name. Count does not require a column argument, but it is often useful to specify a column to count the number of non-missing values. Quantile also takes a second argument for specifying which quantile, e.g. 0.1 for the 10th percentile.

Note: Excluded rows are excluded from Summarize calculations. If all data are excluded, Summarize returns lists of missing values. If all data have been deleted (there are no rows), Summarize returns empty lists.

For example, using Big Class:

```plaintext
summarize(
    a=by(age),
    c=count,
    sumHt=sum(height),
    meanHt=mean(height),
    minHt=min(height),
    maxHt=max(height),
    sdHt=stddev(height),
    q10Ht=quantile(height,.10));
```

Since the script included a By group, the results are a list and six matrices:

```plaintext
show(a, c, sumHt, meanHt, minHt, maxHt, sdHt, q10Ht);
```

You can format the results using TableBox, as discussed in “Build your own displays from scratch,” p. 258 in the “Display Trees” chapter.

```plaintext
NewWindow("Summary Results",
TableBox(
    stringColBox("Age", a),
    NumberColBox("Count", c),
    NumberColBox("Sum", sumHt),
    NumberColBox("Mean", meanHt),
    NumberColBox("Min", minHt),
    NumberColBox("Max", maxHt),
    NumberColBox("SD", sdHt),
    NumberColBox("Q10", q10Ht)));
```
Figure 5.1 Results from Summarize

With a little work, you can combine total and grouped results in the window:

```r
summarize(
    tc=count,
    tsumHt=sum(height),
    tmeanHt=mean(height),
    tminHt=min(height),
    tmaxHt=max(height),
    tsdHt=stddev(height),
    tq10Ht=quantile(height,.10));
```

```r
c=c|/tc;
sumHt=sumHt|/tsumHt;
meanHt=meanHt|/tmeanHt;
minHt=minHt|/tminHt;
maxHt=maxHt|/tmaxHt;
sdHt=sdHt|/tsdHt;
q10Ht=q10Ht|/tq10Ht;
```

NewWindow("Summary Results",
TableBox(
    stringColBox("Age",a),
    NumberColBox("Count",c),
    NumberColBox("Sum",sumHt),
    NumberColBox("Mean",meanHt),
    NumberColBox("Min",minHt),
    NumberColBox("Max",maxHt),
    NumberColBox("SD",sdHt),
    NumberColBox("Q10",q10Ht)));
Chapter 5

5 Scripting Data Tables

Chapter 5

Data Tables

Messages for data tables

Figure 5.2 Summarize with Total

When you do not use a By group, the result in each name is a single value. Compare:

```julia
summarize(
    // a=by(age),
    c=count,
    sumHt=sum(height),
    meanHt=mean(height),
    minHt=min(height),
    maxHt=max(height),
    sdHt=stddev(height),
    q10Ht=quantile(height,.10));
show(c,sumHt,meanHt,minHt,maxHt,sdHt,q10Ht);
```

```
c:40
sumHt:2502
meanHt:62.55
minHt:51
maxHt:70
sdHt:4.242338493971897
q10Ht:56.2
```

Summarize supports multiple By-groups. For example, on Big Class.jmp,

```julia
Summarize(g=by(age, sex), c=count());
show(g, c);
```

results in

```julia
```

The statistical results of Summarize are always matrices if there is a By clause, or scalars if there is no By clause.

Working with metadata

The main purpose for a JMP data table is to store observation data—measurements of various variables on a specific set of subjects. However, JMP data tables can also store metadata, or data about the data. Metadata include information such as the source of the data, comments about each variable, scripts for working with the data, analysis roles for columns, and so on.
Tools for working with metadata are found in the Table pop-up menu in the upper left pane of a data table, in the Column Info dialogs, and so on. This section discusses how to work with metadata in JSL.

### Table variables

Table variables are for storing a single text string value, such as “Notes” in Big Class. To see how variables work, first get its existing value and assign that to a global, `oldvar`, by sending a `Get Table Variable` message:

```julia
dt=open("$SAMPLE_DATA/conditional logit.jmp");
dt<<get table variable("Notes");
"Breslow-Day[1980] Conditional Logistic regression recipe: make all outcomes 1 and freq 1, add a dummy row with outcome 2, with Frequency zero. Regress outcome on the differences of the regressors. [Endometrial cancer data]"
```

Now change the source from the string by using `Set Table Variable`, and then get the variable to check your work.

```julia
dt<<set table variable("Notes","Endometrial cancer data");
dt<<get table variable("Notes");
"Endometrial cancer data"
```

You can create new variables of your own with the `New Table Variable` message, such as to store the source of the data table in its own table variable:

```julia
dt<<new table variable("Origin","Breslow-Day [1980]");
dt<<new table variable("Last used", abbrev date(today()));
```

If you specify too few arguments for any of these messages, JMP presents a dialog to get the necessary information from you. (JMP often presents dialogs when your script is incomplete, a behavior that you can use to your advantage when writing scripts that need to query users for their choices.)

### Properties/Scripts

A `property` is similar to a table variable, but it is used to store an expression, e.g. a JSL script. For example, if you do an analysis and then use the `Save Script to Data Table` command from the report’s pop-up menu, it saves a script named for the analysis platform in the data table. In the data table window, you can double-click the property name to see its script value, or you can select `Run Script` from its pop-up menu to execute the script. Also, when you run Design of Experiments (DOE), the resulting data tables automatically include properties called Model to launch Fit Model with the appropriate design. You can save a table property named QC Alarm Script or QC Test Alert to set up an alert script for control charts; see “Alarm scripts,” p. 218 in the “Scripting Platforms” chapter.

JSL has `New Table Property`, `Set Property`, and `Get Property` messages.

```julia
dt<<new table property("Bivariate", Bivariate(Y(weight),X(height)));
dt<<get property("bivariate");
Bivariate(Y(weight),X(height))
dt<<set property("Bivariate",Bivariate(Y(weight),X(height),Fit Line));
dt<<get property("bivariate");
Bivariate(Y(weight),X(height),Fit Line)
```

If you specify too few arguments for any of these messages, JMP presents the appropriate dialog to get the necessary information from you.
See also “Setting and getting attributes,” p. 148, for ways to store extra information about columns.

**Deleting Variables, Properties, and Formulas**

Given a data table `dt` or column `col` with a property or variable `name`, the following commands delete them.

```
dt << Delete Table Variable(name);
dt << Delete Table Property(name);
col << Delete Formula;
col << Delete Property(name);
col << Delete Column property(name);
```

**On Open property**

A script named `On Open` stored as a property of the data table will automatically be executed when the data table opens. Either use the Save Script to Data Table from pop-up menus to save a script as a property in the current data table and then double-click the property name for a dialog box to change its name to `On Open`, or else store the script using a New Table Property message. In this example, you send the Sort message to Current Data Table() rather than `dt`, since `dt` might not be defined when the data table is opened.

```
dt<<new table property("OnOpen", sortedDt=current data table()<<Sort(By(Name), output table name ("Sorted big class")));```

A system preference lets you suppress the automatic execution of `On Open` scripts. As a precaution, you should consider suppressing auto-execution when opening data tables you receive from others:

```
preference(suppress On Open script eval(1));```

**Running a script at start up**

If you want to run the same script every time you start JMP, name it `jmpStart.jsl` and place it in the Builtin Scripts folder:

- On Windows and Linux, this folder is inside the Support Files folder for your locale, which is within the folder holding the JMP executable file (for example, `C:\Program Files\SAS\JMP7\Support Files <Language>\Builtin Scripts`).
- On Macintosh, this folder is inside the application package. Right-click the JMP application and select Show Package Contents. Then navigate to Contents/Resources/<lan>.lproj/Builtin Scripts. The name of the `.lproj` folder will depend on the locale you are running.

**Scripts**

Suppose you want a text representation of a data table, perhaps to email to a colleague or to use as part of a script. You can obtain a script that will reconstruct the information in a data table with Get Script:

```
dt<<get script;
New Table("Class", Add Rows(19), New Table Variable("Notes", "Example data to use for simple demonstrations of each platform. Use Name as a label, and any combination of the variables as Xs and Ys"), New Table Variable("Distribution", ""), New Table Variable("Oneway", ""), New Table Property("Distribution", Distribution(Column(Age, Height))));
```
Data Tables

Chapter 5

Messages for data tables


Controlling formula evaluation

The message Suppress Formula Eval takes a boolean argument to specify whether formula evaluation should be suppressed or not. You might want to suppress evaluation if you plan to make numerous changes to the data table and don't want to wait for formula updates between steps.

\[
dt<<\text{suppress formula eval}(1);
\]
\[
dt<<\text{suppress formula eval}(0);
\]
\[
dt<<\text{suppress formula eval}; //toggle current state
\]

To accomplish the same effect for all data tables, use the Suppress Formula Eval command to turn off formulas globally. This is the same as the message above except that you do not send it to a data table object.

\[
\text{suppress formula eval}(1); // make formulas static globally
\]
\[
\text{suppress formula eval}(0); // make formulas dynamic globally
\]

Note that formulas are not evaluated when they are installed in the column. Even when you force evaluation, they end up getting evaluated again in the background. This is a problem for scripts if they depend on the column having the values while the script is running. Some users need a mechanism to control evaluation, provided by the following commands.

To force a single column to evaluate, send an EvalFormula command to the column. You can even do this inside the command to create the column, after the formula clause. For example,

\[
currentDataTable()<<(\text{New Column("Ratio", Numeric, Formula(:height/:weight), EvalFormula})
\]
\[
dt<<\text{Run Formulas} performs all pending formula evaluations—those that are pending, or those that are scheduled as a consequence of evaluating other formulas. It is useful when there are a whole series of columns to run. This method is preferred, since although EvalFormula evaluates the formulas, it does not suppress the background task from evaluating them again. The background task takes great care to evaluate the formulas in the right order.

If you also send the Run Formulas command to data column, the evaluation is done at the time of the command, but it does not suppress the scheduled evaluations that are pending. Therefore, formulas may end up being evaluated twice if you also send the command to the data table. Being evaluated twice may be desired for formulas that have random function in them, or it may be undesirable if they
depend on randomization seeds being set. If you use random numbers and use the Random
Reset(seed) feature to make a duplicatable sequence, then use the Run Formulas command, since
it avoids a second evaluation.

**Note:** Starting in version 6, all platforms send a Run Formulas command to the data table to assure
that all formulas have finished evaluating before analyses start.

### Setting Values Without a Formula

col<<Set Each Value(expression) evaluates the expression for each row of the data table and
assigns the result to the column. It does not store the expression as a formula.

### Controlling Display Updating

Whenever you change values in a data table, messages are sent to the displays to keep them up-to-date. However, if you have thousands of changes in a script, this increases the time it takes to complete the updates.

In order to speed up changes, use Begin Data Update before the changes to block these update mes-
sages, and End Data Update after the changes have been completed to release the messages and
update the displays. Make sure to always send the End Data Update message, otherwise the display
will not update until forced to do so in some other way.

For example,

```plaintext
Current Data Table() << Begin Data Update;
...<many changes>...
Current Data Table() << End Data Update;
```

### Resizing the Data Table

dt << Maximize Display forces the data table to remeasure all its columns and zoom to the
best-sized window. For example,

```plaintext
Current Data Table() << Maximize Display;
```

### Converting between matrices and data tables

If `dt` is a variable containing a reference to a data table, then Get As Matrix creates a matrix `A` with
the values from all numeric columns in the data table:

```plaintext
A = dt<<GetAsMatrix;
```

This also works with column objects, as discussed under “Values,” p. 148. If `col` is a variable containing
a reference to a data column in a data table, then Get As Matrix will create a matrix `A` as a column
vector of values from the column.

```plaintext
A = col<<GetAsMatrix;
```

In addition, the As Table (matrix) command creates a new data table from a matrix argument.
The columns are named Col1, Col2, and so forth.

Get All Columns As Matrix returns the values from all columns of the data table in a matrix.
Character columns are numbered according to the levels, starting at 1.

Set Matrix creates a data table (or adds data to an existing data table), making new rows and new col-
umns as needed to store the values of the matrix. The new columns are named col1, col2, and so forth.
For example,

```julia
newTable("A", Set Matrix([1 2 3, 4 5 6, 7 8 9]));
```

creates a new data table named A with three rows and three columns.

```julia
dt=New Table("B");
dt<<Set Matrix([1 2 3 4 5, 6 7 8 9 10]);
```

creates a new data table with two rows and five columns.

### Commands from the Rows menu

This section describes the messages for working with the rows in a data table. Row messages are directed to a data table reference, and most act on the currently selected rows. Many of the row messages would not usually be useful in scripts, unless you were writing demo scripts that simulate interactive use of JMP.

#### Adding rows

To add rows, send an **Add Rows** message and specify how many rows, and (optionally) after which row to insert them. The arguments can either be numbers or expressions that evaluate to numbers.

```julia
dt=Open("$SAMPLE_DATA/Big Class.jmp");
dt<<Add Rows(3); // adds 3 rows to bottom of data table
```

```
dt<<Add Rows(3, 10); // adds 3 rows after the 10th row, moving the 11th and lower rows further down
```

A variation of **Add Rows** lets you specify an argument yielding a list of assignments. Assignments can be separated with commas or semicolons.

```julia
dt<<Add Rows({:name="Peter", :age=14, :sex="M", :height=61, :weight=124});
```

```julia
add point = expr( dt<<addRows({:xx=x; :yy=y}) );
```

You can send several arguments yielding lists, or even a list of lists. This script creates a data table with **Add Rows** commands of each variety.

```julia
dt = new Table("Cities");
dt<<NewColumn("xx",Numeric);
dt<<NewColumn("cc",Character,width(12));
```

```julia
dt<<AddRows({xx=12,cc="Chicago"});                  // single list
```

```julia
dt<<AddRows({xx=13,cc="New York"},{xx=14,cc="Newark"}); // several lists
```

```julia
dt<<AddRows({{xx=15,cc="San Francisco"},{xx=sqrt(256),cc="Oakland"}}); // list of lists
```

```julia
a = {xx=20,cc="Miami"};
dt<<AddRows(a); // evaluates as single list
```

```julia
b={{xx=17,cc="San Antonio"},{xx=18,cc="Houston"},{xx=19,cc="Dallas"}};
dt<<AddRows(b); // evaluates as list of lists
```

Further details for rows can be specified with messages described under “Row State operators,” p. 154.
Deleting rows

To delete rows, send a `Delete Rows` message and specify which row or rows to delete. To delete more than one row, give a list or matrix as the `rownum` argument, or combine `Delete Rows` with other commands such as `For`. The `rownum` argument can be a number, list of numbers, range of numbers, matrix, or an expression that yields one of these. Without an argument, `Delete Rows` deletes the currently selected rows. With neither an argument nor rows selected, `Delete Rows` does nothing.

```julia
dt = Open("$SAMPLE_DATA/Big Class.jmp");
selected(rowstate(10)) = 1; dt << delete rows;  // selects, deletes row 10

dt << Delete Rows(10);  // deletes row 10
dt << Delete Rows({11, 12, 13});  // deletes rows 11-13
myList = {11, 12, 13}; dt << Delete Rows(myList);  // deletes rows 11-13
dt << Delete Rows(1::20);  // deletes first 20 rows
dt << Delete Rows([1 2 3]);  // deletes first 3 rows
```

You can list duplicate rows, and you can list rows in any order with no consequence.

Here’s a general way to remove the bottom \( x \) rows of a data table of any size. `NRow` (see “For Each Row,” p. 128) counts the rows in the table.

```julia
x = 5;
n = NRow(dt); For(i = n, i > n - x, i --,
    dt << Delete Rows(i));
```

Selecting rows

Select All Rows selects (highlights) all the rows in a data table.

```julia
dt << Select All Rows;
```

If all rows are selected, you can deselect them all by using Invert Row Selection, which simply reverses the selection state for each row, so that any selected rows are deselected, and any unselected rows are selected.

```julia
dt << Invert Row Selection;
```

With the exception of Invert Row Selection, whose result depends on the current selection, any new selection message starts over with a new selection, so if you have certain rows selected and you then send a new message to select rows, an implied ‘deselect selected rows’ happens first.

To select a certain row, use Go To Row:

```julia
dt << Go To Row(9);
```

You can select rows that are currently excluded, hidden, or labeled:

```julia
dt << Select Excluded;
dt << Select Hidden;
dt << Select Labeled;
```

To select rows that are not excluded, hidden, or labeled, stack a select message and an invert selection message together in the same statement, or send the two messages sequentially:

```julia
dt << Select Hidden << Invert Row Selection;

dt << Select Hidden;
dt << Invert Row Selection;
```
To select rows according to data values, use `Select Where`, specifying a logical test inside the parentheses. You can use all the usual functions and operators from the formula calculator, which are documented in the “JSL Operators” chapter and summarized in “Data Tables, Rows, and Columns,” p. 512 in the “JSL Syntax Reference” appendix. For example, using `Big Class.jmp`, you might use:

\[
\text{dt<<Select Where(age>13);} \\
\]

To select specific rows in a data table based on their row number, use the `Select Rows` command. The parameter to the command is a list of row numbers. For example, to select rows 1, 3, 5, and 7 of a data table,

\[
\text{dt<<Select Rows({1, 3, 5, 7})} \\
\]

To obtain a random selection:

\[
\begin{align*}
\text{dt<<Select Randomly(number)} \\
\text{dt<<Select Randomly(probability)}
\end{align*}
\]

These will use a conditional probability to obtain the exact count requested.

The row menu command `Select Matching Cells` is also implemented in JSL.

\[
\begin{align*}
\text{Current Data Table()} & \text{<< Select Matching Cells;} \\
\text{Current Data Table()} & \text{<< Select All Matching Cells}
\end{align*}
\]

selects matching cells in the current data table.

For more complicated selections, or to store selections permanently as row state data, see the discussion of “Row State operators,” p. 154.

### Moving Rows

These commands move the currently selected rows to the indicated destination point.

\[
\begin{align*}
\text{dt<<Move Rows(AtStart)} \\
\text{dt<<Move Rows(AtEnd)} \\
\text{dt<<Move Rows(After(rowNumber))}
\end{align*}
\]

### Finding Rows

These two commands return a matrix of row numbers:

\[
\begin{align*}
\text{result = dt<<Get Rows Where(condition);} & \text{ // e.g. } \text{<<Get Rows Where(:age>=16)} \\
\text{result = dt<<Get Selected Rows;}
\end{align*}
\]

### Assigning colors and markers to rows

You can use the `Colors` and `Markers` messages to assign (or change) colors and markers used for rows. These settings mostly affect graphs produced from the data table. Both messages expect numeric arguments to pick which color or marker to use; how numbers correspond to colors and markers is summarized in “Colors and markers,” p. 165.

\[
\begin{align*}
\text{dt<<Colors(3);} & \text{ //set selected rows to red} \\
\text{dt<<Markers(2);} & \text{ //pick the X marker for selected rows}
\end{align*}
\]

As with other row messages, you can stack selection and other messages together:

\[
\begin{align*}
\text{dt<<Select Where(age==13)} & \text{ // select the youngest subjects}
\end{align*}
\]
<<colors(8)<<markers(8); // and use purple open circles for them

Color By Column sets colors according to the values of a column you specify, and Marker by Column is analogous:

```
dt<<Color By Column(:age);
dt<<Marker By Column(:age);
```

### Toggling rows between states

For example, to hide all rows in Big Class where age is greater than 13, you could do the following.

```
dt<<Select Where(age>13);
dt<<Hide;
```

Since messages to the same object can be stacked together in a single statement, you could equivalently do this:

```
dt<<Select Where(age>13)<<hide;
```

Similar messages let you exclude or unexclude, label or unlabel:

```
dt<<Exclude;
dt<<Label;
```

These messages are all toggles; send the message again to unhide, unexclude, and so on.

### Locating selected rows

Next Selected and Previous Selected scroll the data table window up or down so that the next selected row that is not already in view moves into view. The table “wraps,” so Next Selected will jump from the bottommost selected row to the topmost and vice versa for Previous Selected.

```
dt<<Next Selected;
dt<<Previous Selected;
```

### Clear selection

To cancel a selection, leaving no rows selected, use Clear Select:

```
dt<<Clear Select;
```

### Working with row states

Row states are a special type of data. You can work with row states both through the messages to data tables discussed above and through row state operators, which are discussed at length in the section “Row State operators,” p. 154. Please refer to that discussion for details about the row states mentioned above.

### Data Filter

Data Filter provides a variety of ways to identify subsets of data. Using Data Filter commands and options, you can select complex subsets of data, hide these subsets in plots, or exclude them from analyses.

The basic syntax is:

```
dt<<Data Filter( <mode( ... )>, <Add Filter ( ... )>, ... );
```
Options for Data Filter include:

- Show Columns Selector, Animation, Animate, Clear, Delete All, Delete, columns, Filter Columns, Filter Column, Filter Group, Add Filter, Add, Match, Mode, Display, Where, Make Subset, display

You can send an empty Data Filter message to a data table, and the initial Data Filter window appears, showing the Add Filter Columns panel that lists all the variables in the data table.

Mode takes three arguments, all optional:

- Select(bool), Show(bool), Include(bool)

They either turn on or turn off the corresponding options. The default value for Select is true (1). The default value for Show and Include is false (0).

Add Filter adds rows and builds the where clauses that describe a subset of the data table. The basic syntax is:

```
Add Filter( columns( col, ... ), Where( ... ), ... )
```

To add columns to the data filter, list the columns names separated by commas. Note that this is not a list data structure.

You can define one or more where clauses. For example:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );

df = dt << Data Filter(
    Mode( Show( 1 ) ),
    Add Filter(
        columns( :age, :sex, :height ),
        Where( :age == {13, 14, 15} ),
        Where( :sex == "M" ),
        Where( :height >= 50 & :height <= 65 )
    )
);
```

You can also send messages to an existing Data Filter object:

- Clear(), Display( ... ), Animate(), Mode(), ...

Clear takes no arguments and clears the data filter.

---

**Commands from the Tables menu**

The messages in this section provide different ways of rearranging data tables and making new tables. If no arguments are supplied to these messages, they produce the usual dialog box for interactive use. If these operations are unfamiliar, see the User Guide for more details and examples.

Column arguments also allow list arguments, so the following script fragments are all valid:

```
Stack(height,weight)
Stack({height,weight}) // equivalent
```

Most of these commands also generate small scripts that record how the resulting tables were made. The script is stored in the new table as a table property with the name “Source.” Exceptions are Summary when performed by script or Subset when performed by script or the dialog box. Most peo-
ple don’t write data table scripts from scratch, but instead use the Table menu items and then copy the Source property from the resulting table.

Summary

Summary creates a new table of summary statistics according to the grouping columns you choose.

```
summDt = dt<<Summary(
    Group(groupingColumns),
    Subgroup(subGroupColumn),
    Statistic(columns),//where statistic is Mean, Min, Max, Std Dev, etc.
    Output Table Name("newName"));
```

For example:

```
summDt = dt<<Summary(
    Group(Age),
    Mean(Height,Weight), Max(Height), Min(Weight),
    output table name("Height-Weight Chart"));
```

(Do not confuse Summary, which creates a data table with summary statistics, with Summarize (p. 129), which stores summary statistics in global variables for later use in custom result displays, etc.)

Subset

Subset creates a new data table from rows you specify. If you specify no rows, Subset uses the selected rows; if no rows are selected or specified, it uses all rows. If no columns are specified, it uses all columns.

```
dt << Subset(
    Columns(columns),
    Rows(row matrix),
    Linked,
    Output Table Name("name"),
    Copy Formula (1 or 0),
    Sampling rate (n),
    Suppress Formula Evaluation(1 or 0));
```

For example:

```
//To select all columns for all rows where age is 12:
for each row(Selected(Rowstate())=(age==12));
subdt = dt<<Subset(output table name("subset"));

//To select three columns and all rows:
subDt1 = dt << Subset (Columns(Name,Age,Height), Output Table Name("Big Class NAH"));

//To select specified rows of two columns, linking:
subDt2 = dt << Subset (Columns(Name,Weight),Rows([2,4,6,8]),Linked);
```
Sort

Sort rearranges the rows of a table according to the values of one or more columns, either replacing the current table or creating a new table with the results. Specify ascending or descending sort for each column listed for By.

\[ \text{dt} \ll \text{Sort(} \]
\[ \begin{align*}
\ & \text{By(columns), Order(Descending | Ascending, ...),} \\
\ & \text{Replace Table | Output table name("name"))}
\end{align*} \]

For example:

\[ \text{sortedDt} = \text{dt} \ll \text{Sort(} \]
\[ \begin{align*}
\ & \text{By(Age, Name),} \\
\ & \text{Order(Descending, Ascending),} \\
\ & \text{output table name("Sorted age name")};
\end{align*} \]

Stack (unsplit)

Stack combines values from several columns into several rows in one column.

\[ \text{dt} \ll \text{Stack(} \]
\[ \begin{align*}
\ & \text{Stack(columns), // the columns to stack together} \\
\ & \text{Id(columns), // to identify source columns} \\
\ & \text{Stacked("name"), // name for the new stacked column} \\
\ & \text{Output Table Name("name")); // name for the new data table}
\end{align*} \]

For example, where \( dt \) is a reference to Big Class:

\[ \text{stackedDt} = \text{dt} \ll \text{Stack(} \]
\[ \begin{align*}
\ & \text{Stack(Height, Weight),} \\
\ & \text{id("ID"),} \\
\ & \text{Stacked("Y"),} \\
\ & \text{Output table name("Stacked");}
\end{align*} \]

Split (unstack)

Split breaks a column into several columns, mapping several rows in the first into single rows in other columns.

\[ \text{dt} \ll \text{Split(} \]
\[ \begin{align*}
\ & \text{Split(columns), // the column to split} \\
\ & \text{Group(columns), // (optional) column to identify rows uniquely} \\
\ & \text{ColID(column), // the grouping variable on which to split} \\
\ & \text{Remaining Columns( Keep All | Drop All | Drop(columns) | Keep(columns)),} \\
\ & \text{Output Table Name("name"));}
\end{align*} \]

The optional Remaining Columns argument specifies which of the other columns from the source data table (those not specified for Split, Group, or ColID) to include in the new data table. The default is to Keep All, or you can explicitly list columns to Keep or Drop.

This example reverses the previous example for Stack, returning essentially the original table, except that the height and weight columns now appear in alphabetic order:

\[ \text{splitDt} = \text{stackedDt} \ll \text{Split(} \]
\[ \begin{align*}
\ & \text{split(y),} \\
\ & \text{ColID(ID),}
\end{align*} \]
output table name("Split");

**Transpose**

Transpose creates a new data table by flipping a data table on its side, interchanging rows for columns and columns for rows. If you specify no rows, Transpose uses the selected rows; if no rows are selected, it uses all rows.

```plaintext
dt << Transpose(
  Columns(columns),
  Rows(row matrix),
  Output Table Name("name"));
```

For example:

```plaintext
tranDt = dt<<Transpose(Columns(Height,Weight),
  output table name("Transposed Columns"));
```

**Note:** In previous versions of JMP, Transpose did not have a dialog. Starting with JMP 5.1, a dialog appears when the transpose command is invoked. The simple transpose command `dt<<Transpose` shows this dialog. If you are doing a simple transpose (of all selected columns and no By column) and you do not want the dialog to appear, invoke the transpose as `dt<<Transpose(no option)`.

**Concatenate (vertical join)**

Concatenate combines rows of several data tables top to bottom.

```plaintext
dt << Concatenate( DataTableReferences,...,Keep Formulas,
  Output Table Name("name"));
```

For example, if you make subsets for males and females, you can stick them back together with Concatenate:

```plaintext
dt<<Select Where(sex=="M"); m=dt<<Subset(output table name("M"));
dt<<Invert Row Selection; f=dt<<Subset(output table name("F"));
both=m<<Concatenate(f,output table name("Both"));
```

Instead of creating a new table containing all the concatenated data, you can append all the data to the current data table:

```plaintext
dt << Concatenate(DataTableReferences, Append to first table);
```

**Join (horizontal concatenate)**

Join combines data tables side to side.

```plaintext
dt<<Join(  // message to first table
  With(dataTable), // the other data table
  Select(columns), // optional column selection
  SelectWith(columns), // optional column selection
  By Row Number, // default join type; alternatives are Cartesian Join or
  // By Matching Columns(col1==col2,col)
  // options for each table:
  Drop Multiples(boolean,boolean),
  Include NonMatches(boolean,boolean),
  Output Table Name("name")); // the resulting table
```
To try this, first break Big Class into two parts.

```plaintext
part1=dt<<Subset(Columns(Name, Age, Height), Output Table Name("NAH_Big Class"));
part2=dt<<Subset(Columns(Name, Sex, Weight), Output Table Name("NSW_Big Class"));
```

To make it a realistic experiment, rearrange the rows in part 2.

```plaintext
sortedPart2=part2<<sort(by(name), Output Table Name("SortedNSW_Big Class"));
```

Now you have a data set in two separate chunks, and the rows aren’t even in the same order, but you can join them together by matching on the column the two chunks have in common.

```plaintext
joinDt = part1<<Join(
    With(sortedPart2),
    By Matching Columns(name==name),
    output table name("Joined Parts"));
```

The resulting table has two copies of the name variable, one from each part, and you can inspect these to see how Join worked. Notice that you now have four Robert rows, because each part had two Robert rows (there were two Roberts in the original table) and Join formed all possible combinations.

### Update

Update replaces data in one table with data from a second table.

```plaintext
dt<<Update(              // message to first table
    With(dataTable), // the other data table
    By Row Number, // default join type; alternative is
    // By Matching Columns(col1==col2)
    Ignore Missing, // optional, does not replace values with missing values
);    
```

To try this, make a subset of Big Class.

```plaintext
NewHt=dt<<Subset(Columns(Name, Height), Output Table Name("hts"));
```

Next, add 0-6 inches to each student’s height.

```plaintext
diff = random uniform(6,0);
for each row(height+=diff);
```

Finally, update the heights of students in Big Class with the new heights from the subset table.

```plaintext
dt<<Update(
    With(NewHt),
    By Matching Columns(name==name),
);
```

### Merge Update

Merge Update is an alias for Update.
Tabulate

Tabulate constructs tables of descriptive statistics. The tables are built from grouping columns, analysis columns, and statistics keywords. This example tabulates Big Class.jmp.

```
dt<<Tabulate( // message to data table
    Add Table( // start a new table
        Column Table( Grouping Columns( :sex ) ), // group using the column sex
        Row Table( // add rows to the table
            Analysis Columns( :height, :weight ), // use the height and weight columns for the analysis
            Statistics( Std Dev, Mean ) // show the standard deviation and mean
        )
    ));
```

Missing Data Pattern

If your data table contains missing data, you may want to see if there is a pattern that the missing data creates.

```
dt<<Missing Data Pattern( // message to data table
    columns( :miss ), // find missing data in this column
    Output Table( "Missing Data Pattern" ) // name the output table
);
```

**Manipulating columns**

This section discusses ways to set, create, and modify columns. Recall that to add and delete columns, you need to send a message to the data table; see “Messages for data tables,” p. 122. To get or set data column values, you specify the current data table, current row, and column, as detailed in “Accessing values,” p. 125.

Obtaining Column names

Column Name(n) returns the name of the n-th column.

```
column name(2); // returns age
```

The returned value is a name value, not a quoted string. What this means is you can use it anywhere you would normally use the actual name in a script. For example, you could subscript it:

```
column name(2)[1];
```

If you want the name as a text string, just quote it with Char:

```
char(column name(2));
"age"
```

To retrieve a list of the names of all columns in a data table, submit Get Column Names.

```
dt<<Get Column Names(argument)
```

where the optional argument controls the output of the Get Column Names function as follows:
Data Tables
Chapter 5

Manipulating columns

- Specify `Numeric`, `Character`, or `RowState` to include only those column data types.
- Specify `Continuous`, `Ordinal`, or `Nominal` to include only those modeling types.
- Specify `String` to return a list of strings rather than column names.

For example,
```julia
names=dt<<Get Column Names (Numeric, Continuous)
returns
{height, weight}
```

Also,
```julia
names=dt<<Get Column Names(Numeric, Continuous, String)
returns
{"height", "weight"}
```

**Select Columns**

To select a column, use the `Set Selected` message.
```julia
col<<Set Selected (1);
```

For example, to select all Continuous variables in the Big Class table, use the following script:
```julia
dt=Open("$SAMPLE_DATA/Big Class.jmp");
cc=dt<<Get Column Names(Continuous);
ncols = N Items(cc);
for(i=1,i<=ncols,i++,
    cc[i]<<Set Selected(1);
);
```

**Get Selected Columns**

To get a list of currently selected columns, use the `Get Selected Columns` message.
```julia
dt << Get Selected Columns
```

**Moving Columns**

These commands move the currently selected columns to the indicated destination point.
```julia
dt<<Move Selected Columns(To First)
dt<<Move Selected Columns(To Last)
dt<<Move Selected Columns(After("name"));
```

You can also move columns without first selecting them in the data table by using the following syntax.
```julia
dt<<Move Selected Columns({"name"}, To First)
dt<<Move Selected Columns({"name"}, To Last)
dt<<Move Selected Columns({"name"}, After("name"));
```
Data column objects

Just as you send data table messages to a data table reference, you send column messages to a reference to a data column object. The `Column` function returns a data column reference. Its argument is either a name in quotation marks (or something that evaluates to a name in quotation marks) or a number.

```
Column("age");  // a reference to the age column
col = Column(2); // assigns a reference to the second column
```

This manual uses `col` to represent data column references. To see the messages you can send to data column objects, use `Show Properties`:

```
show properties(col);
```

Accessing cell values through column references

Always use a subscript on a column reference to access the values in the cells of columns. It is common to use an empty subscript to refer to the current row.

```
x = col[irow]   // specific row
x = col[]      // current row
col[irow] = 2; // as an l-value for assignment
currentDataTable() << Select Rows Where(col[]<14); // in a where clause
```

Sending messages to columns

You saw above how to store a data column reference in a global variable, such as `col`. The rest of the ways to manipulate columns are all messages that should be sent to a data column reference.

Sending messages to columns is analogous to sending messages to data tables. Either state the object, a double-angle operator `<<`, and then the message with its arguments in parentheses; or use the `Send()` function with the object and then the message. In some cases the messages themselves need no arguments, so the trailing parentheses are optional.

```
col << message(arg, arg2, ...);
Send(col, message(arg, arg2, ...));
```

As with data tables and other types of objects, you can stack or list messages.

```
col << message << message2 << ...
col << {message, message2, ...};
```

Note that to delete a column, you need to send a message to a data table reference, because objects cannot delete themselves—only their containers can.

Confusion alert! JMP can now display math symbols and Greek letters (controlled in Preferences in the Font category). This means that if you save a column (such as T square limits), the column name could either be “T Square Limits” (no special characters) or “T² Limits” (with special characters). Any reference using the column name must match the name exactly or it will fail.
Setting and getting attributes

A collection of message pairs for data table columns let you control all the various attributes or characteristics of a column, including its name, its data, and its metadata. The messages come in pairs, one to “set” or assign each attribute and one to “get” or query the current setting of each attribute.

Hide, exclude, label, and scroll lock for a column can be activated and deactivated through scripting. Submit a 1 to turn the column attribute on, and a zero to turn it off. For example, the following four lines of code scroll lock a column called Name, add it to the list of label columns, unexclude it, and unhide it.

```
column("name") << scroll lock(1);
column("name") << label(1);
column("name") << exclude(0);
column("name") << hide(0);
```

To clear a column selection, submit a Clear Column Selection message to the data table.

```
dt<<Clear Column Selection;
```

Names

Set Name lets you name or rename a column, and Get Name returns the name for a column.

```
col<<Set Name("ratio");
col<<Get Name;
```

Values

Similarly, Set Values sets values for a column. If the variable is character, the argument should be a list; if numeric, a matrix (vector). If the number of values is greater than the current number of rows, the necessary rows are added. Get Values returns the values in list or matrix form, and Get As Matrix is a synonym.

```
col<<Set Values(myMatrix); // for a numeric variable
col<<Set Values(myList); // for a character variable
col<<Get As Matrix; // returns a matrix, or list if character
```

For example:

```
dt=Open("$SAMPLE_DATA/Big Class.jmp");
column("name")<<values({Fred, Wilma, Fred, Ethel, Fred, Lamont});
myList = :name<<get values; //returns list
myList = :name<<get as matrix; //equivalent

column("age")<<values([28,27,51,48,60,30]);
myVector = :age<<get values; //returns one-column matrix
```
myVector = :age<<get as matrix; //equivalent

Value Labels

Complete details on value labels are found in the JMP User Guide. Essentially, they provide a method of displaying a descriptive label for abbreviated data. For example, you may have a data column holding values that are 0 or 1, where 0 represents a male and 1 represents a female. Value labels allow you to show the descriptive male/female label instead of the 0/1 label.

You can specify value labels in any of three ways. In each of the following examples, assume M maps to Male, and F maps to Female.

Using two lists

:sex << Value Labels({"F", "M"}, {"Female", "Male"});

Using a list of pairs

:sex << Value Labels({"F", "Female", "M", "Male"});

Using a list of assignments

:sex << Value Labels({"F" = "Female", "M" = "Male"})

In any case, activate value labels by sending Use Value Labels as a message to the column.

:sex << Use Value Labels(1);

To revert back to showing the column's actual values,

:sex << Use Value Labels(0);

The same message can be used for the data table to turn value labels on and off for all columns.

Current Data Table()<<Use Value Labels (1)

Data and modeling types

For example, you can set or get the data type of a column from JSL. Choices are Character, Numeric, and Row State.

dt<<new column("new"); column("new")<<data type(character);
column("new")<<get data type;

You can set or get the modeling type of a column.

col<<Set Modeling Type("Continuous");
col<<Modeling Type("Ordinal");
col<<Get Modeling Type; // returns "Continuous", "Nominal", or "Ordinal"

You can specify the format of a column when changing its data type. For example,

column ("date") << data type(numeric, format("ddMonYYYY"));

Display formats

The Format message controls numeric and date/time formatting. The first argument is a quoted string from the list of format choices shown in the Column Info dialog. Subsequent parameters depend on the format choice. You can also set the field width by itself. Examples:

col<<Format("best",5);       // width is 5
col<<Format("Fixed Dec",9,3); // width is 9, with 3 decimal places
Manipulating columns

```plaintext
col<<Format("PValue", 6);
col<<Format("d-m-y", 10, "m-d-y"); // enter month first, display day first
col<<set fieldwidth(30);
```

For date formats, the first argument sets the way dates are displayed in a data table column, and the third argument sets the format that you use for entering data or for displaying the current cell when you have it selected for entry or editing. The date/time format choices are further discussed under “Date/time notation,” p. 81 in the “JSL Operators” chapter.

**Confusion alert!** Do not confuse the `Format` message for columns with the `Format` operator for converting numeric values to strings according to the format specified (typically used for date/time notation as described in “Date/Time Operators,” p. 79 in the “JSL Operators” chapter). Sending a message to an object has a very different effect from using a function that might happen to have the same name.

To get the current format of a row, submit a `Get Format` message.

```plaintext
col<<Get Format;
```

Roles

You can Preselect Roles. Choices are None, X, Y, Weight, and Freq. See “Objects within analysis objects,” p. 187 in the “Scripting Platforms” chapter. `Get Role` returns the current setting.

```plaintext
col<<Preselect Role(X);
col<<Get Role;
```

Formulas

Similarly, you can set, get, and evaluate a formula for a column:

```plaintext
col=New Column("Ratio");       // creates column and stores its reference
col<<Set Formula(:height/:weight); // sets formula
col<<EvalFormula;              // evaluates the formula
col<<Get Formula;              // returns the expression :height/ :weight
```

**Confusion alert!** Be sure to add commands to evaluate the formula in order to use the values from these columns in scripts. Formula evaluation timing is different in version 5 than in version 4. Read the following paragraphs carefully.

When formulas are added, they are scheduled to be evaluated in a background task. This is a problem for scripts if they depend on the column having the values while the script is running.

To force a single column to evaluate, you can send an `Eval Formula` command to the column. You can even do this inside the command to create the column, right after the Formula clause, for example:

```plaintext
dt<<NewColumn("Ratio",Numeric,Formula(:height/:weight),EvalFormula);
```

where `Formula` is an alias for `Set Formula`.

Actually, it is best to wait until you are through adding a set of formulas, and then use the command `Run Formulas` to have all the formulas evaluated in their proper order:

```plaintext
current Data Table() << Run Formulas;
```

The `Run Formulas` command is preferred to `Eval Formula`, since `Eval Formula`, while it evaluates the formulas, will not suppress the background task from evaluating them again. The formula
dependency system background task takes great care to evaluate the formulas in the right order, and
RunFormulas just calls this task until all the formulas are finished evaluating.

If you use random numbers and use the Random Reset(seed) feature to make a duplicatable
sequence, then you have another reason to use Run Formulas to avoid a second evaluation in the
background.

Range and List Checks

List and range check properties can be manipulated with JSL. The following examples set and clear
the list check property in the column Sex.

\[
\text{column("Sex") << list check({"M", "F"}); \hspace{1em} // sets it}
\]

\[
\text{column("Sex") << list check(); \hspace{1em} // clears it}
\]

Range checks require the specification of a range using the following syntax.

So, for example, to specify that the column age must be in the range 0 < age < 120, use

\[
\text{column("age") << range check(LTLT(0, 120));}
\]

Note that any of the \texttt{LXLX} operators can be preceded by \texttt{Not}, and that at most one of them can be missing.

So, to specify that the \texttt{age} column should be \geq 12, use

\[
\text{column("age") << range check(not(lt(12));}
\]

Submit an empty range check() to clear a range check state.

\[
\text{column("age") << range check();}
\]

To retrieve the list or range check assigned to a column, send a Get List Check or Get Range
Check message to the column.

\[
\text{column("sex") << get list check;}
\]

\[
\text{column("age") << get range check;}
\]

For the 0 < age < 120 example with a range check above, JMP returns

\[
\text{range check(LTLT(0, 120))}
\]

Note that you can also use Set Property, Get Property, and Delete Property to set, retrieve,
and remove List Checks and Range Checks. See “Setting, retrieving, and removing column properties.”
p. 152 for more information.

Range checks show warnings in a dialog box, which needs human interaction to continue the opera-
tion, by default. To send the warnings to the log instead, set the range check using Set Property and
add a No Warn option:

\[
\text{col<<Set Property("Range Check", LELE( 80, 140 ), No Warn);}
\]
Setting, retrieving, and removing column properties

Data columns have numerous optional metadata attributes that can be set, queried, or cleared using the messages Get Property, Set Property, and Delete Property. The name of the property in question is always the first argument for Set Property, and what is expected for subsequent arguments depends on which property you're setting. Get Property and Delete Property always take a single argument, which is the name of the property. Get Property returns the property's settings, while Delete Property completely removes the property from the column.

```
col<<Set Property( "propertyName", {argument list} );
col<<Get Property( "propertyName" );
col<<Delete Property( "propertyName" );
```

If you want to set several properties, you need to send several separate Set Property messages. You can stack several message-sends in a single JSL statement if you wish.

```
col<<Set Property( "Axis", {Min(50), Max(180)} )<<Set Property("Notes", "to get proportions");
```

To get a property's value, send a Get Property message whose argument is the name of the property you want:

```
column( "ratio" )<<Get Property("axis"); // returns axis settings
```

Choices for properties are the same as in the New Property pop-up menu in a column info dialog box, and the arguments for each correspond to the settings seen in the graphical user interface. An easy way to learn the syntax is to establish the property you want in the Column Info dialog box first, then use Get Property to view the JSL.

```
dt << Set Label Columns( col1, col2, col3 );
```
sets col1, col2, and col3 as label columns.

```
dt << Set Label Columns();
```
clears all the label columns.

The same syntax works for Set Scroll Lock Columns, and Scroll Lock.

<table>
<thead>
<tr>
<th>Usage</th>
<th>Property</th>
<th>Arguments expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard property for storing notes about a column.</td>
<td>Notes</td>
<td>Quoted text string, e.g.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Notes&quot;, &quot;Extracted from Fisher iris data&quot;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;getproperty(&quot;notes&quot;);</td>
</tr>
<tr>
<td>Standard property for prescribing the possible values that may be entered in a column</td>
<td>List Check Range Check</td>
<td>List of parameter and its values, e.g.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(List Check, {&quot;F&quot;,&quot;M&quot;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(Range Check, LTLLT(0, 120));</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Get Property(List Check);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Delete Property(Range Check);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set the range check warning to appear in the log only:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Range Check&quot;, LELE( 80, 140 ), No Warn);</td>
</tr>
</tbody>
</table>
### Table 5.2 Properties for data table columns (Continued)

<table>
<thead>
<tr>
<th>Usage</th>
<th>Property</th>
<th>Arguments expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most platforms will use this (if it exists) when constructing axes.</td>
<td>Axis</td>
<td>List of parameters and their values, mostly Booleans, e.g.</td>
</tr>
<tr>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Axis&quot;,{Min(50), Max(180), Inc(0), Minor Ticks(10), Show Major Ticks(1), Show Minor Ticks(1), Show Major Grid(0), Show Labels(1), Scale(Linear)});</td>
<td></td>
</tr>
<tr>
<td>Used for capability analysis and variability charts.</td>
<td>Spec Limits</td>
<td>List of parameters and their values, e.g.</td>
</tr>
<tr>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Spec Limits&quot;, {LSL(-1), USL(1), Target(0)}); col&lt;&lt;get property(&quot;spec limits&quot;);</td>
<td></td>
</tr>
<tr>
<td>Used for control charts.</td>
<td>Control Limits</td>
<td>Parameters and their values, e.g.</td>
</tr>
<tr>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Control Limits&quot;, {XBar(Avg(44), LCL(29), UCL(69)}); col&lt;&lt;get property(&quot;control limits&quot;);</td>
<td></td>
</tr>
<tr>
<td>Used for control charts.</td>
<td>Sigma</td>
<td>Specify known sigma value.</td>
</tr>
<tr>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Sigma&quot;,1.332); col&lt;&lt;get property(&quot;sigma&quot;);</td>
<td></td>
</tr>
<tr>
<td>Used for DOE and fitting.</td>
<td>QC Alarm Script</td>
<td>Script to run if values go out of control.</td>
</tr>
<tr>
<td></td>
<td>col&lt;&lt;Set Property(&quot;QC Alarm Script&quot;, speak(&quot;Process out of control.&quot;);</td>
<td></td>
</tr>
<tr>
<td>Used for DOE and fitting.</td>
<td>Coding</td>
<td>List with low and high values, e.g.</td>
</tr>
<tr>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Coding&quot;,{59,172}); col&lt;&lt;get property(&quot;coding&quot;);</td>
<td></td>
</tr>
<tr>
<td>Used for DOE, fitting, and profiling.</td>
<td>Mixture</td>
<td>Set Mixture to 1 to turn it on, or set Mixture to 0 to turn it off.</td>
</tr>
<tr>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Mixture&quot;,1); col&lt;&lt;Set Property(&quot;Mixture&quot;,0); col&lt;&lt;Get Property(&quot;Mixture&quot;);</td>
<td></td>
</tr>
<tr>
<td>Used for DOE.</td>
<td>Design Role</td>
<td>Specify a single role for DOE (Design of Experiments). Choices for role are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Continuous, Categorical, Blocking, Covariate, Mixture, Constant, Signal, Noise.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Design Role&quot;,Covariate); col&lt;&lt;get property(&quot;design role&quot;);</td>
</tr>
</tbody>
</table>
Table 5.2 Properties for data table columns (Continued)

<table>
<thead>
<tr>
<th>Usage</th>
<th>Property</th>
<th>Arguments expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set in DOE and used in desirability profiling.</td>
<td>Response Limits</td>
<td>List of parameters and their values. Choices for Goal are Maximize, Match Target, Minimize, None. Other parameters take numeric value and desirability arguments, e.g.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Response Limits&quot;, {Goal(Match Target), Lower(1,1), Middle(2,2), Upper(3,3)});</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Get Property(&quot;response limits&quot;);</td>
</tr>
<tr>
<td>Provided for custom uses.</td>
<td>Units</td>
<td>Specify the units of measure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;set property(&quot;units&quot;, grams);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;get property(&quot;units&quot;);</td>
</tr>
<tr>
<td>Used to set the Value Ordering column property</td>
<td>Value Ordering</td>
<td>Specify the value order:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>newVO=col&lt;&lt;set property(value ordering, {17, 15, 16, 12, 13, 14});</td>
</tr>
<tr>
<td>Provided for custom uses.</td>
<td>[Custom property]</td>
<td>Corresponds to New Property in the Column Info dialog. The first argument is a name for the custom property, and the second argument is an expression.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Date recorded&quot;,12Dec1999);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long date(col&lt;&lt;get property(&quot;Date recorded&quot;));</td>
</tr>
</tbody>
</table>

**Lock**

To lock or unlock a column, use Lock or Set Lock with a boolean argument. Get Lock returns the current setting.

```plaintext
col<<lock(1);   // lock
col<<set lock(0); // unlock
col<<get lock;   // show current state
```

**Scripts**

Get Script returns a script to create the column:

```plaintext
New Column("Ratio",SetFormula( :height/ :weight));
column("ratio")<<get script;
    New Column("Ratio", Numeric, Continuous, Formula( :height / :weight))
```

**Row State operators**

There is a special data element type called a row state for storing various attributes in the data table, such as whether a row is selected or not, excluded or not, hidden or not, and labeled or not; which
Row states can be freely converted to integers and vice versa.

**What are row states?**

Row state is a collection of six attributes that all rows in a data table have. These six attributes are actually packed into a single number internally. (How row states are defined numerically is invisible and unimportant for most users, but you can find the formal definitions under `AsRowState` if you're interested.) JSL supports row state values as a separate type in expressions. First this section shows what row states do in JMP, and then it examines in greater detail how to work with row states in scripts.

**What row states do**

Once row states are set up, they change the way JMP works with your data. The table below explains each row state separately, but keep in mind that you can also use several row states at once to get the combination of effects you want.

<table>
<thead>
<tr>
<th>Row states</th>
<th>How they affect results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excluded</td>
<td>If rows are excluded, JMP omits them from calculations for statistical analyses (text reports and charts). Results are the same as if you had never entered those rows in the table. However, points are still included in <em>plots</em>. (To omit points from plots, use <code>Hide</code>. To omit points from all results, use both <code>Exclude</code> and <code>Hide</code>.)</td>
</tr>
<tr>
<td>Hidden</td>
<td>If rows are hidden, JMP doesn't show them in plots. However, the rows are still included in text reports and <em>charts</em>. (To omit points from reports and charts, use <code>Exclude</code>.) To omit points from all results, use both <code>Exclude</code> and <code>Hide</code>.)</td>
</tr>
<tr>
<td>Labeled</td>
<td>If rows are labeled, JMP places row number labels, or the values from a designated Label column, on points in scatterplots.</td>
</tr>
<tr>
<td>Color</td>
<td>If rows have colors, JMP uses those colors to distinguish the points in scatterplots and spinning plots.</td>
</tr>
<tr>
<td>Marker</td>
<td>If rows have markers, JMP uses those markers to distinguish the points in scatterplots and spinning plots.</td>
</tr>
</tbody>
</table>
There are several ways to set row states in JMP's graphical user interface.

1. Manually: select one or more rows in the data table, then select row state choices from the pop-up menus in the Rows pane of the data table. See Figure 5.1.
2. Algebraically: set a rowstate with an expression that evaluates to rowstate number values. Create a formula variable of type Row States, and then choose Copy to Row States or Add to Row States from the pop-up menu for the column. See Figure 5.2.

The JSL equivalent would be something like this:

```
dt<<select where(row()==3|row()==6|row()==7|row()==9);
```
dt<<markers(2);

Figure 5.4 Setting row states from a formula:

The JSL equivalent would be something like this:

for each row(
  rowstate() = combine states(marker state(row()), color state(row())));

How row states work

When you give a row state operator a number as its argument, or else something that evaluates to a number, the operator interprets that number as an index to its possible values. Most of the row states—Exclude, Hide, Label, Select—are boolean, which mean they’re either on or off, 1 or 0. Marker can be 0, 1, 2, ..., 15. Color choices are 0, 1, 2, ..., 84; 0–15 for the basic colors, 16–31 for dark shades of the same colors, 32–47 for light shades, 48–63 for very dark shades, 64–79 for very light shades, and 80–84 for shades of gray.

Values outside the range 0 to 84 are undefined. For best results, use Modulo or other operators to enforce valid values.

Missing values result in no row state.
Row state columns vs. row states in effect

There are two places row state data can appear in the data table window. One is in a row state column: a column that is not numeric or character but rather row state. Row state columns store row state information but do not put the information into effect.

Here is an example to show how this works. Submit the first chunk, look at the data table, submit the second chunk, and look at the table again.

```plaintext
/* put some row states in a column */
dt=new table("Row state testing");
dt<<new column("rowstate data",rowstate,setformula(color state(row())));
```

Figure 5.5 Table With No Row States

Notice no row states are in effect.

```plaintext
/* put the row states into effect */
for each row(rowstate()=="rowstate data");
```

Figure 5.6 Table With Row States

Row states are now in effect.
This replaces the row states in effect with the row state combination from a column. “Some operators set one characteristic and clear others,” p. 161, discusses techniques for changing selected attributes of a row state without changing or canceling others.

An overview of row state operators

From JSL, you can get or set row states directly using the Row State operator. You can get or set the state for row \( n \) with Row State(\( n \)). If you don’t supply an argument, you set or get the current row.

To work with all rows, either use a For Each Row loop or else work with formula columns. Continuing the example with the data table just created:

\[
\text{rowstate}(1); // returns row state for row 1}
\text{Color State(1)}
\]

\[
\text{row()}=8;\text{rowstate(); // returns row state for current (8th) row}
\text{Color State(8)}
\]

\[
\text{for each row(print(rowstate()));}
\text{Color State(1)}
\text{Color State(2)}
\text{Color State(3)}
\text{Color State(4)}
\text{Color State(5)}
\text{Color State(6)}
\text{Color State(7)}
\text{Color State(8)}
\text{Color State(9)}
\text{Color State(10)}
\]

To get a row state and store it in a global, place Row State() on the right side of an assignment:

\[
::x=\text{rowstate}(1); // puts the row state of row 1 in x, a global
\]

\[
\text{row()}=8;::x=\text{rowstate(); // puts the row state of the 8th row in x}
\text{show(x);} \\
\text{x:Color State(8)}
\]

\[
// put row states in rscol, a row state column
\text{dt<<new column("rscol", rowstate);} \\
\text{for each row(:rscol=\text{rowstate()});}
\]

To set the current state for a row, use Row State() on the left side of an assignment.

\[
\text{row state(1)=\text{color state}(3); // makes row 1 red}
\text{row()}=8; \text{row state()=\text{color state}(3); // makes the 8th row red}
\text{for each row(\text{row state()=\text{color state}(3); // makes each row red}
\]

Be careful whether you set every aspect of Row State() or just one aspect of it, such as Color Of(Row State()). To see how this works, first color and mark all the rows:

\[
\text{for each row(}
\text{rowstate()=combine states(\text{color state(row()), marker state(row())});}
\]

And now observe the difference between setting one attribute of a rowstate...

\[
\text{color of(rowstate(1))=3; // makes row 1 red without changing marker}
\]
...and setting every aspect of a rowstate to a single state:

```plaintext
row state(1)=color state(5); // makes row 1 blue and loses its marker
```

To copy all the current row states into a row state column:

```plaintext
new column("rscol", set formula(rowstate());
for each row(rscol = rowstate());
```

To copy several but not all the current row states into a row state column, you would use a script like the following, commenting out or omitting any states you don’t want.

```plaintext
new column("rscol2", set formula(CombineStates(
  ColorState( color of() ),
  ExcludedState( excluded() ),
  HiddenState( hidden() ),
  LabeledState( labeled() ),
  MarkerState( marker of() ),
  SelectedState( selected() ))));
```

Confusion alert! The current row for scripting is not related to rows being selected (highlighted) in the data table or to the current cursor position in the data table window. The current row for scripting is defined to be zero (no row) by default. You can set a current row with `Row`, e.g. `Row()=3`, but please note that such a setting only lasts for the duration of that script, and then `Row()` reverts to its default value, zero—so submitting a script all at once can produce different results than submitting a script a few lines at a time.

Another way to establish a current row for a script is to enclose it in `ForEachRow`, which executes the script once for each row of the current data table.

Throughout this chapter, examples without an explicit current row, such as below, should be assumed to take place within a context that establishes a current row, e.g. inside `For Each Row()` or after `Row()=1`.

```plaintext
rowstate()=color state(3);
```

See “What is the current row?”, p. 127, for more information.

**Combinations of characteristics**

You can get or set many characteristics at once by combining state settings inside `Combine States`, or you can get or set each characteristic one at a time, with the ultimate row state being the accumulation of characteristics. For example, with `Big Class` you can set green Y markers for males but for now hide them in plots, and set red X markers for females and not hide them in plots:

```plaintext
for each row(
  if(sex=="M",
    /*then*/ row state()=combine states(
      color state(4), marker state(6), hidden state(1)),
    /*else*/ row state()=combine states(
      color state(3), marker state(2), hidden state(0))));
```

Get the row state for one row, such as the 6th:

```plaintext
row state(6);
  Combine States(Hidden State(1), Color State(4), Marker State(6))
```
Notice that JMP returns a Combine State combination. This is because a row state datum is not just the state of one characteristic, such as color, but the cumulative state of all the characteristics that have been set: exclusion, hiding, labeling, selection, markers, colors, hues, and shades. A list of such characteristics is called a row state combination.

Just as there can be many row state characteristics in effect, a row state column can have multiple-characteristic row states as its values.

**Some operators set one characteristic and clear others**

In addition to the overall Row State operator for getting or setting all the characteristics of a row state, there are separate operators to get or set one characteristic at a time preemptively—that is, to give a row one characteristic, cancelling any other characteristics that might be in effect. These operators are Color State, Combine States, Excluded State, Hidden State, Hue State, Labeled State, Marker State, Selected State, Shade State.

For example, to make row 4 be hidden only:

```
rowstate(4)=hidden state(1);
```

**Other operators set or get one characteristic at a time**

As seen above, a row state is not just one characteristic but many. To work with just one thing at a time, use one of the L-value operators with Row State on either side of the equals sign, depending on whether you want to get or set. There is an L-value operator for each of the characteristics: Color Of, Excluded, Hidden, Labeled, Marker Of, Selected.

```
// hide row 4 without affecting other characteristics:
hidden(rowstate(4))=1;

// store color of row 3 without getting other characteristics:
::color = color of(rowstate(3));
```

**Get row states**

To access a row state, place the row state expression on the right side of an assignment:

```
x = rowstate(i);  // puts row state of row i into x
x = rowstate();   // row state of current row
```

To access just a component of a row state, place the row state expression on the right side of an assignment and also use one of the L-value operators:

```
x = selected(rowState()); // selection index of current row selected
```

**Set row states**

To set a row state, place the row state expression on the left of an assignment (L-value):

```
row state() = expression;     // for the current row
row state(i) = expression;    // for a specified row
```

To set a component of a row state,

```
ColorOf(row state(i)) = 3;   // change color to red for row i
Selected(RowState(i)) = 1;   // select the ith row
```
The following are equivalent to their interactive commands.

Copy From Row States
Add From Row States
Copy To Row States
Add To Row States

Set row state combinations

Use Combine States to put together the settings of the various States:

\[
x = \text{CombineStates}(\text{SelectedState}(1), \text{LabeledState}(0), \text{ColorState}(3));
\]

for each row(rowstate()=x);

Create row state columns

To create a row state directly in a column, use a formula:

\[
dt<<\text{new column}("\text{color formula}", \text{row states}, \text{set formula}(\text{color state(row())}));
\]

To put row states from a column into effect:

for each row(rowstate()=:\text{color formula});

Make a row state handler

The MakeRowStateHandler message (sent to a data table object) obtains a callback when the row-states change. For example,

\[
f = \text{Function}(\{X\}, \text{Show}(\ x\ ));
\]

\[
\text{obj} = \text{Current Data Table()} << \text{MakeRowStateHandler}(\ f\ );
\]

Then when you select a group of rows, the row numbers of any row whose row state changed are sent to the log. For example:

\[
x: [3, 4, 28, 40, 41]
\]
When a group is highlighted, it may call the handler twice, once for rows whose selection is cleared, then again for the new selection.

Each of the row state operators in detail

Here is a comparison chart of the different row state operators, so you can see which to use to convert from row states to indices and vice versa.

Table 5.4 Operators for converting between numbers and row states

<table>
<thead>
<tr>
<th>Numbers</th>
<th>Convert from numbers to row states</th>
<th>Row states</th>
<th>Convert from row states to numbers</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 or 0</td>
<td>Excluded State((n))</td>
<td>Excluded</td>
<td>Excluded((\text{rowstate}))</td>
<td>1 or 0</td>
</tr>
<tr>
<td>1 or 0</td>
<td>Hidden State((n))</td>
<td>Hidden</td>
<td>Hidden((\text{rowstate}))</td>
<td>1 or 0</td>
</tr>
<tr>
<td>1 or 0</td>
<td>Labeled State((n))</td>
<td>Labeled</td>
<td>Labeled((\text{rowstate}))</td>
<td>1 or 0</td>
</tr>
<tr>
<td>1 or 0</td>
<td>Selected State((n))</td>
<td>Selected</td>
<td>Selected((\text{rowstate}))</td>
<td>1 or 0</td>
</tr>
<tr>
<td>0 to 31</td>
<td>Marker State((n))</td>
<td>Marker</td>
<td>Marker Of((\text{rowstate}))</td>
<td>0 to 31</td>
</tr>
</tbody>
</table>
Excluding, hiding, labeling, and selecting

This section discusses the Boolean states—conditions which are either on or off. The next section looks at the Color, Marker, Hue, and Shade states—conditions which have many choices.

Excluded gets or sets an excluded index, which is 1 or 0 (true or false) indicating whether each row is excluded. Similarly, Hidden gets or sets a hidden index, which is 1 for hidden or 0 for not hidden; Labeled gets or sets a labeled index, which is 1 for labeled or 0 for not labeled; and Selected gets or sets a selected index, which is 1 for selected or 0 for not selected. For example:

```cpp
Excluded(Row State()); // returns 1 if current row is excluded, 0 if not
Hidden();              // returns 1 if current row is hidden, 0 if not
Labeled(Row State());  // returns 1 if current row is labeled, 0 if not
Selected();            // returns 1 if current row is selected, 0 if not
```

```cpp
Excluded(Row State())=1; // excludes current row
Hidden()=0;              // unhides current row
```
Row State operators

Labeled(Row State())=1;  // labels current row
Selected()=0;            // unselects current row

Recall that these functions assume the argument Row State() if none is given.

Excluded State, Hidden State, Labeled State, and Selected State do the reverse; that is, they get or set a row state condition as true or false according to the argument. Nonzero values set the row state to true, and zero values set it to false. Missing values result in no change of state.

rowstate()=excluded state(1); // changes current row state to excluded
rowstate()=hidden state(0);  // changes current row state to not hidden
rowstate()=labeled state(1); // changes current row state to labeled
rowstate()=selected state(0); // changes current row state to not selected

Notice that the last two commands replace the row state with just the exclusion or just the unhiding, so that any preexisting row state characteristics are lost. More commonly you would issue the State commands for all the characteristics you want inside a Combine States:

rowstate()=combine states(color state(4), marker state(3), hidden state(1));
// changes the current row to hidden and a green square marker

Another common way to use a -State command would be in a row state data column whose values could be added to the row state (for cumulative characteristics). This example excludes each odd numbered row.

dt<<new column("myExcl",rowstate, set formula(Excluded State(Modulo(Row(),2))));
for each row(rowstate()=combine states(:myExcl, rowstate()));

Colors and markers

Color Of returns or sets the color index, which is a number from the JMP color map that corresponds to the rowstate, or a missing value if there is no assigned color.

ColorOf(rowstate());  // returns color index for current row
ColorOf()=4;           // sets current row to Color 4

Similarly, Marker Of returns or sets the marker index, which is a number from the JMP marker map that corresponds to the active marker, or a missing value if there is no assigned marker.

MarkerOf();             // returns marker index for current row
MarkerOf(rowstate())=4; // sets current row to Marker 4

Both Color Of and Marker Of accept any row state expression or column or Row State() as arguments and assume the argument Row State() if none is given (some examples are shown with and some without).

Color State and Marker State are similar to Color Of and Marker Of, except they work in the opposite direction. Where the -Of functions turn actual states into indices, the -State functions turn indices into states.

color state(rowstate()); // returns color state for current row
marker state();          // returns marker state for current row
rowstate()=color state(4); // changes current row to green
rowstate()=marker state(4); // changes current row to the diamond marker
Notice that the last two commands replace the row state with just the color or just the marker, so that any preexisting row state characteristics are lost. More commonly you would issue the `-State` commands for all the characteristics you want inside a `Combine States`:

```
rowstate()=combine states(color state(4), marker state(3), hidden state(1));
// changes current row to hidden with green square marker
```

This script (which you’ll learn how to create in the “Display Trees” chapter) shows the standard JMP markers, which are numbered 0–31. Indices outside the range 0–31 have undefined behavior.

```
New Window( "Markers",
  Graph Box(  
    FrameSize( 300, 300 ),
    Y Scale( -1, 16 ),
    X Scale( 0, 7 ),
    For(  
      i=0; jj=15, i<16; jj>=0, i++; jj--, // 16 rows, 2 columns
        Marker Size( 3 );
        Marker( i, {1, jj + .2} ); // markers 0-15
        Marker( i + 16, {4, jj + .2} ); // markers 16-31
        Text( {1.5, jj}, "marker ", i ); // marker labels 0-15
        Text( {4.5, jj}, "marker ", i + 16 ); // marker labels 16-31
    ));
)
```

Figure 5.7 JMP Markers

JMP colors are numbered 0 through 84, where the first 16 are the basic colors shown below, and higher numbers are darker or lighter shades of those colors. Indices outside the range 0–84 have undefined behavior. This script shows the standard JMP colors. The first 16 colors also have names. The rest are
shades of those 16. For more information on using JMP colors, see “Colors,” p. 298 in the “Scripting Graphs” chapter.

Text Color( 0 );
New Window( "Colors",
Graph Box(
    FrameSize( 640, 400 ), Y Scale( -1, 17 ), X Scale( -3, 12 ),
k = 0;
    For( jj = 1, jj <= 12, jj += 2,
        l = 15;
        For( i = 0, i <= 15 & k < 85, i++,
            thiscolor = Color To RGB( k );
            Fill Color( k );
            thisfill = 1;
            If( thiscolor == {1, 1, 1},
                Pen Color( 0 ); thisfill = 0,
                Pen Color( k ) );
            Rect( jj, l + .5, jj + .5, l, thisfill );
            Text( {jj - 1, l}, "color ", k );
            k++; l-- );
        jj = -2;
        color = {"Black", "Gray", "White", "Red", "Green",
            "Blue", "Orange", "BlueGreen", "Purple", "Yellow", "Cyan",
            "Magenta", "YellowGreen", "BlueCyan", "Fuschia", "Black"};
        For( i = 0; l = 15, i <= 15 & l >=0, i++; l--,
            Text( {jj, l}, color[ i + 1 ] )));
);
If you prefer to use RGB values, each color should be a list with percentages for each color in red, green, blue order.

```markdown
pen color([.38,.84,.67]); // a lovely teal
```

**Hue and shade**

Hue State and Shade State together are an alternative to Color State for picking colors. You cannot select black, white, or the shades of gray when you use Hue State—for these, you must use Shade State alone, or Color State.

The script below demonstrates how hue and shade values relate to colors:

```markdown
new window("Hues and Shades",
    graphbox(FrameSize(600,300), yscale(-3,3), xscale(-2,12),
    k=0;
    for(h=0,h<12,h++,
        for(s=-2,s<3,s++,
            myMk=combine states(hue state(h),shade state(s),marker state(15));
            markersize(3);
            marker(myMk, {h,s}))));
    text(Center Justified, {5,2.5}, "<--- Hues 0-11 ---> ");
    text(Right Justified, {-5,-2}, "Shade -2",{-5,-2.25},"(Very dark)",
        {-5,-1},"Shade -1",{-5,-1.25},"(Dark)",
        {-5,0},"Shade 0",{-5,-.25},"(Basic hue)",
        {-5,1},"Shade 1",{-5,.75},"(Light)",
        {-5,2},"Shade 2",{-5,1.75},"(Very light)");
```

Figure 5.9 Hues and Shades

Note: there are no “-Of” operators for Hue and Shade. Color Of returns the equivalent Color State index for a color row state that has been set with Hue State and/or Shade State. The following example gives rows 4 and 5 the same dark red marker:

```markdown
rowstate(4)=combine states(hue state(0),shade state(-1),
    marker state(12));
```
Row State operators

rowstate(5) = combine states(color state(color of(rowstate(4))),
marker state(marker of(rowstate(4))));

QC chart example

In the example below (which assumes you have CP and CA data), row state values are prepared ahead and passed to the Marker routine, along with matrices of coordinates.

```julia
// assumes CP and CA data such as simulated below
dt = New Table( "Artificial CP and CA data",
    Add Rows( 26 ),
    New Column( "cover_cp", Numeric, Continuous, Formula( Random Uniform() / 100 + 0.94 ) ),
    New Column( "cover_ca", Numeric, Continuous, Formula( Random Uniform() * 0.04 + 0.94 ) ),
    New Column( "p", Numeric, Continuous, Formula( Random Uniform() ) ) );
dt << Run Formulas;
greenMark = Combine States( Marker State( 2 ), Color State( 4 ) );
redDiamond = Combine States( Marker State( 3 ), Color State( 3 ) );
New Window( "CP and CA Comparisons",
    Graph Box(
        title( "CP and CA Comparison" ),
        FrameSize( 400, 350 ),
        X Scale( 0, 1 ),
        Y Scale( 0.94, 1 ),
        For Each Row( Marker( greenMark, {p, cover_cp} );
            Marker( redDiamond, {p, cover_ca} ) );
    ) );
```

Optional: the numbers behind row states

This section is an optional topic for advanced users who are interested in working with row states through their internal numeric codes.

Earlier the chapter mentioned that all six row states are actually stored as a single number in JMP. You can see row states' internal coding if you want. Simply copy row states to a column, and then change the column's type to numeric, and you will see the numbers JMP uses.

It is also possible to assign row states through their internal numeric codes using the As Row State operator, which simply converts integers to their equivalent row states. For example, to assign row states according to the row number, you could do:

```julia
for each row(rowstate() = as row state(row()));
```

In addition, the Set Row States command allows you to submit a matrix of codes that assign the row states all at once. The matrix should have dimension (number of rows) by 1, and contain one entry for each row. The entries are the row state codes corresponding to the row's desired state.

Such row states are unlikely to be of any use, however. For practical applications, understanding the way numbers are related to row states is important. Briefly, for some row state \( r \), such as the row state of the 3rd row as shown here, the row state code is computed by this formula:
Calculations

This section discusses functions for pre-evaluated columnwise and rowwise statistics and shows how JSL expressions work behind the scenes in the JMP formula calculator. See the earlier discussion of “Summarize,” p. 129, for a way to gather summary statistics on a data table.

Pre-evaluated statistics

JMP has Col Maximum, Col Mean, Col Minimum, Col N Missing, Col Number, Col Quantile, CV (Coefficient of Variation), Col Standardize, Col Std Dev, Col Sum, Maximum, Mean, Minimum, N Missing, Number, Std Dev, and Sum functions, which are special “pre-evaluated” functions.

Confusion alert! Statistics are also computed with Summarize (p. 129). Although the named arguments to Summarize have the same names as these pre-evaluated statistic functions, they are not calling the pre-evaluated statistic functions. The resemblance is purely coincidental.
All the statistics are pre-evaluated; that is, JMP calculates them once over the rows or columns specified and thereafter uses the results as constants. Because they are computed once and then used over and over again, they are more efficient to use in calculations than an equivalent formula-calculated result.

When JMP encounters a pre-evaluated function in a script, it immediately evaluates the function and then uses the result as a constant thereafter. Thus, pre-evaluated functions enable you to use columnwise results for rowwise calculations. For example, if you use Col Mean inside a column formula, it first evaluates the mean for the column specified and then uses that result as a constant in evaluating the rest of the formula on each row. For example, a formula might standardize a column using its pre-evaluated mean and standard deviation:

\[(\text{Height} - \text{Col Mean(Height)}) / \text{Col Std Dev(Height)}\]

For the Big Class data set, Col Mean(Height) is 62.55 and Col Std Dev(Height) is 4.24. So for each row, the formula above would subtract 62.55 from that row's height value and then divide by 4.24.

Note: the pre-evaluated functions disregard the excluded rowstate, so any excluded rows are included in calculations. For summary statistics that obey row exclusion, use the Distribution platform.

**Columnwise functions**

The functions whose names begin with "Col" all work columnwise, or down the values in the specified column, and return a single number. For instance, Col Mean(height) finds the mean of the values in all the rows of the column height and returns it as a scalar result. Some examples:

- \[\text{Average Student Height} = \text{Col Mean(height)};\]
- \[\text{Height Sigma} = \text{Col Std Dev(height)};\]

**Rowwise functions**

The functions without "Col" listed below work rowwise across the values in the variables specified and return a column result. For instance, Mean(height, weight) finds the mean of the height and weight for the current row of the data table. The rowwise statistics are only valid when used in an appropriate data table row context. Following are some possibilities:

- \[\text{// scalar result for row 7 assigned to JSL global variable}\]
  \[\text{row()}=7; ::\text{scalar} = \text{Mean(height, weight)};\]

- \[\text{// formula column created in data table}\]
  \[\text{new column("Scaled Ht-Wt Ratio",}\]
  \[\text{formula(mean(height, weight)/age));}\]

- \[\text{// vector of results}\]
  \[\text{vector}=J(1,40); // create a 1x40 matrix to hold results}\]
  \[\text{for each row(vector[row()]=mean(height,weight)); //fill vector}\]

Rowwise functions can also take vector (column matrix) or list arguments:

- \[\text{myMu=mean([1 2 3 4]); mySigma=stddev([1, 2, 3]);}\]
Calculator formulas

JMP allows you to store formulas in columns which are automatically evaluated to create the values in the cells of the column. If you open the formula, you get a calculator interface to edit the formula structurally, as shown below on the left.

However, the formula is implemented with JSL, and you can obtain the text JSL form of any expression in the calculator by double clicking on it. The text can be edited, and when it is defocused, it is compiled back into the structural form.

Figure 5.10 Viewing a Calculator Formula as JSL.

There is no difference between a formula column created through the calculator window and one created directly through JSL with commands such as `New Column(..., Formula(...)`) or `Col<<Formula(...)`.

For details on calculator functions, see the reference tables in “JSL Operators” chapter.
Chapter 6
Scripting Platforms
Create, repeat, and modify analyses

Platforms can be launched from scripts and subsequently controlled from scripts. If you do an analysis interactively, you can save a script that will recreate the analysis.

Learning how to script platforms is no harder than learning how to control platforms interactively, but actually writing scripts yourself takes some effort, since you do have the burden of typing them and getting the syntax right. Fortunately, you can have the platform write much of a script for you, and the additional work is well worth it if you need to automate repetitive production analyses. Also, scripting presents valuable opportunities to customize and combine analyses.

Be aware that the script-saving feature of platforms currently has limitations. JMP saves analytical commands and most (but not all) presentation customizations. Therefore, if your presentation customizations are important to you, you may occasionally need to learn to program the display interface. Also, JMP records the state of an analysis, but not the sequence of events leading to that state, so if you want scripts to play back demonstrations of JMP platforms in action, you have some learning to do.

Platform results exist in two layers: the platform itself, containing the analytical results and responding to analytical commands; and the presentation display, which responds to a different set of commands. A third object is the data table itself, which can be involved in a live analysis.

1. If you want to add to the analysis, such as to fit a new line, you send a command to the platform itself using techniques discussed in this chapter.
2. If you want to make the frame of a graph larger, you send a command to the display. See “Manipulating displays,” p. 227 in the “Display Trees” chapter.
3. If you want to highlight certain points representing rows in a data table, you send a row state command to the data table. See “Row State operators,” p. 154 in the “Data Tables” chapter.

If you are interested in building your own custom analysis platforms and reports, see “Constructing display trees,” p. 240 in the “Display Trees” chapter. If you would like to build your own custom statistical calculations using JSL’s compact matrix notation, see the “Matrices” chapter.
Contents

Scripting analysis platforms .............................................................. 175
Launching platforms interactively and obtaining the equivalent script .......... 175
Sending script commands to a live analysis ...................................... 182
Launching platforms ........................................................................ 186
Objects within analysis objects ............................................................. 187
General messages for platform windows ............................................. 187
Scripting specific platforms ................................................................. 190
Scripting analysis platforms

Scripting platforms is done with the same keywords you see in the dialogs and menus, including the choices you see in popup menus and context menus inside a report.

You can use JMP interactively and then obtain the script corresponding to the analysis you created. You can take advantage of this behavior in several ways:

1. Learn JSL by example. Launch platforms and work with them interactively, then save and examine the script that matches your work. Make changes through the interface and then study the changes in the script.
2. Save a number of scripts from analyses to a script window, then save it to a file so that you can recreate the analysis later, or reproduce the analysis on new data.
3. Save platform scripts, then edit them to use JSL programming features to make the analyses more general and to customize reports.

Notation in this chapter

This chapter Capitalizes The Names of command words that you need to use exactly as they are shown and shows arguments that are placeholders for actual choices in lowercase. For example, here Connect Color is a command that you need to type as is, and color stands for some color choice that you will make yourself.

Connect Color(color)

In this case, the argument in parentheses must be some color value—which happens to be a number from 0 to 65, or a supported color name like "red", "blue", etc., or an RGB value given as a list, such as {.75, .50, .50}. Sometimes alternatives like these are shown with the | character for "or," like this:

Connect Color( number | "color name" | {r,g,b} );

You don't actually have to use upper- and lower-case letters, and you only type the | character in scripts when you want JMP's logical OR operator |

Launching platforms interactively and obtaining the equivalent script

Here is an example to see how JMP's interface and scripting language are related.

Launch a platform

1. Open the Big Class sample data table.
2. Select Oneway from JMP Starter, or Fit Y by X from the Analyze menu.
3. Cast height into the Y role and age into the X role.
4. Click OK.
Save Script

All analysis platforms have a Script submenu with the commands shown here to create a JSL script that will duplicate the analysis in its current state. These commands are choices for where to send that script.

Try the choices yourself:

**Redo Analysis**  Launches a new copy of the platform, rebuilding all the analyses as specified. This is useful for updating your results when the state of the data table has changed (e.g., after you fix errors in the data table, select a subset, add more data, etc.).
**Save Script to Script Window**  Presents the script for the current state of the platform in a script window, where you can view, edit, and submit it.

**Figure 6.3  Saving a Script to a Script Window**

![Script Window Image]

**Save Script for All Objects**  Saves the script for all objects within a multi-platform report in a script window, where you can view, edit, and submit it. An example would be Fit Y by X, where various combinations of continuous, nominal, and ordinal columns result in Bivariate, Oneway, Contingency, and Logistic platform objects all within a single report. **Save Script to Script Window** saves only the script for the single object from which you selected the pop-up menu item, but **Save Script for All Objects** saves the script for all objects within the window—regardless of which object's pop-up menu you use. Another example is that if you have By groups, **Save Script for All Objects** saves the script for all the By-group objects within the window, whereas **Save Script to...** captures the script for only the one group. All the by-groups appear in the same window when the report is re-run, because **Save Script For All Objects** wraps all By-variable analyses in a **New Window()** command. For example, a bivariate fit of height by weight grouped by males and females would result in the following script.

```julia
New Window("Big Class.jmp: Bivariate",
Bivariate(Y( :height), X( :weight), Fit Line, Where( :sex == "F"));
Bivariate(Y( :height), X( :weight), Fit Line, Where( :sex == "M")));
```

**Save Script to Datatable**  Saves the script as a new property in the data table. In the data table window, you see the script named for the platform, and it has a pop-up menu to let you **Run Script**. Properties such as saved scripts are saved with the data table for later use. Notice that most of JMP’s sample data tables have script properties to run examples shown in the documentation.

**Figure 6.4  Saving a Script to a Data Table**

![Datatable Image]

**Save Script to Report**  Saves the script in a text box at the top of the report. The script to reconstruct an analysis is thus available in any journals you might save, as well as in the platform window.
dow itself. This also serves as a concise overview, listing all the report’s analyses and their parameter settings.

Figure 6.5 Saving a Script to a Report

Data Table Window Displays a data table window with the By-group data associated with the analysis when it has By groups, and brings the data table window forward otherwise.

In addition, the following commands are legal, but not on the menu.

any displaybox command Commands that are not recognized as platform commands are passed to the displaybox tree to see if they are recognized there.

Report Returns a reference to the displaybox tree.

TopReport Like Report, returns a displayBox reference, but goes to the top of the report tree and returns that. For simple platform calls, this is the same as <<report, but for By groups and specialty windows, it may not be.

Make some changes

Now, continue working through the interface. Pop-up menus have commands for expanding the report:
Figure 6.6 Adding Elements to a Report Interactively

If you save to the script window again, the resulting script will be:

```
Oneway(Y(height), X(age),
     Quantiles(1), Means(1), UnEqual Variances(1),
     Box Plots(1), Mean Diamonds(1), Std Dev Lines(1));
```

The script records the choices you made in the menu. Since they are boolean (on or off) options, they have an argument of 1 to turn them on. In addition to the options selected, three additional display options have been implied from the other commands given. Choices you make in pop-up menus and their corresponding JSL commands have exactly the same effects.

You could submit this script to get exactly this report quickly, without all the dialog box and pop-up menu steps. To run a script from a script window, select the text and then either select Edit–>Run Script or type Control-R (Windows) or ⌘-R (Macintosh). If you select no text, the entire window is run.

**Syntax for platform scripting**

Look at the script in more detail.

```
Oneway(Y(height), X(age),
     Quantiles(1), Means(1), UnEqual Variances(1),
     Box Plots(1), Mean Diamonds(1), Std Dev Lines(1));
```

All platform scripts start with a command to call the platform, in this case `Oneway`. Inside the `Oneway` command are two kinds of arguments: arguments like the Y and X column role lists are required at launch, and options like `Quantiles(1)` are sent to the platform after it is launched.

Most options are simple on/off choices with check marks (or not). The scripting equivalent of a check mark or not is a boolean argument 1 or 0.
Other commands lead to dialog boxes where you specify values or make choices. In scripts, such specifications would be given inside parentheses and separated by commas, usually in the same order as they appear in the dialog box (top to bottom, left to right).

**By-group reports**

In most platforms, you can run the platform repeatedly across subgroups of the rows as defined by one or more by columns. To do this in scripts, include a `By` argument in the launch command, listing each column as the argument to `By`. For example:

```plaintext
biv = Bivariate(Y(weight), X(height), By(sex));
```

This launch message would produce a report window with two nodes in the outline, one for the rows of the data table where sex is “F” and a second for rows where sex is “M.” Rather than returning a reference to a platform, `Bivariate[]`, it would return a list of references `{Bivariate[], Bivariate[]}`. You can direct messages either to each reference individually or to the list of references, depending on whether you want to change selected nodes individually or all nodes simultaneously.

```plaintext
show(biv);

biv:{Bivariate[], Bivariate[]}
biv << fit line; //adds regression fit to both nodes
Bivariate[1] << fit polynomial(3); //adds cubic fit to the F node
Bivariate[2] << fit polynomial(4); //adds quartic fit to the M node
```
Multiple columns listed for By() produce nodes for each subgroup. For example, By(sex, age) would produce nodes for females age 12, females age 13, ..., females age 17, males age 12, males age 13, ..., and males age 17.

The following shows how to launch a platform with By groups and extract results from each group:

```plaintext
// open data table Big Class
dt=Open("$SAMPLE_DATA/Big Class.jmp");

// launch Oneway platform
onew = oneway(x(age), y(height), by(sex), anova);
  // onew will be a list of platform object refs
r = onew<<report;  // r will be a list of displayBoxes
nBy = nItems(r);  // the number of by groups
vc = j(nBy,1,0);  // a place to store the variances

// now extract results
```
Scripting analysis platforms

for(i=1, i<=nBy, i++,
    vc[i] = r[i][
        OutlineBox("Analysis of Variance"),
        ColumnBox("Sum of Squares")][2]);
show(vc);

summarize(byValues=by(sex));
newTable("Variances")
  << newColumn("Sex",character,width(8),values(byValues))
  << newColumn("Variance",numeric,continuous,values(vc));

Saving By-Group scripts

In addition to the Script submenu in an analysis platform, a Script All By-Groups submenu appears when appropriate. This submenu lets you save scripts to reproduce a report created with By groups. It contains the following options:

- Redo Analysis
- Relaunch Analysis
- Copy Script
- Save Script to Data Table
- Save Script to Journal
- Save Script to Script Window

All of these work just like their counterparts on the Script submenu, except they reproduce all By-groups in a report.

Sending script commands to a live analysis

After the platform has been launched, there is a different syntax to send messages to control the live platform, using the Send function or its operator equivalent, <<.

First you need a way to address the object. The simplest way is to launch the platform from a script which assigns a reference to the object to a global variable. For example, this script saves a reference to a oneway analysis in the variable oneObj:

    oneObj = Oneway(Y(height), X(age));

Another way is to make an assignment by referring to the platform name with a subscript. This is the method to use for a platform that has already been launched, such as from a dialog instead of a script. If there is only one current, live instance of the platform, this is straightforward: the subscript is 1:

    oneObj = Oneway[1];

Then, you use either the Send function or the equivalent << operator to send a message:

    Send(oneObj, Unequal Variances(1));
    oneObj << Unequal Variances(1);
Conventions for commands and arguments

1. You can omit the argument for a boolean option to toggle the state: if the state is off, the message turns it on; if it is on, the message turns it off. In the example above, these following two commands would be equivalent. If you resubmit the first command repeatedly, it has no further effect, but if you submit the second command repeatedly, it flips the feature on and off.

   ```
   oneObj << Unequal Variances(1);
   oneObj << Unequal Variances;
   ```

2. In cases where the menu gives several options separated by comma or slash, as in the case of Means/Anova_t Test above, you can use any one of the commands. In cases where several commands have the same alias, as in the two cases of “Means” above, the first one takes precedence in the scripting language.

3. If you make changes to the display, such as resizing the graph, they are also saved in the saved script.

4. If there are submenus whose items represent commands rather than settings, then the corresponding script is the items themselves without the parent item. For example, above see that Oneway has a menu item Nonparametric with three commands in a submenu, including Wilcoxon Test. You would use just the subitem name in scripts, e.g.:

   ```
   oneObj = Oneway(Y(height), X(age), Wilcoxon Test(1));
   oneObj << Wilcoxon Test(1);
   ```

5. If there are submenus whose items are values for a setting rather than independent commands, then in script, you give the parent item with the submenu choice as its argument. For example, Oneway has a submenu for Set Alpha Level, whose choices are .10, .05, .01, and Other... To change the value in script, you give your choice as an argument to Set Alpha Level:

   ```
   oneObj << SetAlphaLevel(.01);
   ```

6. The script returned from a platform will often look different from a script that you write yourself. For example, you could launch Distribution with this brief script:

   ```
   dist = Distribution(Y(Height, weight));
   ```

   But if you then ask Distribution to save its script, you see considerably more detail:
Scripting analysis platforms

Sending several messages

To send several messages you can add more << operators or more Send arguments:

\[
\text{dist}<<\text{quantiles}(1)<<\text{moments}(1)<<\text{more moments}(1)<<\text{horizontal layout}(1);
\]

\[
\text{Send}(\text{dist}, \text{quantiles}(1), \text{moments}(1), \text{more moments}(1), \text{horizontal layout}(1));
\]

Because << is an eliding operator, it combines arguments and works differently than if its arguments were grouped. You can stack up multiple messages with extra << symbols to perform them all in order (left to right). You can use grouping parentheses to send a message to the result of sending a message:

\[
(\text{dist}<<\text{stem and leaf}(1)) << \text{horizontal layout}(0);
\]

In this case, the associative grouping is of no consequence, because messages are performed left-to-right anyway. However, a case where it would matter is when sending messages to child objects, discussed next.

Another way to stack messages is to send a list of messages:

\[
\text{dist}<<\{\text{quantiles}(1), \text{moments}(1), \text{more moments}(1), \text{horizontal layout}(1)\};
\]

Learning the messages an object responds to

Now that you have an object, what messages can you send it? There are several ways to learn your options:

1. Try the procedure through the interactive interface first, then study the saved script.
2. Study the interface in the platform window. Items in the pop-up menus and context menus all have JSL equivalents with the same names and arguments, as discussed under “Launching platforms interactively and obtaining the equivalent script,” p. 175.
3. Go to the Help menu, select Indexes > Object Scripting, find the object type you are interested in, and click the item in the list.
4. Show Properties(objectRef) lists to the Log window all messages the object can receive:

\[
\text{show properties(oneObj)};
\]

- \text{Quantiles} [Boolean]
- \text{Means} [Boolean]
- \text{Means and Std Dev} [Boolean]
- \text{Compare Means} [Subtable]
  - Each Pair, Student's t [Boolean]
  - All Pairs, Tukey HSD [Boolean]
  - With Best, Hsu MCB [Boolean]
  - With Control [Boolean](!({level value}))
- \text{Nonparametric} [Subtable]
  - Wilcoxon Test [Boolean]
  - Median Test [Boolean]
  - van der Waerden Test [Boolean]
- \text{UnEqual Variances} [Boolean]
- \text{Power...} [Action]
Scripting Platforms

Chapter 6

Set Alpha Level [ActionChoice] {.10,.05,.01,Other...}
...etc...

Show Properties also works with data tables and display boxes; see “Learning about data tables’ messages,” p. 121 in the “Data Tables” chapter and “Learning what you can do with a display box,” p. 232 in the “Display Trees” chapter.

How to interpret the listing from Show Properties

Notice that most items in the Show Properties output have hints inside brackets [ ] at the end of each line. This section examines the Show Properties for Bivariate as a general example.

```
biv=Bivariate(Y(height), X(age));
show properties(biv);
```

(Column Names) properties are at the bottom. For example, X, Y, Freq, and Weight are things that need to be declared when you create a Bivariate analysis, and they cannot be set or changed in an existing analysis.

```
X, Regressor [Column Names] [Scripting Only] (1UNC)
Y, Response [Column Names] [Scripting Only] (1UNC)
Freq [Column Names] [Scripting Only] (01N)
Weight [Column Names] [Scripting Only] (01N)
```

(Subtable)s refer to a set of commands that are put in a submenu. The commands in the subtable are indented, and you use the subitem itself, not the parent item.

```
Script [Subtable]
  Redo Analysis [Action]
  Save Script to Datatable [Action]
  Save Script to Report [Action]
```

(Boolean)s turn an option on or off, and their arguments are usually 1 or 0. If specified without an argument, sending the message will flip it to the opposite state. Frequently [Boolean] messages also indicate that they by [Default On].

```
Show Points [Boolean] [Default On]
```

[Action]s and [New Entity]s are general purpose commands, often leading to dialogs in the user interface. Actions don’t have a specific standard for their arguments, so try the item in the interface first and then study the script that the platform saves.

[Action Choice]s and [Enums]s have a specific set of choices for their arguments.

```
Fit Mean [New Entity]
Fit Line [New Entity]
Fit Polynomial [ActionChoice] {1,2,quadratic,3,cubic,4,quartic,5,6}
```

Confusion alert! Do not confuse a reference to a platform with a reference to a report. They are different types of objects and can receive different types of JSL messages. For example, platforms can do such things as run tests, draw plots, or close entire windows. Reports can do such things as copy pictures, select display boxes, or close outline nodes.

This chapter discusses how to script platforms. To learn how to script reports, see the section “Manipulating displays,” p. 227 in the “Display Trees” chapter.
Launching platforms

Specifying columns

Scripts to launch a platform should generally specify the columns to analyze. If you submit a script that launches a platform without specifying columns and roles, you get the dialog box for launching the platform. After you choose columns and click OK, you’ll get the analysis you specified by script—in other words, JMP will remember and obey any other messages in your script after getting the column assignments it needs.

If you want to use an expression to be evaluated for column arguments, put the names inside an Eval or EvalList function.

\[
\text{Distribution}(Y(\text{Eval}("X"||\text{char}(i)))));
\]

Column arguments for platform launch scripts can also be lists with braces \{\}, so the following are all valid:

\[
\begin{align*}
\text{Distribution}(Y(\text{height,weight})); \\
\text{Distribution}(Y(\text{\{height,weight\}})); & \quad \text{// equivalent} \\
\text{Distribution}(\text{Weight}(\text{\{})); & \quad \text{// empty specification}
\end{align*}
\]

// (presents the platform’s launch dialog)

Throughout this manual, a \textit{col} placeholder represents any data table column reference, and a \textit{nomCol}, \textit{ordCol}, or \textit{contCol} placeholder suggest that a nominal, ordinal, or continuous column, respectively, would be most appropriate. In many cases, columns of other modeling types would also be accepted. JMP returns an error if you try to cast a column into a role that is strictly disallowed.

Platform Action Command

The command \textit{Action}, if sent to a platform, simply evaluates the expression, whatever it is. This command allows you to chain invocations to platforms where you want to use the platform launch dialog to ask the user to choose columns, but you want to continue the script after the platform is launched.

In the example below, the script first asks you to choose columns for the Distribution platform, when Distribution is run, the action clause brings up the dialog for Bivariate, its action clause for Oneway, its action clause for Contingency, its action clause for printing “Done” on the log. The script is effectively stopped four times, each to prompt for columns for a platform launch. At each step, the user inputs values into a launch dialog. Four reports are open at the completion of the script.

This script runs to completion when the first platform launch dialog is brought up. The other behaviors are run from the stored expressions.

\[
\begin{align*}
\text{Distribution}(\text{Action}(\text{doit})); \\
\text{doit} = \text{expr(} \\
\quad \text{Bivariate}(\text{Action}(\text{doit2})) \\
\); \\
\text{doit2} = \text{expr(} \\
\quad \text{Oneway}(\text{Action}(\text{doit3})) \\
\); \\
\text{doit3} = \text{expr(} \\
\quad \text{Contingency}(\text{Action}(\text{doit4})) \\
\); \\
\text{doit4} = \text{expr(}
\end{align*}
\]
Invisible Reports

Platform launches allow an invisible option, which suppresses the showing of the window. Using this option on a Fit Model script suppresses both the model dialog and the results window.

When using this option, be careful to keep track of the window in the script and close it when the script is done with it, because the invisible windows use resources that must be manually freed.

The following example extracts the F-Ratio from a bivariate report. Make sure Big Class.jmp is open when running this script.

```julia
biv = bivariate(x(height), y(weight), invisible);
biv<<fit line;
r = biv<<report;
fratio = r[ColumnBox("F Ratio")][1];
r<<close window;
```

The invisible option also works on the Table menu operations, suppressing the creation of the window (rather than just hiding it). However, after first use of a data table, it automatically deletes itself when the analysis that uses it is closed.

```julia
currentDataTable() << Select Where(:age==14);
subDt = currentDataTable() << Subset(invisible);
subDt<<bivariate(x(height), y(weight), fit line);
```

After closing the Bivariate results window, the data table automatically deletes itself. Note that the above script could more easily be done by a Where clause in the Bivariate command.

```julia
bivariate(x(height), y(weight), Where(:age==14), fit line);
```

Titles

You can specify the title (shown in the title bar of a platform’s report) by adding the title command to the launch request. For example, the following replaces the standard bivariate report’s title with a user-specified one.

```julia
Bivariate(x(height), y(weight), title("my title"));
```

Objects within analysis objects

Some platforms have a substructure: they have objects inside them that can be addressed with internal Send operators. These are discussed separately for each platform in the following section, “Scripting specific platforms,” p. 190.

General messages for platform windows

The Save Script commands are applicable to almost all of the platforms. Some additional commands are shown in Table 6.2 “Messages that can be sent to platform windows,” p. 188. In particular the Report command lets you access the display surface to control appearance details such as window
zooming, scrolling, and so on. For example, to close an outline node for a report, you first obtain a reference to the DisplayBox tree, subscript to navigate to the outlineBox, and send it the close message:

```plaintext
r = platformRef << Report;
r["Summary of Fit"] << close;
```

This is discussed in greater detail in “Manipulating displays,” p. 227 in the “Display Trees” chapter.

**Table 6.1 Scripting analysis platforms**

<table>
<thead>
<tr>
<th>Goal</th>
<th>Command</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>See available messages</td>
<td>Show Properties(obj)</td>
<td>Shows the messages that a given object can interpret, along with some basic syntax information. Works with all scriptable objects, not just platforms.</td>
</tr>
<tr>
<td>Send message</td>
<td>obj &lt;&lt; message Send(obj, message)</td>
<td>Sends a message to a platform object.</td>
</tr>
<tr>
<td>Send several messages</td>
<td>obj &lt;&lt; message &lt;&lt; message...</td>
<td>Sends a series of messages (in order, left to right) to the platform object.</td>
</tr>
<tr>
<td>Send message to child</td>
<td>obj &lt;&lt; (child &lt;&lt; message)</td>
<td>Sends a message to a child object within a platform object.</td>
</tr>
<tr>
<td>Suppressing output</td>
<td>Platform name(...,invisible)</td>
<td>Keeps output from a report from displaying on-screen. With Fit Model, both the model and dialog are suppressed.</td>
</tr>
<tr>
<td>Changing a report's title</td>
<td>Platform name(...,title(&quot;string&quot;))</td>
<td>Changes the report title to string.</td>
</tr>
<tr>
<td>Changing the automatic recalc setting</td>
<td>Platform name (...automate recalc(boolean))</td>
<td>True (1) sets automatic recalc on, so that any changes in the data or in the excluded state are immediately reflected in the report. False (0) sets automatic recalc off, so you must use redo analysis to see the changes.</td>
</tr>
</tbody>
</table>

**Table 6.2 Messages that can be sent to platform windows**

<table>
<thead>
<tr>
<th>Message</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redo Analysis</td>
<td>obj&lt;&lt;Redo Analysis</td>
<td>Launches the platform again with the same options.</td>
</tr>
<tr>
<td>Save Script to Datatable</td>
<td>obj&lt;&lt;Save Script to Datatable</td>
<td>Saves script to reproduce analysis as a property in the associated data table.</td>
</tr>
<tr>
<td>Save Script to Report</td>
<td>obj&lt;&lt;Save Script to Report</td>
<td>Saves script to reproduce analysis as a text box at the top of the report.</td>
</tr>
<tr>
<td>Save Script to Script Window</td>
<td>obj&lt;&lt;Save Script to Script Window</td>
<td>Saves script to reproduce analysis in the Script Journal window.</td>
</tr>
<tr>
<td>Save Script for All Objects</td>
<td>obj&lt;&lt;Save Script for All Objects</td>
<td>Saves script to reproduce all analyses found within the object’s window in the Script Journal window.</td>
</tr>
<tr>
<td>Get Script</td>
<td>x = obj&lt;&lt;Get Script</td>
<td>Returns script to reproduce the analysis as an expression.</td>
</tr>
<tr>
<td>Data Table Window</td>
<td>obj&lt;&lt;Data Table Window</td>
<td>Makes the associated data table window active (front-most).</td>
</tr>
</tbody>
</table>
Table 6.2 Messages that can be sent to platform windows

<table>
<thead>
<tr>
<th>Message</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close Window</td>
<td>obj&lt;&lt;Close Window</td>
<td>Closes the window identified by the <code>obj</code>, typically a platform surface.</td>
</tr>
<tr>
<td>Move Window</td>
<td>obj&lt;&lt;Move Window(x, y)</td>
<td>Moves the window x pixels to the right and y pixels down.</td>
</tr>
<tr>
<td>Show Window</td>
<td>obj&lt;&lt;Show Window</td>
<td>Shows the identified window.</td>
</tr>
<tr>
<td>Zoom Window</td>
<td>obj&lt;&lt;Zoom Window</td>
<td>Resizes the window to the maximum size of its contents.</td>
</tr>
<tr>
<td>Scroll Window</td>
<td>obj&lt;&lt;Scroll Window(x, y)</td>
<td>Scrolls the window x pixels to the left and y pixels down from the current position; negative coordinates go right and up. If the coordinates are a list in braces <code>{ }</code>, they are absolute coordinates—that is, the window scrolls to the point x pixels from the left and y pixels from the top.</td>
</tr>
<tr>
<td>Bring Window to Front</td>
<td>obj&lt;&lt;Bring Window To Front</td>
<td>Brings the identified window to the front.</td>
</tr>
<tr>
<td>Size Window</td>
<td>obj&lt;&lt;Size Window(x, y)</td>
<td>Resizes the window to x pixels wide by y pixels high.</td>
</tr>
<tr>
<td>Maximize Window</td>
<td>obj&lt;&lt;Maximize Window</td>
<td>Maximizes the window; equivalent to clicking the maximize button in the upper right corner of the window.</td>
</tr>
<tr>
<td>Minimize Window</td>
<td>obj&lt;&lt;Minimize Window</td>
<td>Minimizes the window; equivalent to clicking the minimize button in the upper right corner of the window.</td>
</tr>
<tr>
<td>Print Window</td>
<td>obj&lt;&lt;Print Window</td>
<td>Prints the selected window.</td>
</tr>
<tr>
<td>Get Window Size</td>
<td>obj&lt;&lt;Get Window Size</td>
<td>Gets the size of the selected object. Returns an ordered pair showing width and height.</td>
</tr>
<tr>
<td>Get Window Position</td>
<td>obj&lt;&lt;Get Window Position</td>
<td>Gets the position of the selected object. Returns an ordered pair.</td>
</tr>
<tr>
<td>Report TopReport</td>
<td>obj&lt;&lt;Report Report(obj)</td>
<td>Returns a display box reference for the report in the platform window. See “DisplayBox object references,” p. 230 in the “Display Trees” chapter for a discussion. TopReport goes to the top displayBox, and is useful for By-groups or other cases when several platform reports are in one window.</td>
</tr>
<tr>
<td>Journal Window</td>
<td>obj&lt;&lt;Journal Window</td>
<td>Appends the contents of the window to the journal.</td>
</tr>
</tbody>
</table>
Scripting specific platforms

This section discusses scripting conventions for each of JMP’s platforms, in order of appearance in the Analyze and Graph menu. Simple options such as booleans or actions without arguments are simply listed. Options with arguments or special syntax rules are documented in more detail.

You could use the following terms interchangeably depending on emphasis: command, option, message. The term argument refers to anything that goes in parentheses after an item, and arguments can also contain options with further arguments.

All the scripting commands available for each platform are available from the Indexes > Object Scripting menu of the Help (Windows and Linux) or View Menu. This online list is built by JMP automatically (rather than being hand-typed by a person), so it always contains the most current list of available options.

By

Almost all platforms support the By argument in the launch command, so By is not documented separately for each platform. See “By-group reports,” p. 180, for a general discussion.

Title

You can use a Title command across all platforms, which replaces the standard title. For example,

```
Bivariate (X(height), Y(weight), Title("my title"));
```

Axes

Many of the platforms mention that messages can be sent to axes inside the platform. Messages for numeric axes include:

```
Scale(Log|Linear), Min(value), Max(value), Inc(value), Interval(Numeric|Year|Month|Week|Day|Hour|Minute|Second), Show Major Ticks, Show Minor Ticks, Show Major Grid, Show Minor Grid,
Format(Best|FixedDec|PValue|m#d#y|mmddyyyy|m#y|d#m#y|ddmonyyyy|ddmonyyyy|y#m#d|yyyymmmd|m#d#y h:m|m#d#y h:m|d#m#y h:m|d#m#y h:m:s|ddmonyyyy h:m:s|:day:hr:m|:day:hr:m:s|Date Long|Date Abbrev),
Decimal(value), Show Labels, Rotated Labels, Minor Ticks,
Add Ref Line(number, "Solid|Dashed|Double", Color), Get Script
```

Messages for nominal and ordinal axes:

```
Rotated Tick Labels, Divider Lines, Lower Frame
```

DOE

The DOE platforms (Design of Experiments) are usually used as interactive dialogs, but are scriptable if you wish to drive the platform through scripting. The result of DOE is a data table containing a table property named Model, which is a script to launch Fit Model with the settings appropriate for your design.

The following table lists the available DOE commands.
Save Responses, Load Responses, Save Factors, Load Factors, Save Constraints, Load Constraints, Set Random Seed, Simulate Responses, Suppress Cotter Designs, Show Diagnostics, Save X Matrix, Optimality Criterion, Number of Starts, Sphere Radius, Disallowed Combinations, Advanced Options (Search Points Per Factor, Mixture Sum, Split Plot Variance Ratio, Number of Monte Carlo Samples, N Monte Carlo Spheres), Add Response, Add Factor, Add Term, Add Potential Term, Add Constraint, Make Model, Make Design, Make Table, Set Sample Size, Set N Whole Plots, Set N Subplots, Set Runs Per Random Block, Make Strip Plot Design, Center Points, Replicates, Report

### Tuning Commands

There are scripting commands that let you predefine guidelines for the design search by the custom designer. The following commands are described briefly here, and examples also appear in the *JMP Design of Experiments*.

**DOE Mixture Sum**  If you want to keep a component of a mixture constant throughout an experiment then the sum of the other mixture components must be less than one. In defining the factors you enter the constant mixture component as a constant factor. For example, if you have a mixture factor with a constant value of 0.1, then the command

```
DOE Mixture Sum = 0.9;
```

constrains the remaining mixtures factors to sum to 0.9 instead of the default 1.0.

**DOE Starts**  is the number of random starts used by the custom designer. In some situations, the default number of starts might not produce the design you want. You can increase the number of starts with the `DOE Starts` command. For example, submitting the JSL statement

```
DOE Starts = 100;
```

overrides the default number of starts and sets the number of starts to 100.

**DOE Starting Design**  As an example,

```
DOE Starting Design = matrix;
```

replaces the random starting design with a specified matrix. If a starting design is supplied, the custom designer has only one start using this design.

**DOE Search Points Per Factor**  For a linear model, the coordinate exchange algorithm in the custom designer only considers the high and low values by default. Suppose the low and high values for a factor are -1 and 1 respectively. If you submit the JSL command:

```
DOE Search Points Per Factor = 11;
```

then for each row, the coordinate exchange algorithm checks the eleven values, –1, -0.8, –0.6, –0.4, –0.2, 0, 0.2, 0.4, 0.6, 0.8, 1.0.

**DOE K Exchange Value**  By default, the coordinate exchange algorithm considers every row of factor settings for possible replacement in every iteration. Some of these rows may never change. For example

```
DOE K Exchange Value = 3;
```

sets this value to a lower number, three in this case, so the algorithm only considers the most likely three rows for exchange in each iteration.
DOE Bayes Diagonal  As an example,

DOE Bayes Diagonal = vector:

This vector is used to modify the diagonal elements of the \( X'X \) matrix used for finding the \( D \)-optimal design. The supplied vector is added to the current diagonal elements of the \( X'X \) matrix.

DOE Sphere Radius  constrains the Custom Designer to a sphere instead of a hypercube.

DOE Sphere Radius = \( n \)

where \( n \) is the radius of the constraining sphere.

Tabulate

Tabulate is used as an interactive method of building summary tables. It may be manipulated through JSL with the following commands.

- Add, Add Table, Modify Table, Add Grouping Columns, Add Columns by Categories, Category Table, Add Analysis Columns, Add Statistics, Grouping Columns, Grouping Column, Columns by Categories, Analysis Columns, Analysis Column, Statistics, Add Grouping Columns, Add Analysis Column, Add Aggregate Statistics, Aggregate Statistics, Show Table, Show Control Panel, Show Control, Show shading, Show tool tip, Include missing for grouping columns, Order by count of grouping columns, Show Test Build Panel, Test Build, Test Data View, Make Into Data Table, Set Format, Change Item Label, Remove Column Label, Restore Column Label, Uniform plot scale, Change Plot Scale, Plot Scale, Undo

Distribution

Distribution has two levels of objects, a Distribution Analysis for the collection, and Continuous Distribution and Nominal Distribution objects for the analysis of each column. These objects correspond to the red pop-up menus seen on outline nodes within a Distribution report. (Throughout JMP, anything with a red pop-up menu is an object or a child object.)

There are two styles of launching distribution, a basic launch designed for applying the same options to many variables, and a detailed launch where each column can be analyzed with different options. When you ask Distribution to save a script, it saves it in detailed form.

A basic launch:

\[
\text{dist} = \text{Distribution}(Y(\text{height, weight}), \text{normal quantile plot});
\]

A detailed launch:

\[
\text{dist} = \text{Distribution(}
\begin{align*}
\text{Continuous distribution(Column(\text{height}), \text{normal quantile plot}),} \\
\text{Continuous distribution(Column(\text{weight}), \text{outlier box plot(0))});}
\end{align*}
\]

Launching a platform returns an object reference to the Distribution Analysis collection. References to the Distribution platform can also be subscripted by variable number or by the name of the variable. For example,

\[
\text{myDist} = \text{Distribution}(Y(\text{height, weight}));
\]
myDist["height"] << Test Mean(90);
myDist[2] << Test Mean(90);
results in testing the mean of both height and weight columns.

Options That Are Applied to All Distributions
Uniform Scaling, Stack, Vertical, Automatic Recalc

Complete launching specifications:
Distribution(Y(columns), Freq(frequency column), Weight(weight column),...);

If you send a command to the Distribution Analysis, unless it is one of its own commands, Distribution Analysis simply passes the command to each Distribution in the collection.
The two kinds of Distribution analyses take slightly different commands. If you send Continuous Distribution commands to a Nominal Distribution object, they are ignored, and *vice versa*.

Options for Continuous Distribution:
Quantiles, Moments, More Moments, Horizontal Layout, Histogram, Histogram Color(number), Vertical, Std Error Bars, Count Axis, Prob Axis, Density Axis, Show Percents, Show Counts, Normal Quantile Plot, Outlier Box Plot, Quantile Box Plot, Stem and Leaf, CDF Plot, Test Mean(number), Test Std Dev(number), Confidence Interval(probability), Prediction Interval, Tolerance Interval, Capability Analysis((LSL(number), USL(number), Target(number), Sigma(number), Group(column), Moving Range(number))), Fit Distribution(Normal | LogNormal | Weibull | Weibull with threshold | Extreme Value | Exponential | Gamma | Beta | Johnson Su | Johnson Sb | Johnson Sl | Poisson | Binomial | Smooth Curve), PpK Capability Labeling, Save(Level Numbers | Level Midpoints | Ranks | Ranks averaged | Prob Scores | Normal Quantiles | Standardized | Spec Limits | Script to Log), Axis Settings(axis options)

Options for Nominal Distribution:
Frequencies, Horizontal Layout, Histogram, Histogram Color(number), Std Error Bars, Count Axis, Prob Axis, Density Axis, Show Percents, Show Counts, Mosaic Plot, Test Probabilities(), Confidence Interval(probability), Save(Level Numbers | Script to Log)

Options supported inside Fitted distributions:
Quantile Plot, Density Curve, Goodness of Fit, Fix Parameters(), Quantiles(), Spec Limits(), Save Fitted Quantiles, Save Density Formula, Save Spec Limits, Remove Fit

Options supported inside Quantile Plot:
Rotate, Confidence Limits, Line of Fit, Mean Reference Line, Probability Labels
Fit Y by X

Fit Y by X analyzes the distribution of the Y variable as conditioned by the values of the X variable. To do this, it looks at the modeling type of the variable and calls one of the four platforms, depending on modeling type:

\[
\text{Fit Y by X}( \text{X(columns)}, \text{Y(columns)}, \text{Freq(frequency column)}, \\
\text{Weight(weighting column)},\ldots) 
\]

Table 6.3 Platforms in Fit Y by X

<table>
<thead>
<tr>
<th>Y variable</th>
<th>X variable</th>
<th>Platform that is launched</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous</td>
<td>Continuous</td>
<td>Bivariate</td>
</tr>
<tr>
<td>Continuous</td>
<td>Nominal/Ordinal</td>
<td>Oneway</td>
</tr>
<tr>
<td>Nominal/Ordinal</td>
<td>Nominal/Ordinal</td>
<td>Contingency</td>
</tr>
<tr>
<td>Nominal/Ordinal</td>
<td>Continuous</td>
<td>Logistic</td>
</tr>
</tbody>
</table>

If you specify several X's and/or Y's, JMP creates an analysis for each combination. However, there is no parent object analogous to Distribution Analysis that collects the analyses together.

Bivariate

Bivariate shows how a continuous response variable Y changes with respect to another continuous variable X:

\[
\text{Bivariate}( \text{Y(columns)}, \text{X(columns)}, \text{Freq(frequency column)}, \\
\text{Weight(weight column)},\ldots) 
\]

Bivariate is oriented towards fitting curves, and most of its commands add some form of fit to the graph, along with a report on that fitting curve. These curves are child objects that can be addressed by a subscripted curve notation.

\[
biv = \text{Bivariate}(\text{X(Height)}, \text{Y(Weight)}, \text{Fit Line}); // produces curve[1] \\
biv<<\{\text{curve}[1]<<\text{LineColor("Blue")}; 
\]

Messages sent to child objects can also be sent directly in the launch statement in a list after any arguments needed for the subcommand. For example, the following steps could also be performed in the launch script shown after them:

\[
biv << (\text{curve}[1] << \text{line color(blue)}); \\
biv << (\text{curve}[1] << \text{save residuals }); \\
biv << (\text{curve}[1] << \text{save predicteds }); \\
biv << (\text{curve}[1] << \text{confid curves fit }); 
\]

\[
biv=\text{bivariate}(\text{x(height)}, \text{y(weight)}, \\
\text{fit polynomial}(3, \{\text{LineColor(Blue)}, \text{Plot Residuals,} \\
\text{Save Predicteds, Confid Curves Fit}\}));
\]

If you use the Group By feature to produce curves for groups identified by a grouping column, then the script that results has Fit Where clauses to define the subsets for each curve.

\[
\text{Bivariate}(\text{Y(weight)}, \text{X(height)}, \text{Fit Where( :sex == "F", Fit Line)}, \text{Fit Where( :sex == "M", Fit Line)});
\]

Options include:
Show Points(boolean), Fit Mean, Fit Line, Fit Polynomial(degree), Fit Spline(lambda smoothness parameter), Fit Each Value
Fit Special(
   XTran( None | Log | Sqrt | Square | Reciprocal | Exp ),
   YTran( None | Log | Sqrt | Square | Reciprocal | Exp ),
   Intercept(value to constrain intercept to ),
   Slope(value to constrain slope to ),
   Degree(degree of polynomial, defaults to 1)),
Fit Orthogonal(Univariate Variances | Prin Comp | Equal Variances | Fit X to Y | Specified Variance Ratio),
Density Ellipse(normal probability contour),
Nonpar Density(options), Histogram Borders, Paired t test, Kernel Smoother

Different types of curves support different options including:
Line of Fit, Confid Curves Fit, Confid Curves Indiv, Line Color, Line Style, Line Width, Report, Save Predicteds, Save Residuals, Remove Fit, Plot Residuals, Set Alpha Level, Confid Shaded Fit, Confid Shaded Indiv

Nonpar Density supports:
   Kernel Control, 5% contours, Mesh Plot, Modal Clustering, Save Density Grid, Set Kernal(xBandwidth, yBandwidth), Report, Remove Fit

XAxis and YAxis messages can be used to route Axis commands to graphs.

Spline Fits
There are also functions that allow spline fits. In each of the following functions, $x$ is a vector of regressor variables, $y$ is the vector of response variables, and $\lambda$ is the smoothing argument (larger values result in smoother splines).

coef = Spline Coef(x, y, lambda);
returns a five column matrix of the form
   knots||a||b||c||d
where knots is the unique values in $x$, and the spline calculated using the coefficients in the other columns as described with the Spline Eval function.

yhat=Spline Smooth(x, y, lambda);
returns the smoothed predicted values from the spline fit.

yhat=Spline Eval(x, coef);
evaluates the spline predictions using the coef matrix in the same form as returned by SplineCoef, i.e. knots||a||b||c||d. The $x$ argument can be a scalar or a matrix of values to predict. The number of columns of coef can be any number greater than 1 and each is used for the next higher power. The powers of $x$ are centered at the knot values. For example, the calculation for coef of knots||a||b||c||d is
   $j$ is such that knots[$j$] is the largest knot smaller than $x$
   xx = $x$-knots[$j$] is the centered $x$ value
   result = a[$j$]+xx*(b[$j$]+xx*(c[$j$]+xx^2*d[$j$]))
or, equivalently,
   result = a[$j$] + b[$j$]*xx + c[$j$]*xx^2 + d[$j$]*xx^3
**Oneway**

Oneway shows how a continuous response distributes differently across groups of a categorical column:

\[
\text{Oneway}(X(\text{grouping columns}), Y(\text{response columns}), \text{Block}(\text{blocking column}), \text{Freq}(\text{frequency column}), \text{Weight}(\text{weighting column}),...) \]

Options available include the following:

- Quantiles, Means/Anova, Means/Anova/Pooled \(t\) (boolean), Means and Std Dev (boolean), \(t\) Test, Each Pair, Student's \(t\), All Pairs, Tukey HSD, With Best, Hsu MCB, With Control, Dunnett's({level value}), Wilcoxon Test, Median Test, van der Waerden Test, UNEqual Variances, Equivalence Test, Power, Set Alpha Level (probability), Plot Actual by Quantile, Plot Quantile by Actual, Line of Fit, Probability Labels, CDF Plot, Compare Densities, Composition of Densities, Matching Column (column), Save Residuals, Save Centered, Save Standardized, Save Normal Quantiles, Save Predicted

Display options include:

- All Graphs, Points, Box Plots, Mean Diamonds, Mean Lines, Mean CI Lines, Mean Error Bars, Grand Mean, Std Dev Lines, Comparison Circles, Connect Means, Mean of Means, X Axis proportional, Points Spread, Points Jittered, Matching Lines, Matching Dotted Lines, Histograms

Dunnett's test requires a control group to be identified, e.g.

\[
\text{oneway}(x(\text{age}), y(\text{height}), \text{With Control}({13})); \]

\[
\text{oneway}(x(\text{sex}), y(\text{height}), \text{Dunnett's}({"M"})); \]

If no control group is specified, JMP looks for a single group with a row selected in the data table.

**XAxis** and **YAxis** messages can be used to route Axis commands to graphs.

**Means Comparisons** supports:


---

**Contingency**

Contingency shows how a categorical response distributes differently across groups:

\[
\text{Contingency}(X(\text{grouping column}), Y(\text{categorical response column}), \text{Freq}(\text{frequency count column}), \text{Weight}(\text{weighting column}), \text{Block}(\text{blocking column}),...) \]

Options include:

- Mosaic Plot, Contingency Table, Crosstabs, Tests, Correspondence Analysis, Cochran Mantel Haenszel (blocking column), Agreement Statistic, Relative Risk, Risk Difference, Odds Ratio, 3D Chart

Display Options include **Horizontal Mosaic**.

Crosstab options can be specified directly (without being routed to a child object) and include:
Logistic

Logistic shows how a categorical response distributes differently as a function of a continuous variable.

Logistic( X(continuous regressor column), Y(categorical response column), Freq( frequency count column), Weight(weighting column),...)

Options include:
Odds Ratios, Range Odds Ratios, Unit Odds Ratios, Inverse Prediction(
  Alpha(.05), // the alpha level  
  Response( probabilities to inverse-predict,... ),
  Individual(1), // should not be specified in Logistic 
  Factor(covariate values)  // ignored in Logistic)
),
Logistic Plot, ROC Curve, Lift Curve, Positive Level

Plot options include:
  Show Points, Show Rate Curve, Line Color(color)
XAxis and YAxis messages can be used to route Axis commands to those graphs.

Odds Ratios, Range Odds Ratios, Unit Odds Ratios, ROC curves and Inverse Prediction are only available for two-level responses. Positive Level("level") is used in conjunction with ROC curve to specify which level is viewed as positive.

Matched Pairs

Matched Pairs analyzes how matched variable pairs differ in their means.

Matched Pairs( Y(two or more response columns), X(zero or more grouping columns), Freq(frequency column), Weight(weighting column), ...)

Options include:
  Plot Dif By Mean, Plot Dif By Row, Reference Frame, Wilcoxon Signed Rank, Separate Pairs

Fit Model

Fit Model is a single model specification dialog with a whole collection of platforms for different fitting personalities. A Fit Model script by itself only brings up the dialog; a Run Model command with additional arguments is needed to do the actual fitting.

The following commands correspond to items on the dialog and determine which platform is used:
  Personality( Standard Least Squares | Stepwise | Manova | Loglinear Variance | Nominal Logistic | Ordinal Logistic
| Proportional Hazard | Parametric Survival | Generalized Linear Model),
| Emphasis ( Effect Leverage | Effect Screening | Minimal Report ),
| Method("REML (Recommended)" | "EMS (Traditional)" ),
| Distribution( Weibull | LogNormal | Exponential)

Effects in the model can be more than just a list of columns, and have a specialized syntax:

\[ \text{Effect}( \text{list of effects and/or effect macros}) \]

An effect can be a column name, a crossing of several column names with asterisk * notation, or nested columns specified with subscript bracket [ ] notation. Additional effect options can appear after an ampersand (&) character. Some examples:

- A, \hspace{1cm} // a column name alone is a main effect
- A*B, \hspace{1cm} // a crossed effect, interaction, or polynomial
- A[B], \hspace{1cm} // nested
- A*B[C D], \hspace{1cm} // crossed and nested
- effect&Random, \hspace{1cm} // a random effect
- effect&LogVariance, \hspace{1cm} // a variance model term
- effect&RS, \hspace{1cm} // a response surface term
- effect&Mixture, \hspace{1cm} // for an effect participating in a mixture
- effect&Excluded, \hspace{1cm} // for an effect that doesn't have model parameters
- effect&Knotted, \hspace{1cm} // for a knotted spline effect

Effect macros are:

- Factorial(columns), \hspace{1cm} // for a full factorial design
- Factorial2(columns), \hspace{1cm} // for up to 2nd-degree interactions only
- Polynomial(columnn), \hspace{1cm} // for nth degree polynomial

Scripts for the fitting personality are put inside \texttt{RunModel}, if you want to run the model immediately, or \texttt{Add Script} if you want to set the options but not immediately run the model.

Additional options available include:

- No Intercept, Save to DataTable, Save to Script Window, Load Version 3 Model, Create SAS Job, Submit to SAS, Maximum Iterations, Convergence Limit

Certain scripting-only commands exist so that internal regression matrices can be accessed from scripts.

- \texttt{Get X Matrix}
- \texttt{Get Y Matrix}
- \texttt{Get Parameter Names}
- \texttt{Get Estimates}
- \texttt{Get Std Errors}
- \texttt{Get XPX Inverse}
- \texttt{Get Effect Names}
- \texttt{Get Effect PValues}
- \texttt{Get Random Effect Names}
- \texttt{Get Variance Components}

To implement these commands, assign a reference to the Fit Model call, then send a message to this reference:

\[ \texttt{ft = Fit Model (y(...),effects(...), Run Model);}
\] \[ \texttt{x = ft<<Get X Matrix;} \]
When there are missing values in the data, there is an option to fit the Y's separately in the model. To make a scripted model fit separately, put a `Fit Separately` command in the Run Model clause.

Details of this situation are found in the *JMP Statistics and Graphics Guide*.

If you included `Run Model` in your Fit Model command, you can close the model dialog window:

```julia
model dialog[n] << close window;
```

The `n` is an integer that specifies which model dialog to close. Fit Model dialogs are numbered starting from 1. The subscript is required, even if you have only one Fit Model dialog open. You can also use Model Dialog to run newly constructed models.

```julia
Fit = Fit Model(
    Y( :height ),
    Effects( :sex, :weight ),
    Personality( Stepwise ),
    Run Model( Prob to Enter( 0.5 ), Direction( Mixed ), Prob to Leave( 0.5 ) );
)

Fit << go;
Fit << finish;
Fit << Make Model;

fm = model dialog[2] << run model;
```

**Standard Least Squares**

This platform has several layers corresponding to the overall fit, each response's fit, and each effect's fit within a response.

The overall layer has only a few commands which apply themselves to all responses:

- Profiler,
- Cube Plots,
- Contour Profiler,
- Surface Profiler

There are four commands to get specific information, which are sent to the Fit Model object:

- Get Effect Names,
- Get Effect PValues,
- Get Random Effect Names,
- Get Variance Components

The command `Save Coding Table` produces a new table containing all the coded columns on the X and Y side. Coding refers to all the transformations, including indicator columns, interactions, centerings, transformations. The syntax for this is:

```julia
fitObj <<Save Coding Table;
```

Most of the options are applied to the `Each Response` layer, although you usually send a command to the overall layer and it resends it to each response.

- Summary of Fit,
- Analysis of Variance,
- Parameter Estimates,
- Effect Tests,
- Effect Details,
- Lack of Fit,
- Show Prediction Expression,
- Sorted Estimates,
- Expanded Estimates,
- Indicator Parameterization Estimates,
- Sequential Tests,
- Custom Test...
- Joint Factor Tests,
- Inverse Prediction...
- Parameter Power,
- Correlation of Estimates,
- Scaled Estimates,
- Normal Plot,
- Bayes Plot...
- Pareto Plot,
- Profiler,
- Interaction Plots,
- Contour Profiler,
- Cube Plots,
- Box Cox Y Transformation,
- Surface Profiler,
- Plot Regression,
- Plot Actual by Predicted,
- Plot Effect
Leverage, Plot Residual by Predicted, Plot Residual by Row, Press, Durbin Watson Test

The Save commands are:

- Prediction Formula, Predicted Values, Residuals, Mean Confidence Interval,
  Indiv Confidence Interval, Studentized Residuals, Hats, Std Error of Predicted, Std Error of Residual, Std Error of Individual, Effect Leverage Pairs, Cook's D Influence, StdErr Pred Formula

Commands used when saving formulas with Random Effects:

- Conditional Pred Formula, Conditional Pred Values, Conditional Residuals,
  Conditional Mean CI, Conditional Indiv CI

The commands you can send to an individual effect are:

- LSMMeans Table, LSMMeans Plot, LSMMeans Contrast..., LSMMeans Student's t,
  LSMMeans Tukey HSD, Test Slices, Power Analysis

To send commands to an individual response column's fit, use a syntax like this:

```plaintext
fitObj << (responseName<<{options,...});
```

The `Send` command inside the `Send` command finds the named response and sends the list of commands to it. If you instead send the options directly to the `fitObj` with a single `Send` command, the options are sent to all responses.

To send commands to an individual effect, you nest `Send` commands even further:

```plaintext
fitObj << (responseName<<(effectName)<<effectOption));
```

The following options have special syntax conventions:

- Inverse Prediction(``
  Alpha(.05),       // the alpha level
  Response(response values to inverse-predict)
  Individual(1)    // if the predictions are on individuals
  Factor(covariate values) // values for the x factors
```

- Custom Test(matrix, <Power Analysis(...)>, <Label("...")>)
  LSMMeans Contrast(matrix, <Power Analysis(...)>)

For `Custom Test`, each row of the `matrix` specifies coefficients for all the parameters in the model. For `LSMeans Contrast`, each row of the `matrix` has coefficients for all the levels in the effect. For either, the syntax for the `Power Analysis` portion is:

```plaintext
Power Analysis(``
  Alpha(low, high, increment),
  Sigma(low, high, increment),
  Delta(low, high, increment),
  Number(low, high, increment),
  Solve for Power, Solve for Least Significant Number, Solve for Least Significant Value, Adjusted Power and Confidence Interval,
  Power Plot,
  Done, // to execute the specifications
```

The Scheffé Cubic function is used in fitting certain models. Scheffé Cubic \((X_1, X_2)\) is equivalent to \(X_1X_2^2(X_1-X_2)\)
**Stepwise**

Stepwise options include all the buttons:
- Go, Step, Stop, Finish, Make Model, Enter All, Remove All

The Finish command is like Go, but it forces it to finish before returning from the command, unlike the Go command, which runs in the background.

And other options:
- Prob to Enter (pValue),
- Prob to Leave (pValue),
- Direction (Forward | Backward | Mixed),
- Rules (Combine | Restrict | No Rules | Whole Effects),
- Enter (names of effects,...),
- Enter All (names of effects,...),
- Remove (names of effects,...),
- Remove All (names of effects,...),
- Lock (names of effects,...),
- Unlock (names of effects,...),
- All Possible Models

**Manova**

Manova options include:
- Response Function (matrix | Repeated Measures | Sum | Identity | Contrast | Polynomial | Helmert | Profile | Mean | Compound | Custom), Save Discrim, Save Predicted, Save Residuals, Response

Compound and Repeated are not supported from Scripts yet.

To address an individual response function analysis, use a subscripted Response:

```plaintext
manovaObj << (Response[1]<<{response options});
manovaObj << (Response["Contrast"]<<{response options});
```

Each response function supports this:

- Custom Test(matrix, <Power Analysis(...)>), <Label("...")>

where each row of the matrix specifies coefficients for all the parameters in the model.

To address an individual Effect test, use subscripted Effect with a name or number:

```plaintext
manovaObj << (Response[1]<<{Effect["Whole Model"]<<{effect options}});
manovaObj << (Response[1]<<{Effect[i]<<{effect options}});
```

The effects are numbered 0 for intercept, 1, 2, etc. for regular effects, and n+1 for the “Whole Model” test, where n is the number of effects, not including the intercept.

Each effect in each response function supports these, where each row of the matrix has coefficients for all the levels in the effect:

- Test Details(1),
- Centroid Plot(1),
- Save Canonical Scores,
- Contrast(matrix, <Power Analysis(...)>)

Here is a test script that works on the Dogs data table:
manObj = Fit Model(
    Y(logHist0, logHist1, logHist3, logHist5),
    Effects(depl, drug, drug*depl),
    Personality(Manova),
    Run Model);
manObj<<response function(Contrast, Go);
manObj<<(response[1]()<<CustomTest(
    [0 1 0 0,
     0 0 1 0,
     0 0 0 1])); // same as whole model
manObj<<(response["contrast"]<<(effect["whole model"]<<TestDetails));
manObj<<(response["contrast"]<<(effect[3]()<<TestDetails));

Loglinear Variance

Options include:
- Plot Actual by Predicted, Plot Studentized Residual by Predicted, Plot Studentized Residual by Row, Profiler, Contour Profiler, Surface Profiler

Save Column options include:
- Prediction Formula, Variance Formula, Std Dev Formula, Residuals, Studentized Residuals, Std Error or Predicted, Std Error of Individual, Mean Confidence Interval, Indiv Confidence Interval

Nominal Logistic

Options include:
- Logistic Plot, Likelihood Ratio Tests, Wald Tests, Confidence Intervals, Odds Ratios, Range Odds Ratios, Unit Odds Ratios, Inverse Prediction, Save Probability Formula, ROC Curve, Lift Curve, Profiler, Positive Level(Level)

Plot Options include:
- Show Points, Show Rate Curve, Line Color

Ordinal Logistic

Options include:
- Logistic Plot, Likelihood Ratio Tests, Wald Tests, Confidence Intervals, ROC Curve, Lift Curve, Profiler

Save Options include:
- Save Probability Formula, Save Quantiles, Save Expected Value

Proportional Hazard

Proportional Hazard has one option:
- Risk Ratios
Scripting Platforms

Chapter 6

Scripting specific platforms

Parametric Survival

Options include:

- Likelihood Ratio Tests, Confidence Intervals, Correlation of Estimates,
- Covariance of Estimates, Estimate Survival Probability (regressorTerm = value,..., matrixOfTimes, Alpha(0.05)), Estimate Time
- Quantile (regressorTerm = value,..., matrixOfProbabilities, Alpha(0.05)),
- Residual Quantile Plot, Save Residuals, Get Parameter Names, Get Estimates, Get Std Errors, Get Effect Names, Get Effect PValues

Generalized Linear Model

Fits Generalized Linear Models. Options include:

- Custom Test, Contrast, Covariance of Estimates, Correlation of Estimates,
- Profiler, Contour Profiler, Surface Profiler, Studentized Deviance Residuals by Predicted, Studentized Pearson Residuals by Predicted,
- Deviance Residuals by Predicted, Pearson Residuals by Predicted, Actual by Predicted, Power Link Parameter

Distributions include:

- Normal, Binomial, Poisson, Exponential

Link Functions include:

- Identity, Logit, Probit, Log, Reciprocal, Power(n), Comp LogLog

Save Options include:

- Prediction Formula, Predicted Values, Mean Confidence Interval, Save Indiv Confid Limits, Deviance Residuals, Pearson Residuals, Studentized Deviance Residuals, Studentized Pearson Residuals

Screening

Screening aids model selection for screening designs.

- Screening(Y(numeric columns), X(columns), By(column))

Screening currently has no additional scripting options.

Nonlinear Fit

Nonlinear Fit fits models that are nonlinear in the parameters, defined by a formula with the parameters to estimate:

- Nonlinear(Y(formula columns with parameters), Loss(formula column),
  Freq(frequency column), Weight(column allowing formula),...)

Nonlinear can be launched even if there is no model, as long as there is a Loss function.

The model can be further specified with the following iteration options:

- Newton, QuasiNewton SR1, QuasiNewton BFGS

These commands all have predictable arguments:
Parameter Bounds, Plot, Iteration Log, Numeric Derivatives Only, Second Deriv Method, Accept Current Estimates, Show Derivatives, Unthreaded, Profiler, Contour Profiler, Surface Profiler, Parameter Profiler, Parameter Contour Profiler, Parameter Surface Profiler, Revert to Original Parameters, Remember Solution, Custom Estimate, Custom Inverse Prediction, Save Pred Confid Limits, Save Indiv Confid Limits, Show Prediction Expression, Loss is Neg LogLikelihood, Check Derivatives

The following commands work like the corresponding buttons.

Go, Stop, Step, Reset, Save Estimates, Confidence Limits, Finish

**Note:** Finish is a JSL-only command to keep iterating until converging (or failing to converge) before returning a result and continuing with the script; otherwise Nonlinear Fit runs in the background and JMP proceeds to execute subsequent JSL commands. The Finish message returns the convergence message as a string, which start with either the word “Failed” or “Converged”, so it may be useful to test that the message contains these words. For example:

```javascript
if contains(msg, “Converged”, ...)
```

These commands have specialized arguments:

- `Model(formula with parameters)`, // at launch only
- `Loss Formula(formula with or without parameters)`, // at launch only
- `SSE Grid(parameterName(low, high, increment),...)
- `Delta(very small number like 1e-6)
- `Set Parameter(ParameterName=Expr, ...)
- `Lock Parameter(ParameterName, ...)
- `Unlock Parameter(ParameterName, ...)

These commands for Nonlinear set iteration control arguments, as labeled on the control panel. They are labeled with their current defaults:

- `Iteration Limit(number)default: 60`
- `Shortening Limit(number)default: 15`
- `Obj Change Limit (small value)default: .0000001`
- `Relative Gradient(small value)default: .0000001`
- `Gradient Limit(small value)default: .000001`
- `CL Limit(small value)default: .05`
- `CL Alpha(pvalue)default: .00001`
- `Prim Change Limit (number)`

Testing and estimation features include

- `Custom Estimate(expression)`
- `Custom Inverse Prediction(y Value)`
- `Remember Solution (“Model Name”)`
- `Gradient Limit`
- `Relative Gradient`

Gradient Limit is an absolute test on the gradient, while Relative Gradient is relative to the magnitude of the parameter with which it is associated.

Other scripting options include:

- `Get SSE, Get Parameter Names, Get Estimates, Get Std Errors, Get Corr, Get CI`

Save Options include:
Chapter 6

Scripting Platforms

Scripting specific platforms

Save Prediction Formula, Save Std Error of Predicted, Save Std Error of Individual, Save Residual Formula, Save Inverse Prediction Formula, Save Specific Solving Formula, Prediction Formula, Std Error of Predicted, Std Error of Individual, Inverse Prediction

Neural Net

Performs a neural net fit.

Neural Net( Y(columns),X(columns),Freq(column),Weight(column)...)  

Options include

Crossvalidation, Diagram, Sequence of Fits, Profiler, Categorical Profiler, Contour Profiler, Surface Profiler, ROC Curve, Lift Curve, Save Predicted, Save Predicted and Limits, Save Formulas, Save Hidden and Scaled Cols, Set Random Seed, Log the tours, Log the iterations, Log the estimates, Hidden Nodes(number), Overfit Penalty(number), Number Of Tours(number), Max Iterations(number), Converge Criterion(number), Save iterations in table, Go

Gaussian Process

Gaussian Process is a smoothing fit based on the distance to near neighbors.

Gaussian Process(Y(columns), X(columns), ...)  

The options include:

Estimate Nugget, Profiler, Contour Profiler, Surface Profiler, Save Prediction Formula

Partition

Partition recursively partitions data according to a relationship between the y- and x-values, creating a tree of partitions.

Partition( Y(columns),X(columns),Freq(frequency column),Weight(weight column),...)

Options include:

Split Best, Prune Worst, Criterion, Missing Value Rule(closest|random), Minimum Size Split(number), Lock Columns(columns), Plot Actual By Predicted, Small Tree View, Tree 3D, Leaf Report, Column Contributions, Split History, K Fold Crossvalidation, ROC Curve, Lift Curve, Color Points

Display options include:

Show Points, Show Tree, Show Graph, Show Split Bar, Show Split Stats, Show Split Prob, Show Split Candidates, Sort Split Candidates

Save options include:
Save Residuals, Save Predicteds, Save Leaf Numbers, Save Leaf Labels, Save Prediction Formula, Save Leaf Number Formula, Save Leaf Label Formula

**Time Series**

Time Series models the evolution of a series of observations over time.

\[
\text{Time Series}(Y(\text{columns}), X(\text{time id column}), \ldots);
\]

Options include:


**ARIMA and Seasonal ARIMA**

The syntax to create an ARIMA fit, where \( p \) is autoregressive order, \( d \) is differencing order, and \( q \) is the moving average order, is:

\[
\text{ARIMA}(p, d, q, \text{options})
\]

The syntax to create a seasonal ARIMA fit with \( s \) periods per season is similar but also has arguments for the seasonal orders:

\[
\text{ARIMA}(p, d, q, pSeas, dSeas, qSeas, s, \text{options})
\]

**Smoothing**

To create a smoothing fit, the syntax begins with the name of the model (Simple Exponential Smoothing, Double Exponential Smoothing, … Winters Method from the list above) and then either makes one of three choices:

\[
\text{nameOfSmoothingModel}(\text{ZeroToOne | Unconstrained | StableInvertible, options})
\]

Or is a Custom Model, where you either directly specify each smoothing weight (Level, Trend, Damping, and Seasonal) or specify how it is to be fitted. Not all models use all weights.

\[
\text{nameOfSmoothingModel}(\text{ Custom( Level ( Bounded(min,max) | Fixed(value) | Unconstrained ) Trend ( Bounded(min,max) | Fixed(value) | Unconstrained ) Damping ( Bounded(min,max) | Fixed(value) | Unconstrained ) Seasonal ( Bounded(min,max) | Fixed(value) | Unconstrained ) }}, \text{options})
\]

**Options**

Options inside any ARIMA, Seasonal ARIMA, or smoothing fit:

- Show Points, Show Confidence Interval, Save Columns

Residual Statistics options for ARIMA or smoothing fits:
Scripting Platforms

Chapter 6

Plot, Autocorrelations, Partial Autocorrelations, Variogram

Options for ARIMA and Seasonal ARIMA only (not smoothing fits):

No Intercept, No Constrain, Confidence Intervals

Options inside Differencing:

Difference Graph, Show Points, Connecting Lines, Mean Line, Autocorrelation, Partial Autocorrelation, Variogram, Save, Nonseasonal Order(0|1|2|3|4|5|6|7|8), Seasonal Order(0|1|2|3|4|5|6|7|8), Periods Per Season(number)

Transfer Functions

Options for Transfer Function models include:

Plot, Autocorrelation, Partial Autocorrelation, Variogram, Actual, Time, Predicted, Input List, Std Error of Predicted, Residuals, Upper Confidence Limit, Lower Confidence Limit, No Intercept, Alternative Parameterization, No Constrain, Confidence Intervals, Number of Forecast Periods, Maximum Iterations

Categorical

Categorical models show a distribution of categorical responses from a variety of different response roles.

Categorical(X(columns), Response Role(columns), Response Role(columns), ...)

There are several different Response Roles that you can specify:

Separate Responses, Aligned Responses, Repeated Measures, Rater Agreement, Multiple Responses, Multiple Response by ID, Multiple Delimited, Indicator Group, Responses Frequencies

You can use one or more response roles, and each role can be assigned to one or more columns. You can also add the following optional roles:

X (as a grouping category), Sample Size, Freq, ID, By

Options include:

Frequencies, Share of Responses, Rate Per Case, Share Chart, Frequency Chart, Crosstab Format, Crosstab Transposed, Table Format, Table Transposed, Legend, Test Response Homogeneity, Test Each response, Agreement Statistic, Transition Report

Choice

The Choice platform is designed for use in market research experiments, where the ultimate goal is to discover the preference structure of consumers. Then, this information is used to design products or services that have the attributes most desired by consumers.
When scripting the Choice platform, you must designate the data tables that contain your data. If all your data is in one table, just use `Profile DataTable`. Otherwise, specify each table as needed.

```
Choice( Response DataTable(table), Profile DataTable(table), Subject DataTable(table), Response Profile ID Chosen(column), Response Grouping(table), Response Profile ID Choices(columns), Response Subject ID(column), Response Weight(column), Response Freq(column), Profile ID(column), Profile Grouping(column), Profile Effects(columns), Subject Subject ID(column), Subject Effects(column) )
```

Options include:

- Firth Bias-adjusted Estimates, Likelihood Ratio Tests, Joint Factor Tests, Confidence Intervals, Correlation of Estimates, Effect Marginals, Profiler

Save options include:

- Save Utility Formula, Save Gradients by Subject

**Multivariate**

Multivariate analyzes how variables relate to each other:

```
Multivariate( Y(columns), Freq(frequency column), Weight(weight column), ...)
```

These commands all have predictable arguments:


These commands take specialized arguments:

- Principal Components(on Correlations | on Covariances | on Unscaled | None), Cronbach's Alpha, Standardized Alpha

These commands actually return values:

- Get Correlation Matrix, Get Inv Correlation Matrix, Outlier Analysis

Options that can be specified inside `Outlier Analysis`:

- Mahalanobis Distances, Jackknife Distances, T Square

Options that can be specified inside `Scatterplot Matrix`:

- Density Ellipses, Ellipse Alpha(normal contour probability), Ellipse Color(color), Show Correlations, Horizontal|Vertical(for the Histograms)

Options that can be specified inside `Principal Components`:

- Factor Rotation(Name("1|SMC", number of components, method), Spin Principal Components, Save Principal Components, Save Rotated Components, Reset Scaling, Eigenvectors, Scree Plot, Loading Plot, Score Plot
Methods for factor rotation include

- Varimax
- BiQuartimax
- Equamax
- Factorparsimax
- Orthomax
- Parsimax
- Quartimax
- Biquartimin
- Covarimin
- Obbiquartimax
- Obequamax
- Obfactorparsimax
- Obobimin
- Obparsimax
- Obquartimax
- Obvarimax
- Quartimin

Jackknife limits can be saved as column properties and retrieved using JSL. For example,

```javascript
myJackVal = column(“Jackknife Distances”)<<Get Property(“Jackknife value”);
```

### Hierarchical Clustering

Cluster analysis identifies when data with similar values clump together to form clusters.

```javascript
Hierarchical Cluster( Y(columns), Weight(weight column),
Freq(frequency count column), Ordering(ordering column),
Label(labeling column),...);
```

Options include:

- Color Clusters
- Mark Clusters
- Save Clusters
- Save Display Order
- Save Cluster Hierarchy
- Number of Clusters(n)
- Orientation
- Color Map
- Dendogram Scale
- Geometric X Scale
- Distance Graph
- Two way clustering
- Legend
- Cluster Treatment Comparisons
- Get Column Names

The `Get Column Names` command returns the names of the columns as a list of character values. It is useful when you do two-way clustering—the names are in the order the columns are clustered.

```javascript
obj<<Hierarchical Cluster (Y(...), Two Way Clustering);
colNames = obj<<Get Column Names;
```

### K-means Clustering

Given a specified number of clusters, K-means is a more practical method for clustering larger data sets.

```javascript
KMeans Cluster( Y(columns), Freq(frequency count column),
Weight(weight column), Label(labeling column), Ordering(ordering column),...);
```

Options include:

- Mark Clusters
- Number of Clusters(n)
- Biplot
- Biplot 3D
- Parallel Coord Plots
- Seed With Selected Rows
- Save Clusters
- Save Mixture Probabilities
- Save Mixture Formulas
- Simulate Mixtures
- Save Density Formula
- Get Statistics
- Standardize Data
- Shift distances by rates, Use within cluster std dev, Normal Mixtures, SOM, Color Clusters
- Go, Step, Diagnoal Variance, Log the Tours, Log the Estimates, Log the Iterations, Huber Coverage, Complete Tours, Initial Guesses, Max Iterations

Options for Biplot are:

- Show Biplot Rays
- Biplot Ray Position
Principal Components

The purpose of principal components analysis is to derive a small number of independent linear combinations (principal components) of a set of variables that capture as much of the variability in the original variables as possible. Principal components is available both as a stand-alone platform and within other platforms (Multivariate and Scatterplot 3D).

Principal Components(Y(columns), Freq(columns), Weight(), ...)

Options include:
- Eigenvectors, Scree Plot, Loading Plot, Score Plot, Spin Principal Components, Factor Rotation, Save Principal Components, Save Rotated Components

Discriminant

Performs linear, quadratic, and regularized-parameter discriminant analyses, categorizing multiple y-values into the groups of the x-values.

Discriminant( Y(columns), X(columns), Freq(frequency column), Weight(weight column), ...)

Options include:
- Stepwise Variable Selection, Discriminant Method(Linear, Common Covariance | Quadratic, Different Covariances | Regularized, Compromise Method)
- Score Data, Show Interesting Rows Only, Show Classification Counts, Show Distances to each group, Show Probabilities to each group, ROC Curve, Select Misclassified Rows, Select Uncertain Rows, Save Formulas
- Canonical Plot, Show Points, Show Means CL Ellipses, Show Normal 50% Contours, Show Biplots Rays, Biplot Ray Position, Color Points, Show Canonical Details, Save Canonical Scores, Save to New Data Table
- Canonical 3D Plot, Specify Priors(numbers), Consider New Levels, Save Discrim Matrices, Get Discrim Matrices, Show Within Covariances, Show Group Means,
- Step Forward, Step Backward, Enter All, Remove All, Apply This Model

PLS

PLS analyzes data according to a specified number of latent variables, balancing prediction of the y variables with prediction of the x variables.

PLS( Y(columns),X(columns),...)

Options include:
- Show Points, Show Confidence Lines, Save Prediction Formula, Save Outputs
Chapter 6

Scripting Platforms

Scripting specific platforms

Item Analysis

Item Analysis examines parameters for test items.

Item Analysis(Y(test items), Freq(frequency column),...)

Options include:

Number of Plots Across, Save Ability Formula

Life Distribution

The Life Distribution platform provides tools for discovering distributional properties of time-to-event data.

Life Distribution(Y(time to event), Failure Cause(column), Censor (censor column), ...)

Options include:

Censor Code, Maximum Iterations, Show Quantile Functions, Show Hazard Functions, Show Statistics, Tabbed Report, Show Confidence Area, Change Confidence Level, Comparison Criterion, Set Scale, Suppress Plot

Fit options include:

Fit Nonparametric, Fit Lognormal, Fit Weibull, Fit Loglogistic, Fit Frechet, Fit Normal, Fit SEV, Fit Logistic, Fit LEV, Fit Exponential, Fit LogGenGamma, Fit GenGamma, Fit TH Weibull, Fit TH Lognormal, Fit TH Frechet, Fit TH Loglogistic, Fit ZI Weibull, Fit ZI Lognormal, Fit ZI Frechet, Fit ZI Loglogistic, Fit All Distributions, Fit All Non-negative

Fit Life by X

Fit Life by X helps you analyze lifetime events when only one factor is present.

Fit Life by X(Y(time to event), X(columns), Censor(censor columns), ...)

Options include:

Censor Code, Set Time Acceleration, Time Acceleration Baseline, Change Confidence Level, Confidence Level, Set Scale, Tabbed Report, Show Surface Plot, Maximum Iterations, Add Density Curve to Scatterplot, Add Quantile Line to Scatterplot, Probability, Quantile, Density, Hazard, TAF

Fit options include:

Fit Weibull, Fit Lognormal, Fit Loglogistic, Fit Frechet, Fit All Distributions
Recurrence

Recurrence analyzes how a recurring event is distributed over time, per system, until the system goes out of service.

\[
\text{Recurrence}( \text{Y(age at event column)}, \text{Cost(cost and end of service column)}, \text{SystemID(system ID Column)}, \text{Label(label column)}, \text{Grouping(grouping column)})\]

Options include:
- MCF Plot
- MCF Confid Limits
- Event Plot
- Plot MCF Differences

Survival

Survival analyzes how time-to-event is distributed where times may be censored.

\[
\text{Survival}( \text{Y(survival time columns)}, \text{Freq(frequency count column)}, \text{Grouping(grouping column)}, \text{Censor(censor column)})\]

Options include:
- Survival Plot
- Failure Plot
- Exponential Plot
- Exponential Fit
- Weibull Plot
- Weibull Fit
- LogNormal Plot
- LogNormal Fit
- Generalized Gamma Fit
- Fitted Distribution Plots
- Competing Causes
- Estimate Survival Probability
- Estimate Time Quantile
- Save Estimates

Plot Options include:
- Show Points
- Show Kaplan Meier
- Show Combined
- Reverse Y Axis
- Show Confid Interval
- Show Simultaneous CI
- Midstep Quantile Points
- Connect Quantile Points
- Fitted Quantile
- Fitted Quantile CI Lines
- Fitted Quantile CI Shaded
- Fitted Survival CI
- Fitted Failure CI

Graph Builder

Graph Builder platform allows the interactive display of both continuous and categorical data.

\[
\text{Graph Builder}( \text{Variables(...), Elements(...), ... })\]

Assign variables to roles within the Variables argument:

\[
\text{Variables( X(columns), Y(columns), Group X(column), Overlay(column), Wrap(column) )}\]

Plot options are set within Elements. Options include:
- Points
- Box Plots
- Line
- Smoother
- Bar
- Histogram
- Mosaic

Additional graph options are:
- Show Control Panel
- Size
- X Group Edge
- Y Group Edge
- Back Color
- Grid Color
- Title Fill Color
- Title Text Color
- Title Frame Color
- Title Transparency
- Title Underline
- Level Fill Color
- Level Text Color
- Level Frame Color
- Level Transparency
- Level Underline
Chart

Chart charts data or summary statistics for numeric columns.

\[
\text{Chart}( \ Y(\text{statistic}(\text{column},\ldots),\ldots), \ X(\text{level column}), \\
\quad \text{Grouping}(\text{columns}), \text{Freq}(\text{frequency count column}), \\
\quad \text{Weight}(\text{column}),\ldots) 
\]

For example:

\[
\text{ch} = \text{Chart}(X(\text{age}), Y(\text{Min}(\text{height}), \text{Mean}(\text{height}), \text{Max}(\text{height}))); 
\]

Platform options:

Overlay, Vertical, Horizontal, Pie, Range Chart, Stack Bars, Thick Connecting Line, Show Y Legend, Show Level Legend, Separate Axes, Ungroup Charts

Options for each Y:

Bar Chart, Line Chart, Needle Chart, Point Chart, Show Points, Connect Points, Std Error Bars, Overlay Color(color), Overlay Marker(marker), Overlay Pattern(pattern), Pen Style(line style), Label Format

Label options include:

Show Labels, Remove Labels, Label by Value, Label by Percent of Total Values, Label by Row

Options for each level:

Colors(color), Markers(marker)

Send messages to a subscripted Y or Level to set the attributes for an individual Y or individual level, e.g.

\[
\text{chart} \ (x(:\text{age}), y(\text{mean}(:\text{height})), \text{level}[1] << \text{colors}(8)); \\
\text{chart} \ (x(:\text{age}), y(\text{min}(:\text{height}), \text{mean}(:\text{height}), \text{max}(:\text{height})), \text{y}[2] << \text{line chart}); 
\]

The Level index goes from 1 to as many levels there are, and the Y index goes from 1 to as many Ys as there are. The argument to Send can be a list of attributes; if there is only one argument, the list braces {} are optional.

You can send messages to the X Axis, the overlay Y Axis, or individual Y axes. Here, X Axis refers to the common categorical axis (shared by all plot cells), Y Axis[i] refers to the axis for the ith Y, and Y Axis with no subscript applies the commands to the axis of all the Ys.

\[
\text{ch} = \text{Chart}(X(\text{age}), Y(\text{Min}(\text{height}), \text{Mean}(\text{height}), \text{Max}(\text{height}))); \\
\text{ch} << (x \text{ axis} << \{ \text{axis name("abc"), No Major Tick Mark(0)} \}); \\
\text{ch} << (\text{overlay y axis} << \text{show major ticks}(0)); \\
\text{ch} << (\text{y axis}[2] << \text{show major ticks}(0)); 
\]

Overlay Plot

Overlay Plot overlays several lines or markers on a Y axis against a common variable on the X axis:

\[
\text{OverlayPlot}(Y(\text{columns}), X(\text{column}), \text{Grouping}(\text{grouping column}),\ldots); 
\]
Platform options:
Overlay, Separate Axes, Uniform Y Scale, Connect Thru Missing, Range Plot,
Ungroup Plots, Arrange Plots, Lineup

To change a particular axis, use the Send operator to Left Axis or Right Axis. To change options
for a particular variable, use the Send operator to the name of the Y column.

\[
\text{overlayPlotObj} \ll (\text{Left Axis} \ll \{\text{axis options}\});
\]

\[
\text{overlayPlotObj} \ll (\text{X Axis} \ll \{\text{x options}\});
\]

\[
\text{overlayPlotObj} \ll (\text{yColumn} \ll \{\text{y options}\});
\]

To separate the plots and also change the scale of all y-axes:

\[
\text{overlayPlotObj} \ll \{\text{overlay(0), Left axis} \ll \{\text{axis options}\}\};
\]

Axes have the standard axis options. Other options are:
Show Points, Connect Points, Needle, Step, Function Plot, Connect
Color(color), Overlay Marker(marker), Overlay Marker Color(color), Line
Style(style), Line Width

\section*{Scatterplot 3D}

Scatterplot 3D provides a three-dimensional view of your data and an approximation of
higher dimensions through principal components. Scatterplot 3D replaces Spinning Plot,
which is no longer available through the menus.

\[
\text{Scatterplot 3D( Y(columns), Weight(weight column), Freq(frequency
column)...)}
\]

Options include:
Show Points, Drop Lines, Connect Points, Sized Points, Normal Contour
Ellipsoids, Ellipsoid Coverage(n), Ellipsoid Transparency(n), Nonpar
Density Contour, Drop Line Thickness(n), Principal Components, Std Prin
Components, Rotated Components(number of rotated components), Biplot
Rays, Remove Prin Components, Save Prin Components, Save Rotated
Components, Add Variables(column names)

Spinning commands include:
Spin, Spin Start(Pitch|Yaw|Roll, degrees), Spin Stop, Spin Pitch(degrees),
Spin Yaw(degrees), Spin Roll(degrees)

To change columns in the plot:
X Axis (column name), Y Axis(columnname), Z Axis(columnname)

To change the plot’s scaling:
X Scale(Center,Scale), Y Scale(Center,Scale), Z Scale(Center,Scale), Reset
Scaling

Other options:
Viewpoint(n), Perspective(n), Frame3D(n), Home
Spinning Plot

Spinning Plot produces a three-dimensional scatterplot to look at three variables at a time.

Note: Spinning Plot has been replaced by Scatterplot 3D. Old scripts containing Spinning Plot code may not function properly.

Contour Plot

Contour Plot draws contours of a Y variable with respect to two X variables:

Contour Plot(X(two columns), Y(column), ...)

Options:
Show Data Points, Show Contours, Show Boundary, Fill Areas, Label Contours,
Reverse Colors, Save Contours, Save Triangulation,
Specify Contours( Min(value), Max(value), Inc(increment value) ) |
Specify Contours( Min(value), Max(value), N(number of contours) )

Retrieve Contours and Generate Grid don't support arguments.

In addition, you can specify specific contours to be shown in the plot. To do so, place
Contour (index, x level, color) inside the Specify Contours portion of the script. The index value is the number of the contour that you would like to specify. For example, the following script specifies the values for each of six contours.

Contour Plot(X(:X, :Y), Y(:Z), Show Data Points(0), Fill Areas(1), Label Contours(0), Specify Contours(Min(-5), Max(7.5), N(6), Contour(1, -5, 3), Contour(2, -2.5, 6), Contour(3, 0, 9), Contour(4, 2.5, 12), Contour(5, 5, 4), Contour(6, 7.5, 7)));

You do not have to specify all contours. Those not specified maintain their default values.

Bubble Plot

A bubble plot is a scatter plot which represents its points as circles (bubbles). By specifying additional variables as size, color, group, and time, a bubble plot has the potential to show up to five dimensions at once.

Bubble Plot(X(column), Y(column), Sizes(column), Time(column), ID(column), Coloring(column), ...)

Options include:

Filled, Trails, All Labels, Selectable Across Gaps, Show Roles, Split All, Combine All, Circle Size, Speed, Time Index

With a time variable, you can also step through or animate the bubble plot in sequence. Options to control the time variable are:

Step, Prev, Go, Stop, Split, Combine
Parallel Plot

Parallel plots draw graphs of connected lines, representing each row of the data as a line in the plot.
Options include:
- Show Reversing Checkboxes, Reversed, Scale Uniformly, Center at Zero

Cell Plot

Cell plots, or heat maps, display the data table as a matrix of colored cells.
Options include:
- Legend, Scale Uniformly, Center at Zero

Tree Map

Tree maps draw tiled representations of data.
Options include:
- Change Color Column(column), Color Theme(theme)

Scatterplot Matrix

Scatterplot Matrix produces a grid of bivariate graphs so that comparisons among many variables can be conducted visually.
Options include:
- Density Ellipses, Matrix Format(Lower Triangular | Upper Triangular | Square)

Ternary Plot

Ternary Plot produces a plot to show proportional contribution to a sum of three or more variables in a mixture.

Ternary Plot(Y(3 or more columns), Contour Formula(formula column),...)
Options include:
- Ref Labels, Ref Lines, Contour Fill, 3D Graph, Show Points

Diagram

Diagram plots Ishikawa (fishbone) diagrams.

Diagram(X(Parent), Y(Child),...
Options include:

- Change Type(Hierarchy|Fishbone|Nested), Text(Rotate Left|Rotate Right|Horizontal), Text (Font("font", height, style)),
- Move(First|Last|Other Side), Insert(Above|Below|Before|After)

**Control Chart**

Control Chart draws charts for monitoring variation in a process under statistical control. It has two levels of objects; for each column analyzed there is an object holding the analysis for that column, and within those are objects holding each chart.

To launch Control Chart, use **ChartCol** for each column, with a column name as the first argument. Within that, use the chart names, like **XBar**, **R**, etc. to specify what to chart. Here is a launch script for a control chart on two columns, with two charts for each:

```plaintext
cChart = Control Chart( Sample Size(5), K Sigma(3),
          Chart Col(Weight,
                       XBar(ConnectPoints(1),ShowCenterLine(1),ShowControlLimits(1)),
                       R   (ConnectPoints(1),ShowCenterLine(1),ShowControlLimits(1))),
          Chart Col(Width,
                       XBar(ConnectPoints(1),ShowCenterLine(1),ShowControlLimits(1)),
                       R   (ConnectPoints(1),ShowCenterLine(1),ShowControlLimits(1))));
```

To send a command to the platform object:

```plaintext
cChart<<message;
```

To send a command to one column:

```plaintext
cChart<<(Column[name or number]<<message);
```

To send a command to a chart in a column:

```plaintext
report(cChart)[framebox(n)]<<message;
```

For example:

```plaintext
report(cChart)[framebox(1)]<<frame size(400,300);
```

Options for Variable Control Charts:

- Box Plots, Needle, Connect Points, Show Points, Connect Color(color),
- Center Line Color(color), Limits Color(color), Line Width(Thin|Medium|Thick), Point Marker(marker), Show Center Line, Show Control Limits, All Tests, Test 1, Test 2, Test 3, Test 4, Test 5, Test 6, Test 7, Test 8, All Rules, Rule 1 2S, Rule 1 3S, Rule 2 2S, Rule R 4S,
- Rule 4 1S, Rule 10 X, Test Beyond Limits, Show Zones, Shade Zones, OC Curve, Avg, LCL, UCL, X Bar, R, Std, Show Limits Legend, Connect Through Missing, Alarm Script, Save Sigma, Save Limits, Save Summaries,
- Capability, ChartType

Use **Control Limits** to assign the numeric values for the limits.

For **Sigma**, each type of chart uses a different calculation. Refer to the *JMP Statistics and Graphics Guide* for details.
**Alarm scripts**

JMP can run a script to alert you when the data fail one or more tests. (Note that you need to turn on one or more Tests.) You can set up an alert script one of three ways, in order of priority:

1. Send it to the platform with an `Alarm Script` message, or
2. Store it as a data column property named QC Alarm Script, or
3. Store it as a data table property named either QC Alarm Script or QC Test Alert.

When a data point fails a test, Control Chart triggers the script and sets these global variables:

- `qc_test`—which test number failed
- `qc_col`—which column is being used in the control chart
- `qc_sample`—the sample number that triggered the test failure

Here are two different ways to implement a typical alert script:

```julia
cChart<<Alarm Script(
  Speak(match(qc_test,
    1, "One point beyond zone A",
    2, "Nine points in a row in zone C or beyond",
    3, "Six points in a row steadily increasing or decreasing"
    // and so on for each test
  ));
);
```

```julia
CurrentData Table()<<Set Property("QC Alarm Script",
  Speak(match(qc_test,
    1, "One point beyond zone A",
    2, "Nine points in a row in zone C or beyond",
    3, "Six points in a row steadily increasing or decreasing"
    // and so on for each test
  ));
);
```

You can have the alert script simply write a message to the log with `Print`, `Show`, or `Write`, or use other commands such as `Caption`, `Speak`, `Mail`, etc. For example;

```julia
write("Out of Control for test ",qc_test," in column ",qc_col," in sample ",qc_sample);
```

**Multivariate Control Chart**

Options for Multivariate Control Chart are:

- T Square Chart, T Square Partitioned, Set Alpha Level, Show Covariance, Show Correlation, Show Inverse Covariance, Show Inverse Correlation, Show Means, Save T Square, Save T Square Formula, Save Target Statistics, Phase Detection, Principal Components, Save Principal Components, Get Targets
**Variability/Gauge Chart**

Variability Chart shows group-to-group variability.

```plaintext
Variability Chart( Y(response columns), X(grouping columns),
Freq(frequency column), Standard(standard column)...
```

The platform invokes REML if the data is unbalanced. To force it to compute REML even when the design is balanced, set the Vari Chart Force REML to 1.

```plaintext
Vari Chart Force REML = 1;
```

Options include:

- Variability Chart, Show Points, Show Range Bars, Show Cell Means, Connect Cell Means, Show Group Means, Show Grand Mean, Show Grand Median, Show Box Plots, Mean Diamonds, XBar Control Limits, Show Box Plot Whisker Bars, Points Jittered, Show Bias Line, Variability Summary Report, Std Dev Chart, Mean of Std Dev, S Control Limits, Group Means of Std Dev, Analysis Type, Variance Components, Gauge RR, Gauge RR Report, Discrimination Ratio, Misclassification Probabilities, Bias Report, Linearity Study, Mean Plots, Std Dev Plots, AIAG Labels

Note that Gage RR, Gage R&R, Gauge RR, and Gauge R&R are all equivalent.

There are also two commands that set preferences for variability charts:

```plaintext
K Sigma Multiplier(number), Reduced Gauge RR Report(boolean)
```

Attribute Charts have the following options:

- Variability Chart, Show Cell Means, Connect Cell Means, Show Group Means, Show Grand Mean, Show Effectiveness Means, Effectiveness Report

**Pareto Plot**

Pareto charts show how things add up, from high to low:

```plaintext
Pareto( Y(cause column), X(grouping column), Freq(frequency column), Weight(weight column),...)
```

Cause is an alias to Y. Grouping is an alias to X.

Platform options are:

- Percent Scale, N Legend, Category Legend, Pie Chart, Reorder Horizontal, Reorder Vertical, Ungroup Plots, No Plot, Per Unit Rates, Test Rate Within Groups, Test Rates Across Groups, Show Cum Percent Curve, Show Cum Percent Axis, Show Cum Percent Points, Label Cum Percent Points, Cum Percent Curve Color(color)

You can use the Send operator on a subscripted `Cause[n]` to send options to specific causes. The options for causes are:

- Combine Causes, Separate Causes, Move to First, Move to Last, Colors(color), Markers(marker), Label

For example, to launch a Pareto plot for the column Age:

```plaintext
par = pareto plot(cause(:age));
```
To change color or pattern of a specific cause level, e.g. the first level:

\[
\text{par} \ll (\text{cause}[1] \ll \{\text{colors}(3), \text{label(true)}\});
\]

To combine causes:

\[
\text{par} \ll \text{combine causes}([14, 13], [15, 17])
\]

In this example, age 14 and age 13 are to be combined, and age 15 and age 17 are to be combined. Instead of the actual causes (age), you can use an index where bars 1 and 2 are combined, 4 and 5 are combined:

\[
\text{par} \ll \text{combine causes}([1, 2], [4, 5])
\]

To reorder cells, where \(a, b, c\) are categories in the horizontal/vertical dimension:

\[
\text{Reorder Horizontal}(["c","a","b"]), \\
\text{Reorder Vertical}(["c","a","b"])
\]

You only have to name categories that you want to appear out of the default order. For example, if there are 5 categories in the order \(a, b, c, d, e\) by default, and you want the cells to be arranged \(c, a, b, d, e\), you just need to specify:

\[
\text{Reorder Horizontal}("c");
\]

**Capability**

Capability analysis measures the conformance of a process to given specification limits. The Capability platform offers graphical tools such as the Goalpost plot and the box plot to give you quick, visual ways of observing within-spec behavior.

\[
\text{Capability}(\text{Y(response columns)}, \text{Spec Limits}(\text{LSL(number)}, \text{Target(number)}, \text{USL(number)}))...
\]

Options include:

- Goal Post, Normalized Box Plots, Capability Box Plots, Make Summary Table, Capability Indices Report, Individual Detail Reports, Goal Plot Labels

**Profiler**

Profiler explores the surface of formula columns with respect to changes in factor values.

\[
\text{Profiler}(\text{Y(formula columns)},...)
\]

Options include:

- Confidence Intervals, Prop of Error Bars, Sensitivity Indicator, Mixture Profile at Boundary (Turn at Boundaries | Stop at Boundaries), Desirability Functions, Maximize Desirability, Maximization Options, Save Desirabilities, Set Desirabilities, Save Desirability Formula, Reset Factor Grid, Reset Y Scale, Output Grid Table, Output Random Table, Default N, Prediction Intervals, Conditional Predictions, Simulator, Interaction Profiler, Term Value, Reset, Noise Factors

Options for factor settings include:
Remember Settings, Set to Data in Row, Copy Settings Script, Paste Settings Script, Append Settings to Table, Link Profilers, Get Factor Settings, Get Responses, Get Simulator, Set Script

You can get desirability values from the profiler as well. The message Get Desirability \((i)\) returns the desirability values for the \(i\)th response in the form of a Response Limits clause suitable for a data column property. Get Desirability without an argument returns a list of all of these.

The Term Value and Contour Value commands are now always saved in a script, both for the Contour Profiler platform, and for its use inside other platforms. The format is

\[
\text{Term Value (xname(current value, <Min(slider min value)>), <Max(slider max value)>)}
\]
\[
\text{Contour Value(yName(current value, <Min(slider min value)>, <Max(slider max value)>), <LoLimit(lower limit value)>, <HiLimit(upper limit value)>)}
\]

Currently the only scripting access to set desirabilities is through column properties.

The JSL function Desirability accesses the same function used for the Desirability feature in the profiler. Given three points, it fits a function to go through the three points, suitable for defining the desirability of a set of response variables \((y's)\).

Its syntax has the form

\[
\text{Desirability(yVector, desireVector, argument)}
\]

where \(y\text{Vector}\) and \(desire\text{Vector}\) are matrices with three values, corresponding to the three points defining the desirability function. The actual function depends on whether the desire values are in the shape of a larger-is-better, smaller-is-better, target, or antitarget.

The \(y\text{Vector}\) values must be unique, and in ascending order.

The \(d\text{Vector}\) values must be between 0 and 1, not including zero.

This function is mainly used by saved formulas from the profiler.

The following script illustrates the function in the way it is used in the Profiler. Move the drag markers around to see how the function changes.

```julia
// testDesirability.jsl
dvec = [.1 .9 .1];
yvec = [1 5 10];
newWindow("Desirability",
graphBox(xScale(0,12),yScale(0,1),frameSize(500,400),markersize(4);
dragMarker(yvec,dvec);
yfunction(desirability(yvec,dvec,x),x)));
```
The Simulator itself is also scriptable:

\[
\text{Simulator}(1, \text{Factors(...), Responses(...), Defect Profiler}(1), \text{Defect Parametric Profile}(1), \text{Simulation Experiment}(1), \text{Automatic Histogram Update}(1));
\]

You can send messages to individual factors and responses to set their values.

\[
\text{Factors}(\text{SILICA} \ll \text{Random( Normal( 1.25, 0.3266 ) )}, \text{SILANE} \ll \text{Fixed( 50 )}, \text{SULFUR} \ll \text{Random( Exponential( 2.25 ) )}, \text{Responses}(\text{Pred Formula ABRASION} \ll \text{No Noise}, \text{Pred Formula MODULUS} \ll \text{Add Random Noise( 1 )})
\]

Simulator options include:

- Spec Limits, Defect Profiler, Defect Parametric Profile, Simulation Experiment, Automatic Histogram Update

### Contour Profiler

Contour Profiler explores the surface of formula columns with respect to changes in two factors at a time.

\[
\text{Contour Profiler}( Y(\text{formula columns}), ...)
\]

Options include:

- Grid Density("10 x 10" | "20 x 20" | "30 x 30" | "40 x 40"), Graph Updating("Per Mouse Move" | "Per Mouse Up"), Surface Plot, Contour Value, Term Value, Contour Value, Contour Grid(Low,High,Increment), Remove Contour Grid, Remember Settings, Set to Data in Row, Copy Settings Script, Paste Settings Script, Append Settings to Table, Link Profilers, Get Factor Settings, Get Responses, Get Simulator, Set Script, Conditional Predictions, Simulator, Up Dots, Reset, Horizontal Factor, Vertical Factor

See the Profiler section for Term Value and Contour Value options.
Surface Plot

Surface Plot displays up to four dependent variables at a time. Each has a multitude of settings for surfaces, contours, and associated graphs.

Execute a Show Properties (Surface Plot) command to get a complete list of each setting available in the plots.

See the “Three-Dimensional Scenes” chapter for details on scripting surfaces, and the *JMP Statistics and Graphics Guide* for working with Surface Plots.

Mixture Profiler

The Mixture Profiler shows response contours for mixture experiment models, where three or more factors in the experiment are components (ingredients) in a mixture.

Mixture Profiler( Y(formula columns), ... )

Options include:
- Noise Factors, Expand Intermediate Formulas, Objective Formula, Top Factor, Term Value, Contour Value, Left Factor, Right Factor, Up Dots, Contour Grid, Remove Contour Grid

See the Profiler section for Term Value and Contour Value options.

Custom Profiler

The Custom Profiler explores the interaction between prediction formulas.

Custom Profiler(Y(formula columns),...)

Options include:
- Optimize, Trips, Max Cycles, Max Iter, Converge Limit, Get Objective, Objective Formula, Term Value(future arguments), Contour Value(future arguments), Log Iterations, Simulator, Reset

See the Profiler section for Term Value and Contour Value options.
Chapter 7

Display Trees
Create and use dialogs and results windows

Now comes the chance to do more than program. This chapter shows how to create things in windows and interact with them. These sections discuss different kinds of display scripting:

- Navigating displays to manipulate items in report windows, and
- Creating display trees to build new windows with custom results or custom combinations of standard results.

For details on scripting 2- and 3-dimensional plots, see the chapters “Scripting Graphs,” p. 277 and “Three-Dimensional Scenes,” p. 309.
Contents

Manipulating displays ......................................................... 227
  Introduction to display boxes ...................................... 227
  DisplayBox object references .................................. 230
  Sending messages ....................................................... 232
  Using the << operator .................................................. 233
  Platform example ....................................................... 234
  Syntax Reference ....................................................... 238

Constructing display trees .................................................. 240
  Basics ........................................................................... 240
  Updating an existing display ....................................... 242
  Controlling Text Wrap .................................................. 244
  Interactive Display Elements .......................................... 244
  Send messages to constructed displays ......................... 257
  Build your own displays from scratch ......................... 258
  Construct display boxes containing platforms ............... 259
  Construct a custom platform ....................................... 260
  Sheets ........................................................................... 263
  Journals ....................................................................... 265
  Picture Display Type ..................................................... 265

Modal Dialogs ...................................................................... 266
  Constructing modal dialog boxes ................................ 266
  General-purpose modal dialog boxes ......................... 268
  Data column dialog boxes ........................................... 271
  Details common to Dialog and Column Dialog ............... 273
  Advanced example ....................................................... 275

Scripting the Script Editor .................................................. 276
Manipulating displays

Reports in JMP are built in a hierarchical manner by nesting and gluing together different kinds of rectangular boxes. To manipulate reports with scripts, you first need to learn a bit about how these boxes work. You might want to learn more about how this works if:

1. You want to manipulate display boxes from existing reports.
   This section introduces JMP reports in general and then discusses how to navigate reports, extract data from them, and change them from JSL.

2. You want to construct your own reports.
   See the next section, “Constructing display trees,” p. 240, after first reviewing the introduction that begins this section.

Introduction to display boxes

Here are a few diagrams of the most common kinds of display boxes in JMP.

Figure 7.1 The most common kinds of display boxes
VListBoxes glue boxes together vertically.

HListBoxes glue boxes together horizontally.

You can nest VListBoxes and HListBoxes together. Here is a VListBox gluing together a VListBox and two HListBoxes.
JSL has some other display box types for custom displays: Button Box, Slider Box, Tab Box and Global Box. These are discussed under “Interactive Display Elements,” p. 244. In addition, editable text boxes (Text Edit Box) and boxes to make trees similar to those from the Diagram platform (Hier Box) are supported.

When a report is displayed, each box asks its children how big they are, so that it can position them and show them in a tidy arrangement.

Here is a report from Bivariate, annotated to show the structure of the display boxes.
Figure 7.2 A report surface with some of its display boxes identified

Table 7.1 “Types of Display Boxes in the surface of a JMP report,” p. 238, describes the box types and how they arrange their contents. Boxes labeled as “leaf” are those that contain no other boxes inside.

To examine a report’s display box tree

On Windows and Linux, hold down the Control and Shift keys while right-clicking on a blue triangle in the report; select Edit > Show Tree Structure from the menu that appears.

On Macintosh, enlarge the report window; in an empty area of the report, ⌘-shift-right-click (or ⌘-shift-control-left-click) and select Show Tree Structure.
If your report window doesn’t have a blue triangle, enlarge the report and perform the above actions on an empty portion of the report.

Figure 7.3  Show Tree Structure

In this example, the red Outline Box represents the closed **Fit Mean** outline node. The green Outline Box represents the open **Bivariate Fit of height By weight** outline node. Other boxes in the display tree are shown in their relationship to other boxes.

You can also obtain the tree structure through a script: send the `<showtreestructure()>` message to any report. Or, send the message to a piece of the report (any displaybox object) to see the tree structure for just that part of the report.

**DisplayBox object references**

A special kind of JSL value called a *display box reference* can own or reference a DisplayBox. (Syntax summaries throughout this manual use *db* as a placeholder for any display box reference, *dt* for a data table reference, and *obj* for a scriptable object reference.)

You can create a display box reference with the `Report` message to access the top of the display tree associated with a scriptable platform.

```
variable = platform object<<report;
```

For example, to obtain a reference to a report for a bivariate analysis, you could say:
Manipulating displays

```
dt = Open("$SAMPLE_DATA/Big Class.jmp");
biv = bivariate(x(height),y(weight),fit mean, fit polynomial(4));
rbiv = biv<<report;
```

Confusion alert! Do not confuse a reference to a report with a reference to a platform. They are different types of objects and can receive different types of JSL messages. For example, reports can do such things as copy pictures, select display boxes, or close outline nodes. Platforms can do such things as run tests, draw plots, or close entire windows.

**Subscripts**

If you want to make a reference to another part of the report, the easiest thing to do is use the subscript operator to find it. You can use any of the following methods, which all work by searching for a complete text string. The "text" argument can also be any expression that evaluates to a text string.

- `db[outlinebox("text")]] // finds the outline node matching the text`
- `db["text"] // same`
- `db[columnbox("text")]] // finds the column matching the text`
- `db[boxType(n)] // finds the nth display element of the given type; // n can be an expression that evaluates to a number`
- `db[n] // finds the nth child of the box; e.g. the nth // column for a table box.`

Typically you would want to assign the part of the report to a variable so that you could send messages to it easily.

```
Variable = rbiv["search string"];
```

For example, to find the Analysis of Variance report for Bivariate, you could say:

```
r1=rbiv["Analysis of Variance"];```

If you want to identify an item nested several layers down in an outline, use a subscript with more than one argument of any of the above types. After locating the first item, JMP looks inside that item for the next, and so on.

```
db[arg1,arg2,arg3] // finds the first item, then finds the next // starting after that location, and so on
db[arg1][arg2][arg3] // same as comma version```

Note that you can string together subscripts of different types. This would select the 3rd column of the first table of the outline node “Parameter Estimate” under the node “Polynomial Fit Degree=4.”

```
rbiv["Polynomial Fit Degree=4"]["Parameter Estimates"][1][3] << select << reshow;
```

**Note:** When numbering Outline Boxes, it does not matter if the outline is closed or not. However, closed If Boxes do cause fragments of the tree to disappear entirely.

**Wildcards**

Within the Column function, you can use the wildcard character “?” to represent places in the search string where you want to match any sequence of any characters. This is an important technique for writing general scripts that would work with titles that vary according to the columns analyzed. For example, you could use a script like this to match the top of a Distribution report on any columns.
Sending messages

The display reference can be used to send scripts to the display elements using the Send or << operator. For example, if r2 is a reference to an outline node, you could ask it to close itself:

```r2<<close; //closes an outline node```

Close toggles an outline node back and forth between closed and open states, just like clicking the triangle controls in the window. You can include a boolean argument (1 or 0) for “close or leave closed” or “open or leave open.”

```rbiv["Fit Mean"]<<close; //toggle```
```rbiv["Fit Mean"]<<close(1); //close or leave closed```
```rbiv["Fit Mean"]<<close(0); //open or leave open```

You can use Select, Reshow, and Deselect to blink the selection highlight on a display box. Notice that you can string together several << clauses to send an object several messages in a row. The results read left to right, e.g., here Select is done first, then Reshow, then Deselect, then the other Reshow.

```for(i=0,i<10,i++,
   db<<select<<reshow<<deselect<<reshow);```

Learning what you can do with a display box

To determine the messages that a given display box reference can understand, use Show Properties for the object, e.g.,

```show properties(rbiv);
show properties(rbiv[picturebox(1)]);```

These messages are the same as items in the context-menu for a given object, plus a few more. The next section, “Constructing display trees,” p. 240, discusses messages for each display box type in more detail. Show Properties also works with data tables and platforms; see “Learning about data tables’ messages,” p. 121 in the “Data Tables” chapter, and “Learning the messages an object responds to,” p. 184 in the “Scripting Platforms” chapter.

Find a window quickly

You can find a window that has gotten buried under others by sending a window message to it:

```rbiv<<zoom window;```

Customizing reports

The Send To Report and Dispatch commands are used in tandem to customize the appearance of a report. For example, they are used to open and close outline nodes, resize graphics frames, or customize the colors in a graphics frame.

To see examples of Send To Report and Dispatch, run any analysis and change the default appearance of the report. Then, select Script > Save Script to Script Window to see the script.

Send To Report contains a list of commands to be sent to the display tree. In the example below, Send To Report contains two Dispatch commands.
Dispatch is used to send a command to a specific part of a display tree. It has four arguments. The first argument is a list of outline nodes that need to be traversed to find the desired part of the display tree. The second and third arguments work together. The second is the name of a display element, and the third is the display element’s type. These two arguments specify which particular part of the display tree is to be sent a command. The command to send is the fourth argument.

In essence, the Dispatch command specifies an outline node in the report (first argument), further specifies something underneath that outline node (second and third arguments), and specifies a command (fourth argument).

As an example, open the Big Class sample data set and run the attached Bivariate script. This generates a report with a fitted line. Then, open the Lack of Fit outline node, and close the Analysis of Variance outline node. Finally, select Script > Save Script to Script Window. The following script appears. (spacing and line breaks are added here for illustration).

```js
Bivariate(Y( :weight), X( :height), Fit Line,
SendToReport(
    Dispatch( {"Linear Fit"}, "Lack Of Fit", OutlineBox, Close(0)),
    Dispatch( {"Linear Fit"}, "Analysis of Variance", OutlineBox, Close(1))
);
```

The Send To Report command contains two Dispatch commands. These correspond to your two customizations to the default report. Let’s examine the first Dispatch command in detail.

The first argument says to find an outline node named “Linear Fit”. The second and third commands say to further find an Outline Box named “Lack of Fit” underneath the “Linear Fit” outline. The fourth argument is the command to send to this outline box. In this case, the message is Close(0), i.e., open the node.

Note: If there are several outline nodes with identical names, JMP assigns them subscripts. For example, if you have a Bivariate analysis with two quadratic fits (resulting in identical titles), when you dispatch a command to the second fit, the subscript [2] is added to the duplicated title.

The best way to deal with Send to Report and Dispatch commands is to first run a report using the mouse, creating the customizations interactively. Then, examine the script that JMP generates. Remember: the best JSL writer is JMP itself.

**Using the << operator**

Using the << operator, messages can be sent to displays as they are being constructed, not just by sending messages to already-constructed displays. This allows you to disambiguate between evaluating children arguments and option arguments. It also helps make it clearer which argument is an option, and which is a script to run inside the graph.

For example, prior to version 5, HListBox only accepted boxes in the argument list. Now, commands can be inserted in the list of boxes using the << operator. For example, to insert a journal command in an HListBox, do the following:

```js
New Window("Title",
    HListBox(
        ...
        <<journal,
    );
```

As another example, the GraphBox constructor accepts the usual named arguments like this:

```javascript
NewWindow("Title",
  GraphBox(FrameSize(400,400),XScale(0,25),yScale(0,25),<<BackGroundColor("Red"), script...));
```

However, you can use the send operator to send commands to the Frame Box as an alternative to the named arguments—an argument that has more features, since anything that can be sent to a Frame Box is a legal command.

```javascript
NewWindow("Title",
  GraphBox(
    <<FrameSize(400,400),
    <<XAxis(0,25,add ref line(10)),
    <<YAxis(0,25),
    <<BackGroundColor("Red"),
    script...));
```

### Nesting and Precedence

When you stack commands, each command is evaluated left to right.

```
box<<command1<<command2<<command3;
```

sends `command1` to `box`, then `command2` to `box`, then `command3` to `box`. Note that any of the commands can change `box` before the next command is sent.

```
((box<<command1) <<command2) <<command3
```

sends `command1` to `box` and gets the result. `Command2` is sent to that result, and `command3` is sent to the result of `command2`.

```
x = box<<command1<<command2<<command3;
```

The result of `command3` is assigned to `x`. The first two commands are not assigned to `x`, although they may have changed `box`.

```
x = Text Box("nothing");
Print( x << settext("the")
  << settext("first")
  << settext(x << gettext() || "thing")
  << gettext()
);
```

"firstthing"

If you stack several commands, you may want to use parentheses to group the commands to be sure your script does what you want it to.

### Platform example

Here's a script to build an analysis report from start to finish using JSL. First, open a data table.
dt=open("$SAMPLE_DATA/Big Class.JMP");

Now launch a bivariate platform and assign it to the platform reference `biv`.

`biv=bivariate(y(weight),x(height));` // a reference to the PLATFORM

To find out what you can do with the platform itself, use Show Properties on the platform object:

```julia
show properties(biv);
```

- **Show Points** [Boolean]
- **Fit Mean** [New Entity]
- **Fit Line** [New Entity]
- **Fit Polynomial** [Enum] `{2,quadratic,,3,cubic,,4,quartic,5,6}`
- **Fit Special...** [New Entity]
- **Fit Spline** [Enum] `{1000000,stiff,100000,10000,1000,100,1,.1,.01,flexible,Other...}
- **Fit Each Value** [New Entity]
- **Fit Orthogonal** [Enum] `{Univariate Variances, Prin Comp,Equal Variances,Fit X to Y,Specified Variance Ratio...}
- **Density Ellipse** [Enum] `{.99,.95,.90,.50,Other...}
- **Nonpar Density** [New Entity]
- **Group By...** [New Entity]
- **Script** [Subtable]
  - **Redo Analysis** [Action]
  - **Save Script to Datatable** [Action]
  - **Save Script to Report** [Action]
  - **Save Script for All** [Action]
  - **Get Script** [Action] [Scripting Only]
  - **Scroll Window** [Action] [Scripting Only]
  - **Close Window** [Action] [Scripting Only]
  - **Zoom Window** [Action] [Scripting Only]
  - **Size Window** [Action] [Scripting Only]
  - **Move Window** [Action] [Scripting Only]
  - **Journal Window** [Action] [Scripting Only]
  - **Report** [Action] [Scripting Only]
  - **X, Regressor** [Column Names] [Scripting Only] (1UNC)
  - **Y, Response** [Column Names] [Scripting Only] (1UNC)
  - **Freq** [Column Names] [Scripting Only] (01N)
  - **Weight** [Column Names] [Scripting Only] (01N)
```

Note that this is a selected portion of the commands available to the Bivariate platform.

The output in the Log window gives a few ideas for messages to send the platform. (For more on this type of scripting, see the chapter “Scripting Platforms,” p. 173.)

```julia
biv<<Fit Spline(1000000)<<Fit Mean;
biv<<show points(0); //hide plotting symbols
biv<<show points(1); //reshow them
biv<<fit polynomial(4, 2); // degree 4 color 2
biv<<fit polynomial(2,4); //degree 2 color 4
```

Next, get the window to a comfortable size and scroll up to the top to see the graph you just tweaked.

```julia
biv<<size window(500,700);
biv<<scroll window({0,0});
```

Now get to work on the report. First, you need to create a reference, and then see what you can do with it.
Display Trees
Chapter 7
Manipulating displays

rbiv=biv<<report; // a reference to the REPORT
show properties(rbiv);

The log lists the messages to use with reports (note that this is a selected portion of the commands
available to the report):

Close   [Boolean]
Horizontal [Boolean]
Open All Below [Action]
Close All Below [Action]
Open All Like This [Action]
Close All Like This [Action]
Close Where No Outlines [Action]
Show HelpKeys   [Boolean]
DisplayBox  [Subtable]
  Select    [Boolean]
  Deselect  [Boolean]
  Reshow    [Action] [Scripting Only]
  Journal   [Action]
  Copy Picture [Action]
  Page Break [Action]
Class Name     [Action] [Scripting Only]
Child [Action] [Scripting Only]
Sib    [Action] [Scripting Only]
Parent   [Action] [Scripting Only]
Append   [Action] [Scripting Only]
Get Text  [Action] [Scripting Only]
Get HTML [Action] [Scripting Only]
Get RTF   [Action] [Scripting Only]
Get Journal [Action] [Scripting Only]
Get Script [Action] [Scripting Only]
  Save Text [Action] [Scripting Only] ((filePath|""))
  Save HTML [Action] [Scripting Only] ((filePath|"",PNG|Native|JPEG))
  Save RTF [Action] [Scripting Only] ((filePath|"",PNG|Native|JPEG))
  Save Journal [Action] [Scripting Only] ((filePath|""))
  Save Picture [Action] [Scripting Only] ((filePath|"",PNG|Native|JPEG))
Clone Box [Action] [Scripting Only]
Scroll Window [Action] [Scripting Only]
Close Window [Action] [Scripting Only]
Zoom Window [Action] [Scripting Only]
Size Window [Action] [Scripting Only]
Move Window [Action] [Scripting Only]
Journal Window    [Action] [Scripting Only]
Set Title   [Action] [Scripting Only]

Open the Fit Mean node of the outline:
rbiv["Fit Mean"]<<close(0);

And practice selecting some results (submit each line alone to see its result):

rbiv["Summary of Fit"]<<select;
rbiv["Parameter Estimates"]<<select;
rbiv["Analysis of Variance"]<<select;
//and dig way down in the outline tree:
rbiv["Polynomial Fit Degree=2","Parameter ?",
    columnbox("Estimate")]<<select;
rbiv<<deselect;
To get the second Analysis of Variance item, you would do this:

\[
\text{rbiv["Polynomial Fit Degree=2", "Analysis of Variance"]<<select;}
\]

Now change the format of one of the columns in the 4-degree polynomial Parameter Estimates report.

\[
\text{pe=rbiv["Polynomial Fit Degree=4","Parameter ?"];}
\]

\[
\text{ests=pe[columnbox("Estimate")];}
\]

\[
\text{ests<<set format(12,6);}
\]

The first argument to \text{Set Format} sets the column width in pixels. The second sets how many decimal places are shown in the table.

**Figure 7.4 Applying Changes to a Report**

You can even get a single number out of the table; for example, the estimate for the cubic term:

\[
\text{terms=pe[columnbox("Term")];}
\]

\[
\text{for(i=1, i<10 & (terms<<get(i))!="(height-62.55)^3", i++, 0);}
\]

\[
\text{estimate = ests<<get(i);}
\]

\[
0.0703948227446
\]

How does this work? You use a \text{For}-loop to count down to the row for the term you want. Recall that the second argument to \text{For} is a condition; as long as the condition tests true, looping continues. Here the test is "when the string in the Terms column is not "(height-62.55)^3" and we haven't reached the tenth row," so as soon as the string \textit{does} match, looping stops and \textit{i}'s value is the number for the matching row. You then use \textit{i} as a subscript to \text{Get} on the Estimates column.

This method enables you to use a given result as a parameter for further tests. In a process control situation, you might want to keep tabs on a particular result, triggering an email message to your pager if it goes outside a certain range.

\[
\text{resume = Expr( /* some script to do next iteration */ );}
\]

\[
\text{message="Current estimate is " || char(estimate) ||
" on " || mdyhms(today()));}
\]

\[
\text{if(estimate<1, resume, //else:
mail("john.doe@company.com", "estimate exceeded limit", message));}
\]

You can also get values from boxes as a matrix, which you can then use for further computations or write to a data table. You can also make data tables directly:

\[
\text{myMatrix=rbiv[tablebox(3)] << get as matrix;}
\]
myVector=rbiv[tablebox(3)][columnbox("Sum of Squares")]] << get as matrix;
dt<<new column("Sum of Squares",values(myVector));

rbiv[tablebox(3)] << make data table("ANOVA table");

Now adjust the scales on the axes.
rbiv[axisbox(1)]<<min(70)<<max(170); // adjust Y axis
rbiv[axisbox(2)]<<min(50)<<max(70);  // adjust X axis

Continuing with this example, copy the graph at the top of the report. Note that you need to select the picturebox containing the graph; selecting just the graph would leave its axes behind.

rbiv[PictureBox(1)]<<Copy Picture;

You can also save the picture:
rbiv[PictureBox(1)]<<save picture("myGraph.png",png);
rbiv[PictureBox(1)]<<save picture("myGraph.jpg",jpeg);

Finally, journal the report:
rbiv<<journal window;

**Window**

Window finds a window by its title and returns a reference to that window. For example, to close a window titled “Analysis Results” submit the following code.

wdw=Window("Analysis Results");
wdw<<closeWindow();

Window() with no arguments returns a list of all windows open in JMP. Window(n) returns the nth window.

**Syntax Reference**

Below is a summary of the JSL for constructing reports.

**Table 7.1  Types of Display Boxes in the surface of a JMP report**

<table>
<thead>
<tr>
<th>Box type</th>
<th>Can be constructed with</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List Box</td>
<td>H List Box, V List Box</td>
<td>Arranges members horizontally or vertically. Can be constructed with HListBox or VListBox.</td>
</tr>
<tr>
<td>Outline Box</td>
<td>Outline Box</td>
<td>Constructs an outline node with a disclosure icon, a title, and a vertical list of members.</td>
</tr>
<tr>
<td>Table Box</td>
<td>Table Box</td>
<td>Constructs a horizontal list containing columns.</td>
</tr>
<tr>
<td>String Col Box</td>
<td>String Col Box</td>
<td>Constructs a column of string values (leaf).</td>
</tr>
<tr>
<td>Plot Col Box</td>
<td>Plot Col Box</td>
<td>Constructs a display box to graph the numbers.</td>
</tr>
<tr>
<td>Number Col Box</td>
<td>Number Col Box</td>
<td>Constructs a column of numbers (leaf).</td>
</tr>
<tr>
<td>Frame Box</td>
<td>Graph Box</td>
<td>Constructs a graphics frame (leaf).</td>
</tr>
</tbody>
</table>
Table 7.1 Types of Display Boxes in the Surface of a JMP Report (Continued)

<table>
<thead>
<tr>
<th>Box type</th>
<th>Can be constructed with</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Picture Box</td>
<td>Graph Box</td>
<td>Constructs a box that does nothing except mark the position where pictures are taken when converted to pictures and text, as in outputting to a word processing file.</td>
</tr>
<tr>
<td>Axis Box</td>
<td>Graph Box</td>
<td>Constructs a vertical or horizontal axis to put next to FrameBox's to make graphs (leaf).</td>
</tr>
<tr>
<td>Text Box</td>
<td>Text</td>
<td>Displays text (leaf).</td>
</tr>
<tr>
<td>Tab Box</td>
<td>Tab Box(&quot;page title1&quot;, contents of page 1, &quot;page title 2&quot;, contents of page 2, ...)</td>
<td>Constructs a tabbed dialog pane.</td>
</tr>
<tr>
<td>Matrix Box</td>
<td>Matrix Box</td>
<td>Formats a matrix.</td>
</tr>
<tr>
<td>Button Box</td>
<td>Button Box(&quot;text&quot;, script);</td>
<td>Constructs a button with the text and executes the script when the button is clicked.</td>
</tr>
<tr>
<td>Slider Box</td>
<td>Slider Box(min, max, global, script)</td>
<td>Constructs a box with a slider for setting values for the global variable within the range given by min and max. The script is executed for each drag.</td>
</tr>
<tr>
<td>Global Box</td>
<td>Global Box(global)</td>
<td>Constructs a box to display the current value of the global variable; the value is directly editable, and editing the value automatically forces graphs to update.</td>
</tr>
<tr>
<td>Text Edit Box</td>
<td>Text Edit Box(&quot;text&quot;)</td>
<td>Displays editable text.</td>
</tr>
<tr>
<td>HierBox</td>
<td>HierBox(&quot;text&quot;, box tree)</td>
<td>Constructs a node of a tree (similar to Diagram output) containing text. The box tree consists of other box types, including other Hierboxes.</td>
</tr>
<tr>
<td>Icon Box</td>
<td>Icon Box(&quot;name&quot;)</td>
<td>Constructs a display box containing an icon, where name can be the name of any JMP icon. For example, Icon Box(&quot;Nominal&quot;) constructs a display box that contains the Nominal icon.</td>
</tr>
<tr>
<td>Script Box</td>
<td>Script Box(&quot;script&quot;, width, height)</td>
<td>Constructs an editable script editor box. All three arguments are optional.</td>
</tr>
</tbody>
</table>

Table 7.2 Subscripts for a Display Box

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>db[&quot;text&quot;]</td>
<td>Finds the outline in db that has the title text. Note that text must be the complete title, not just a substring of the title, but you can use ? as a wildcard character with a substring to match the rest of the title, e.g., &quot;? Estimates&quot; to find &quot;Parameter Estimates&quot;.</td>
</tr>
<tr>
<td></td>
<td>db[Outline Box(&quot;text&quot;)]]</td>
<td>Finds the outline box containing the text.</td>
</tr>
<tr>
<td></td>
<td>db[Column Box(&quot;name&quot;)]]</td>
<td>Finds the column box containing the text.</td>
</tr>
<tr>
<td></td>
<td>db[boxType(n)]</td>
<td>Finds the nth display box of type boxType.</td>
</tr>
</tbody>
</table>
Constructing display trees

You can use constructor functions to construct your own display and install it in a window. This section shows how to put together displays and send messages to them and concludes with some examples.

Basics

First you need to start a **New Window**, starting with its title, and then you list the items to construct inside the window. All the Display Box constructors end in “Box”. See “Introduction to display boxes,” p. 227, to review what each type of display box looks like. You can nest display boxes inside each other as needed.

With an assignment, you can make a reference to a new window, so that you can send messages to it. (Throughout this document, *db* is a placeholder for a display box reference, *dt* for a data table reference, and *obj* for a scriptable object reference.)

When display objects are created or referred to by JSL, they are freely shared references until they are copied into another display box or until you close the window and they disappear. When you plug a display object into another display tree, JMP makes a copy of it that the new box owns.

**Example**

This example uses **OutlineBoxes**, **HListBoxes**, and **Matrix Boxes** to assemble a partial cheat-sheet for matrix JSL.

```javascript
a=[0 1 2,3 4 5,6 7 8]; b=[8 7 6,5 4 3,2 1 0];
c=identity(3); d=j(3,3,0); e=1::4; f=[+ -,- +];
mcs=New Window("Matrix Cheat Sheet",
   TextBox("This script is a quick reference to matrix constructors."),
   OutlineBox("Starting matrices",
      HListBox( 
         OutlineBox("a",MatrixBox(a)),
         OutlineBox("b",MatrixBox(b)))
   ),
   OutlineBox("Constructors",
         OutlineBox("c=Identity(3)",MatrixBox(c)),
         OutlineBox("d=J(3,3,0)",MatrixBox(d))
   )
)
```

---

Table 7.2  Subscripts for a display box (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db[arg1, arg2, arg3, ...]</td>
<td>Matches the last argument to a display box that is contained by the penultimate argument’s outline node, which is in turn contained by the antepenultimate argument, and so on. In other words, it is a way of digging down the generations of an outline tree to identify a nested display box.</td>
</tr>
</tbody>
</table>
Constructing display trees

OutlineBox("e=1::4", MatrixBox(e)),
OutlineBox("f=[+ - - +]", MatrixBox(f)),
OutlineBox("Diag(a)", MatrixBox(diag(a))),
OutlineBox("VecDiag(a)", MatrixBox(vecdiag(a))),
OutlineBox("a||b", MatrixBox(a||b)),
OutlineBox("a\|b", MatrixBox(a|/b)),
OutlineBox("a\`, MatrixBox(a\`)"));

Now clean up the formatting by sending messages to the display box reference stored in `mcs`:

```plaintext```
for(i=1,i<12,i++,mcs[matrixbox(i)]<<set format(2,1));
```

Figure 7.5 A Matrix Example

Observe how placing `Matrix` Boxes inside `Outline` Boxes is a convenient way to arrange items neatly. You could add more topics by adding more outline branches following this pattern.

Another example

Or, you could do some programming to avoid iterating `Outline` Boxes:

```plaintext```
a=[0 1 2,3 4 5,6 7 8]; b=[8 7 6,5 4 3,2 1 0]; c=identity(3);
d=J(3,3,0); e=1::4; f=[+ - , - +];
mnames={a,b,c,d};
mnamesq={"a", "b", "c", "d"};
cnames={Identity(3), J(3,3,0), e=1::4, f=[+ - , - +], Diag(a), VecDiag(a),
a\|b, a\|/b, a\`};
cnamesq={"Identity(3)", "J(3,3,0)", "e=1::4", "f=[+ - , - +]", "Diag(a)",
"VecDiag(a)", "a\|b", "a\|/b", "a\`"};
tnames={a\`, cholesky(a\+b), designf(a), directproduct(a, b),
hdirectproduct(a, b), inverse(a-c), sweep(a), solve(a-c, b)};
tnamesq={"a\`, "cholesky(a\+b)", "designf(a)", "directproduct(a, b)",
"hdirectproduct(a, b)", "inverse(a-c)", "sweep(a)", "solve(a, b)"};
```

```plaintext```
myOBoxM=expr(OutlineBox(mnamesq[i], matrixbox(eval(mnames[i]))));
myOBoxC=expr(OutlineBox(cnamesq[i], matrixbox(eval(cnames[i]))));
```

Observe how placing `Matrix` Boxes inside `Outline` Boxes is a convenient way to arrange items neatly. You could add more topics by adding more outline branches following this pattern.
myOBoxT=expr(OutlineBox(tnamesq[i], matrixbox(eval(tnames[i]))));

nw=New Window("Matrix Cheat Sheet",
  HListBox(
    vb = vListBox(textBox(" ");
    for(i=1,i<=nitems(mnames),i++,vb<<append(myOBoxM));
    OutlineBox("Some matrices", vb),
    cb = vListBox(textBox(" ");
    for(i=1,i<=nitems(cnames),i++,cb<<append(myOBoxC));
    OutlineBox("Some constructors", cb),
    tb = vListBox(textBox(" ");
    for(i=1,i<=nitems(tnames),i++,tb<<append(myOBoxT));
    OutlineBox("Some transformations", tb));
  for(i=1,i<=nitems(mnames)+nitems(tnames)+nitems(cnames),i++,nw[matrixbox(i )]<<set format(5,2));
)

As you can see, the result is too large to show effectively here, so be sure to try this script yourself.

Figure 7.6 Matrix Cheat Sheet

Updating an existing display

Sometimes, you do not know how many display boxes will appear in a future report. For example, you may be writing a generic script that analyzes and reports on one or more variables. You don't know how many display boxes are needed, since the number of variables may change from one run of the script to the next.

Append, Prepend, and Delete

Use the Append message to add a display box to an existing display. In the script, construct a single, empty box, then <<Append boxes to it for each variable in the analysis.
Chapter 7  Display Trees  243

Constructing display trees

The following code example assumes that there is a list of effect names in the variable `effectsList`, and that each one corresponds to a column in a matrix `varprop`. In other words, `effectsList[1]` is the label for `varprop[0,1]`; `effectsList[2]` is the label for `varprop[0,2]`; and so on.

First, an empty `Outline Box` containing an `HListBox` is made. The interior empty container is given the name `hb`:

```plaintext
Outline Box ("Variance Proportions",
    hb = HListBox());
```

Then, a `for` loop steps through the `effectsList` and adds a `Number Col Box` for each element of `effectsList`:

```plaintext
for(i=1, i<=NItems(effectsList), i++,
    eval(substitute(
        expr(hb << append(Number Col Box(effectslist[i], varprop[0,i]))),
        expr(i), i)
    ));
```

The `Prepend` message works just like `Append`, but adds the item at the beginning of the display box rather than at the end. If the display box is one of several that do not allow appending, then it will delegate the command to a child display box that can accept the command. It is fine to apply it to the top of the tree.

For example,

```plaintext
biv = Bivariate(Y(height), X(weight), Fit Line);
(biv<<report)(Outline Box(1))<<Prepend(Button Box("Click Here for curve",
    biv<<Fit Polynomial(2)));
```

creates a Bivariate report with a least-squares line. Clicking the button (which appears at the top of the report) adds a quadratic curve.

The same thing can be accomplished appending to the top of the tree:

```plaintext
biv = Bivariate(Y(Height), X(Weight), Fit Line);
(biv<<report)<<Prepend(Button Box("Click Here for curve", biv<<Fit Polynomial(2)));
```

The `Delete` method for display boxes makes the display box and all its children vanish. This is useful with the `Append` and `Prepend` methods for building completely dynamic displays. In the example below, a text box is replaced with another text box. In this case, the script could have used `Set Text`, but many display boxes cannot change their content.

```plaintext
x=New Window ("X",
    list = vListBox(
        t1 = Text Box("t1"),
        t2 = Text Box("t2")))

    t1 << Delete;
    list << Append(t1 = Text Box("t1new"));
```

The `Sib Append` message allows you to add a display element to an existing tree. Refer back to the display box tree shown in “To examine a report’s display box tree,” p. 229. Under `ListBox(2)`, you see two picture box trees. The one on the left (`PictureBox(1)`) holds the bivariate scatterplot. This is easily determined by looking at the bottom of the tree and seeing the two axis boxes holding the height
Display Trees  
Constructing display trees

and weight axes. Picture Box (2) holds the Fit Mean menu, determined by looking at the bottom of this branch and seeing the red line and the “Fit Mean” text box.

Suppose you wanted to insert a text box in between these two boxes. We want to append a sibling to Picture Box (1), so we send it the Sib Append message:

```plaintext
biv=Bivariate (Y(height), x(weight), Fit Line);
(biv<<report)[PictureBox(1)]<<SibAppend(TextBox("Hello There"));
```

Figure 7.7 Inserting a Text Box Into a Report

Updating a numeric column

To update a numeric column in a display box table, use the Set Values command.

```plaintext
numColBox<<Set Values ([matrix])
```

The matrix argument specifies the new numbers for the table.

Controlling Text Wrap

Generally, JMP automatically wraps text within a Text Box. However, the default wrap point can be overridden with a Set Wrap (n) message, where n is the number of pixels to display before the wrap point.

You can also use bullet points with Bullet Point(1). Sending this message to a text box places a bullet in front of the text and indents subsequent lines within that text box.

Interactive Display Elements

JMP has three special types of display boxes that you don’t normally see in any of JMP’s platforms: Button Box, Slider Box, and Global Box. These are useful for building custom windows with interactive graphics.

The chapter “Scripting Graphs,” p. 277, showed how Handle and MouseTrap both present controls inside the graph itself. You can also place button, slider, and edit-field controls outside the graph using Button Box, Slider Box, and Global Box. Each create separate display boxes that need to be combined in a new window according to the rules described in “Constructing display trees,” p. 240, but for
basic effects, you can follow the pattern in the examples shown here and postpone the detailed discussion.

Slider Box draws a slider control for picking any value for the global variable you specify, within the range given by the min and max you specify. Any time the slider is moved, the value given by the current position of the slider is assigned to the global, and the graph updates accordingly. Thus, Slider Box is another way to parametrize a graph.

SliderBox(min, max, global, script, <set width(n)>, <rescale slider(min, max)>);

You can send Set Width and Rescale Slider as commands to a slider object. For example:

ex = .6;
New Window( "Example", mybox = Slider Box( 0, 1, ex, Show( ex ) ) );
mybox << Set Width( 200 ) << Rescale Slider( 0, 5 );

Button Box draws a button with the name you specify. Any time the button is clicked, the script executes. The button stays alive and remains available for the life of the window. You might want to use Button Box in combination with sliders, as shown here, or to provide a choice to update a graph to reflect changing data conditions.

ButtonBox("text",script expression);

You can also send a click() command to a button object at any time.

New Window("Hi", hello = Button Box("Hello", Print("hello")))
hello<<click(); //

Global Box shows the name and current value of a JSL global variable. The user can assign a new value to the global by editing the value directly in the window and pressing Enter or Return to commit the change. Graphs using the global automatically update for the new value. If you specify an expression, such as Sqrt(4), the global box first evaluates it and then stores and displays the result, 2.

GlobalBox(global);

Examples

This example combines a graph with two sliders and a button by gluing the graph box, two horizontal boxes, and a button together in a vertical list box.

// Slider LogNormal
lU=1; lS=2;
NewWindow("LogNormal Density",
VListBox(
gr=GraphBox(FrameSize(500,300),XScale(0.01,3), yScale(0,4),
Double Buffer, XAxis(Show Major Grid), YAxis(Show Major Grid),
YFunction(exp(-(log(x)-log(lU))^2/(2*lS^2))/(lS*x*sqrt(2*pi())),x);
text({1,.5},"u ",lU," s ",lS)),
HListBox(SliderBox(0,4,lU,gr<<reshow),TextBox("Mu")),
HListBox(SliderBox(0,4,lS,gr<<reshow),TextBox("Sigma")),
ButtonBox("Reshow",gr<<reshow));
show(gr);
gr<<reshow;
Figure 7.8 Example of Sliders and Buttons in a Report Window

This script is similar but uses Global Box instead of Slider Box:

```plaintext
// Global LogNormal
lU=1; lS=2;
NewWindow("LogNormal Density",
    VListBox(  
        gr=GraphBox(FrameSize(500,300),XScale(0.01,3), yScale(0,4),  
            Double Buffer, XAxis(Show Major Grid), YAxis(Show Major Grid),  
            YFunction(exp(-(log(x)-log(lU))^2/(2*lS^2))/(lS*x*sqrt(2*pi())),x));  
        text({1,1},"u ",lU," s ",lS," or type new values below"),  
        HListBox(Globalbox(lU)),  
        HListBox(GlobalBox(lS))));
```

Figure 7.9  Example of Using a Global Box Instead of Sliders

The example under “Drag functions,” p. 306, shows another use of Button Box.

Non-modal dialog boxes, like the ones used for all JMP launch dialogs, are identical to reports and other display lists. Each of the following elements can be incorporated into them.

The main difference between these functions and their modal counterparts is that these can have an optional script attached to them. If you want an action associated with any of these controls, place a script as the control’s last argument. For example,

```
comboObj = comboBox({“True”, “False”}, <<set(1),print(comboObj<<Get))
```

prints the selected item number (a 1 or a 2 in this case, since there are two items in the combo box) each time the selection is changed.

A second difference between modal and non-modal controls is seen in the way that items and strings are specified as arguments. For modal dialogs, items are generally comma-separated. In non-modal dialogs, the items must be placed in a list. For example, compare the following two instances of a Radio Box control.

```
//modal dialog box
dlg=Dialog(
    cb=ComboBox(“True”, “False”), //note that the items are comma separated
    Button(“OK”)
);

New Window(“Combo Box”,
    cb=ComboBox({“True”, “False”}), //note the items are in a bracketed list
    ButtonBox(“OK”)
)
```

Note the modal dialog box stops the execution of the script, and the second dialog is not drawn until the first is dismissed.
The following controls are available for non-modal boxes.

**Check Box**

Check Box ("item 1", "item 2", ...), <script>)

Any number of the items in the check box can be selected simultaneously.

**Radio Box**

Radio Box({"item 1", "item 2", ...}, <script>)

Only one item in the radio box can be selected at any time.

**Combo Box**

Combo Box({"item 1", "item 2", ...}, <script>)

Items in a combo box are drawn in a drop-down menu.

**List Box**

List Box({"item 1", "item 2", ...}, <width(n)>, <max selected(n)>, <nlines(n)>, <script>)

*Width* is measured in pixels. *Max selected* is the maximum number of items that may be selected in the list box. *Nlines* is the number of lines to display in the box, with a default value of 3.

**Button Box**

Button Box("text", <script>)

Draws a button containing *text*.

You send the message Open Next Outline as a script command, which causes the next outline box to open. If you are sending more than one message to the button box, this must be the first command listed.

You can add a tooltip for your button by sending a Set Tip message to it. For example:

    Button Box( "Hello", <<Set Tip( "World" ) );

creates a button named Hello that, when you hover over it with your mouse, displays a tooltip containing the word World:

Figure 7.10 Tooltip for a button

![ToolTip for a button](image)

Other messages for Button Box include <<Set Button Name("string") and <<Get Button Name.

**Popup Box**

Popup Box({"command1", script1, "command2", script2, ...})

Constructing display trees

Creates a red triangle menu. The following example stores a command list in a global variable, then uses `Popup Box` to display the items.

```plaintext
commandList = {
    "command1", print("command1"),
    "command2", print("command2"),
    "command3", print("command3"),
    "command4", print("command4"),
    ", empty(), //makes a separator line
    "command5", print("command5"),
    "commandThrow", throw("commandThrow1"),
    "commandError", sqrt(1,2,3),
    "commandEnd", print("commandEnd")};

New Window("Test Popup",
    Text Box("Popup Test"),
    Popup Box(commandList);
);
```

Figure 7.11 Pop-up Menu

Note that you can also disable and re-enable the menu using the message `<<enable(boolean)`. An argument of 1 turns the menu on, and an argument of 0 turns the menu off. Using the previous example, you would assign the popup box to a variable, and then send messages to it:

```plaintext
New Window("Test Popup",
    Text Box("Popup Test"),
    mymenu = Popup Box(commandList);
);

mymenu << enable(0);
```

The red triangle is there, but the menu itself is disabled.

**Tab Box**

```
Tab Box("page title 1", contents of page 1, "page title 2", contents of page 2, ...)
```

Draws a tabbed dialog pane.
Tab boxes can also be used in display trees, as in the following example.

```plaintext
NewWindow("test tabbed pages",
  TabBox("First Page",
    vlistBox(
      textBox("first line of first page"),
      textBox("Second line of first page")
    ),
    "Second page",
    vlistBox(
      textBox("first line of second page"),
      textBox("Second line of second page")
    ),
    "Third page",
    vlistBox(
      textBox("first line of Third page"),
      textBox("Second line of Third page")
    )
  ));
```

Figure 7.12 Tab Boxes

You can specify which tab should be selected by sending `<<SetSelected(n)`, where n is the tab number, to the tab box object.

**Text Box**

- **Text Box(“text”)**

  Draws a non-editable text box. Text Boxes are frequently used as labels of other controls.

**Text Edit Box**

- **Text Edit Box ("text")**

  Draws an editable text box.

You can add a script to a Text Edit Box by sending it a script message. This is usually most convenient at the time the box is created. Simply add the script message as the last argument.

For example, the following script sends a message to the log each time the text edit box is changed.

```plaintext
New Window("Text Edit Box",
  TextEditBox("Change Me", <<Script(Print("Changed")))
)
```

By assigning a reference to the Text Box object, its contents can be accessed. The following script echoes the value of the Text Edit box to the log each time it is changed.
New Window("Text Edit Box",
    teb=TextEditBox("Change Me", <<Script(Print(teb<<Get Text)))
)

**Col List Box**

Col List Box (<All>, <width(n)>, <max selected(n)>, <nlines(n)>, <script>)

All specifies that all columns in the current data table should be included. Width is measured in pixels. Max selected is the maximum number of items that may be selected in the list box. Nlines is the number of lines to display in the box.

You can send a Get Items message to a col list box to retrieve a list of all columns selected. Here is an example script showing Get Items in use:

```julia
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
New Window( "Get Items Demonstration",
    H List Box(
        chooseme = Col List Box( All, width( 100 ), nlines( 6 ) ),
        Lineup Box(
            N Col( 1 ),
            Spacing( 3 ),
            Button Box( "Add Column >>",
                listocols << Append( chooseme << GetSelected );
            // Send Get Items to a Col List Box
            Chosen Columns = listocols << GetItems; ),
            Button Box( "<< Remove Column",
                listocols << Remove Selected;
            // Send Get Items to a Col List Box
            Chosen Columns = listocols << GetItems; ),
        ),
        // listocols is a Col List Box
        listocols = Col List Box( width( 100 ), nlines( 6 ) ),
    ),
    Text Box( " " ),
    // Show what Get Items returns
    stuff = Global Box( Chosen Columns ),
);
```

**Line Up Box**

Line Up Box (NCol(nc), <Spacing(pixels)>, displaybox args)

Display boxes specified in the displaybox arguments are drawn in nc columns. Optional spacing can be specified, in pixels, for the space between columns.

**Border Box**

Border Box (Left(pix), Right(pix), Top(pix), Bottom(pix), Sides(int), displaybox)

Used to add space around the displaybox argument. Left, Right, Top, and Bottom add space around the displaybox argument. Sides draws border around the box, as described in Table 7.3.
Additional effects can also be applied to the borders using `Sides`, as described in Table 7.4. To add both an effect and a border, add the two numbers.

For example, this code produces a text box with a border at the top and bottom (Draw Border value of 5) and a white background (Effect value of 32):

```cpp
New Window("Borders",
    Border Box( Sides( 37 ), Text Box("Hello World!" ))
);
```

<table>
<thead>
<tr>
<th>Number</th>
<th>Draw border</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>Top</td>
</tr>
<tr>
<td>2</td>
<td>Left</td>
</tr>
<tr>
<td>3</td>
<td>Top and Left</td>
</tr>
<tr>
<td>4</td>
<td>Bottom</td>
</tr>
<tr>
<td>5</td>
<td>Top and Bottom</td>
</tr>
<tr>
<td>6</td>
<td>Left and Bottom</td>
</tr>
<tr>
<td>7</td>
<td>Top, Left, and Bottom</td>
</tr>
<tr>
<td>8</td>
<td>Right</td>
</tr>
<tr>
<td>9</td>
<td>Top and Right</td>
</tr>
<tr>
<td>10</td>
<td>Left and Right</td>
</tr>
<tr>
<td>11</td>
<td>Top, Left, and Right</td>
</tr>
<tr>
<td>12</td>
<td>Bottom and Right</td>
</tr>
<tr>
<td>13</td>
<td>Top, Bottom, and Right</td>
</tr>
<tr>
<td>14</td>
<td>Left, Bottom, and Right</td>
</tr>
<tr>
<td>15</td>
<td>Top, Left, Bottom, and Right</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Add to Number Above</th>
<th>Additional Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Make the border the highlight color as defined in Preferences. The default is blue.</td>
</tr>
<tr>
<td>32</td>
<td>Make the borderbox's background white.</td>
</tr>
<tr>
<td>64</td>
<td>Erase the background of the border box's container.</td>
</tr>
<tr>
<td>128</td>
<td>Draw the border in an embossed style.</td>
</tr>
</tbody>
</table>

**Panel Box**

Panel Box ("title", displaybox args)

Encloses the `displaybox` argument in a labeled border.
Constructing display trees

Journal Box

The Journal Box function, like other functions ending in Box, constructs a DisplayBox appropriate for gluing together with other Display Boxes to create a display in a window.

The usage is:

```
box = JournalBox("journal text")
```

where "journal text" is text that has been extracted from a journal file.

Since journal text has lots of rules about what boxes can be with other boxes, we recommend that the only way you obtain journal text is to highlight an area, use the Journal command to make a journal containing only that item, save it. Now open the file in a text editor (you may have to change the file extension to do this). Then paste it into your script as the Journal Box argument. We highly recommend that you use the "\[ ... \]" quoting mechanism so that you don’t have to escape double quotes within the journal text.

Another way to get journal text is to send <<GetJournal to displayBoxes.

Below is an example that makes a mosaic plot:

```
NewWindow("MosaicPlot", TextBox("Here is a mosaic Plot"), JournalBox("\[ //Note the quoting mechanism here
PictureBox(sub( BorderBox(top(12), left(5), bottom(5), right(7), sides(143), options(0), xmin(0), ymin(0), sub( ScaleBox(id(2), axis(scaleType(0), scaleOrig(0), scaleWidth(1), widthMajor(0.25), nbin(4), nminor(0), timeCode(0), ndec(3), ndecSpec(0), MinInit(0), MaxInit(1), LinearInit, MajorInit(0.25), MinorInit(0), NObsInit(0), options(showMajorTicks, showMinorTicks, showLabels, fixMinimum, fixMaximum)), length(180), sub( ScaleBox(id(1), length(200), sub( ListBox(horizontal, near, sub( ListBox(horizontal, near, sub( CenterBox(horiz, sub( TextEditBox("age", left))), AxisBox(side(R), size(33, 180), locked(false), scales(0, 2, 0, 2)), ))) ListBox(vertical, near, sub( BorderBox(left(2), bottom(1), right(2), sides(31), options(0), xmin(0), ymin(0), sub( FrameBox(size(200, 180), border(0), flags(0), markerSize(-1), help name("Conting Mosaic"), scales(1, 2, 1, 2), seg( MosaicSeg( num x(2), num y(6), totals(18, 22), cross tabs(0.277777, 0.4444444, 0.722222, 0.833333, 0.9444444, 1, 0.136366, 0.31818, 0.63636, 0.863636, 0.909090909090909, 1), sum weight(40), ycolors(5, -2142812212, -2138140980, -2134077810, -2134096057, 3), vertical ))))), NomAxisBox(size(200, 36), sizeID(1, 0), num labels(2), labels(F, M), value(18, 22), total(40), horizontal, left)), CenterBox(horiz, sub(```
Display Trees

Chapter 7

Constructing display trees

```
TextEditBox("sex")),
NomAxisBox(size(29, 180), sizeID(0, 2), num labels(6), labels("12", "13",
"14", "15", "16", "17"), value(8, 7, 12, 7, 3, 3), total(40), vertical,
left),
BorderBox(left(2), bottom(1), right(2), sides(31), options(0), xmin(0),
ymin(0), sub(
FrameBox(size(10, 180), border(0), flags(0), markerSize(-1), help
name("Conting Mosaic Single"), scales(0, 2, 0, 2), seg(
MosaicSeg(
num x(1), num y(6), totals(40),
cross tabs(0.2, 0.375, 0.675, 0.85, 0.925, 1),
sum weight(40), ycolors(5, -2142812212, -2138140980, -2134077810,
-2134096057, 3), vertical)
))))))))))))))
"") //End of quoting mechanism here
)

Complete Example

The following script generates a sample of many controls illustrated above. The Big Class sample data table is open.

```
New Window("Nonmodal Controls",
Line Up Box(NCol(2), Spacing(3),
Panel Box("Panel Box",
   Check Box("
Check Box 1", "Check Box 2"),
   Radio Box("
Radio Box 1", "Radio Box 2"),
   Combo Box("
Combo Box"),
   Button Box("Button Box"),
   List Box("
List Box 1", "List Box 2"),
),
Col List Box(all)
)
)
```

Figure 7.13  Example of Many Interactive Display Elements
Getting and Setting values of interactive display elements

You can use <<Set Selected (item number, state) command to pre-select an item. You can use this command separately to a saved display box reference, or you can specify it inline as a list box << argument.

To retrieve the selected value, use <<Get Selected, which returns the value of the selected item. << Get Selected Indices returns the index number of the selected item.

Two examples follow to illustrate all three commands.

antennaList = {"Dish","Helical","Polarizing","Radiant Array"};
//method 1
New Window("Test List",
    listObj = List Box (antennaList, print("iList",
        listObj<<getSelected,listObj<<getSelectedIndices))
    );
listObj<<Set Selected(2, 1);

//method 2
New Window("Test List",
    listObj = List Box (antennaList,<<Set Selected(2, 1), print("iList",
        listObj<<getSelected, listObj<<getSelectedIndices))
    )

Figure 7.14  Testing a List

Advanced Example

The following example code draws a replica of the Cluster platform display box, which then launches the platform with the given arguments. Note that some functions (Recall and Help) are not implemented in the script, so an alert dialog is shown when they are clicked. In addition, switching from hierarchical to k-means clustering does not change the dialog like the real cluster dialog does.

    // Launching Dialog for Cluster Platform
    dt = currentDataTable(); nc = ncol(dt);
    lbWidth = 130;
    methodList = {"Average","Centroid","Ward","Single","Complete"};
    notImplemented = expr(Dialog("Feature Not Implemented Yet",Button("OK")));

    clusterDlg = NewWindow("Clustering",
        BorderBox(left(3),top(2),
            VListBox(
                TextBox("Finding points that are close, have similar values"),
                )
        );
Display Trees
Chapter 7
Constructing display trees

HListBox(
    VListBox(
        PanelBox("Select Columns",
            colListData=ColListBox(All,width(lbWidth),nLines(min(nc,10)))))
    PanelBox("Options",VListBox(
        comboObj=comboBox({"Hierarchical","K-Means"},<<Set(1)),
        PanelBox("Method",
            methodObj=RadioBox(methodList,<<Set(3))
        ),
        checkObj=CheckBox({"Standardize Data"},<<Set(1,1))
    ),
    PanelBox("Cast Selected Columns into Roles",
        LineupBox(NCol(2),Spacing(3),
            ButtonBox("Y, Columns",
                colListY<<Append(colListData<<GetSelected)),
                colListY = ColListBox(width(lbWidth),nLines(5),numeric),
            ButtonBox("Ordering",
                colListO<<Append(colListData<<GetSelected)),
                colListO = ColListBox(width(lbWidth),nLines(1),numeric),
            ButtonBox("Label",
                colListL<<Append(colListData<<GetSelected)),
                colListL = ColListBox(width(lbWidth),nLines(1)),
            ButtonBox("By",
                colListB<<Append(colListData<<GetSelected)),
                colListB = ColListBox(width(lbWidth),nLines(1))
        ),
        PanelBox("Action",
            LineupBox(NCol(1),
                ButtonBox("OK", if ((comboObj<<Get)==1,
                    HierarchicalCluster( Y(eval(colListY<<GetItems)),
                        Order(eval(colListO<<GetItems)),
                        Label(eval(colListL<<GetItems)),
                        By(eval(colListB<<GetItems)),
                        Method(methodList[methodObj<<get]),
                        Standardize(checkObj<<get(1))),
                    KMeansCluster( Y(colListY<<GetItems)
                        )
                    )
                )
            ),
            clusterDlg<<CloseWindow
        ),
        ButtonBox("Cancel",
            clusterDlg<<CloseWindow),
        TextBox(""),
        ButtonBox("Remove",
            colListY<<RemoveSelected;
    )
)
Chapter 7
Display Trees

Constructing display trees

colListO<<RemoveSelected;
colListL<<RemoveSelected;
colListB<<RemoveSelected;
);
ButtonBox("Recall",notImplemented),
ButtonBox("Help",notImplemented))
);
);

Figure 7.15 The Cluster Launch Dialog

Send messages to constructed displays

If you assign a construction to a name, that name becomes a reference to the window, which in turn owns the display boxes inside it. Using subscripts, you can then send messages to the display boxes inside the window.

For example, the graphing section shows how to make an interactive sine wave graph. This example automates the interaction, as it were, by sending messages to the frame box inside the window (note the assignment to \( tf \)).

\[
\begin{align*}
&\quad \text{amplitude} = 1; \quad \text{freq} = 1; \quad \text{phase} = 0; \\
&t = \text{New Window( "Wiggle Wave",} \\
&\quad \text{Graph Box(FrameSize(500,300),X Scale(-5,5),Y Scale(-5,5),Double Buffer,} \\
&\quad \quad \quad \text{Y Function(amplitude*Sine(x/freq+phase),x);} \\
&\quad \quad \quad \text{Handle(phase,amplitude,phase=x;amplitude = y);} \\
&\quad \quad \quad \text{Handle(freq,.5,freq=x);} \\
&\quad \quad \quad \text{Text({3, 4},"amplitude: ",Round(amplitude,4),}
\end{align*}
\]

Constructing display trees

{3.5,"frequency: ",Round(freq,4),
{3,3,"phase: ",Round(phase,4))};
tf = t[framebox(1)];
For(amplitude=-4,amplitude<4,amplitude+=.1,tf << reshown);

Use For loops for more complex movement:

amplitude = 1; freq = 1; phase = 0;
for(i=0,i<1000,i++,
    amplitude+=(Random Uniform()-0.5);
    amplitude = if(amplitude>4,4,amplitude<-4,-4,amplitude);
    freq += (Random Uniform()-0.5)/20;
    phase+=(Random Uniform()-0.5)/10;tf<<reshown);

Build your own displays from scratch

This script uses the Summarize operator to collect summary statistics on the Height column of Big Class and then uses display box constructors to show the results in a nicely-formatted window.

dt=Open("$SAMPLE\_DATA/Big Class.jmp");
summarize( a=by(age), c=count,
    sumHt=sum(Height), meanHt=mean(Height),
    minHt=min(Height), maxHt=max(Height));
sr=NewWindow("Summary Results",
    TableBox(
        stringColBox("Age",a),
        NumberColBox("Count",c),
        NumberColBox("Sum",sumHt),
        NumberColBox("Mean",meanHt),
        NumberColBox("Min",minHt),
        NumberColBox("Max",maxHt)));

This produces a window called “Summary Results” like this:

Figure 7.16 Producing a Customized Summary Report

You can use the usual commands for display boxes:

show properties(sr);
sr<<journal;
Construct display boxes containing platforms

Another type of display you might want to construct is simply your own combination of results from the analysis platforms in JMP. Simply script the platform inside a display box, assemble the display boxes into a window, and for ease in routing messages to it later, assign the whole thing to a reference.

This example assembles a “six-pack” of capability analysis tests for a convenient QC overview:

```julia
csp=NewWindow("Capability Sixpack", 
OutlineBox("Capability Sixpack", 
  HListBox(
    VListBox(cc=Control Chart(chart Col(Height, Individual 
      Measurement, Moving Range),K Sigma(3))), 
    VListBox(dist=Distribution(columns(Height))))));
```

Figure 7.17  Example: Capability Sixpack

Now you can work with the window by sending messages to the reference `csp`. This is a display box reference, whose capabilities are similar to those of a `Report` for a platform. You can use multiple-argument subscripting to locate specific items within the outline tree:

```julia
csp["Control ?", "moving range ?"]<<close;
csp["Dist?", "quantiles"]<<close;
```
Notice that this script not only assigned the whole window to a reference but also assigned the platform-launch scripts to names (cc and dist) within their display boxes. This makes it easy to route messages to the platforms. You could in turn get the reports for these and have yet another way to manipulate display boxes. The following are equivalent messages that will reopen the nodes:

\[
\text{rcc} \leftarrow \text{cc} \ll \text{report}; \quad \text{rdist} \leftarrow \text{dist} \ll \text{report}; \\
\text{rcc}"moving range ?" \ll \text{close}; \\
\text{rdist}"quantiles" \ll \text{close};
\]

Naturally, you can send messages directly to the platform references themselves. First find out your options:

\[
\text{show properties(cc)}; \\
\text{show properties(dist)};
\]

The log shows choices matching the pop-up menus for each platform, as usual. To execute choices from JSL, just direct them as messages to the platform references:

\[
\text{cc} \ll \text{needle}; \quad \text{dist} \ll \text{normal quantile plot};
\]

Figure 7.18 Changing a Custom Report

---

**Construct a custom platform**

An example in “Manipulating expressions,” p. 425 in the “Advanced Concepts” chapter, showed how to use JSL’s **SubstituteInto** operator to plug coefficients for a quadratic polynomial into the
Chapter 7  Display Trees  261

Constructing display trees

quadratic formula and then use the formula to calculate the roots of the polynomial. That example required specifying the coefficients as arguments to SubstituteInto.

The section "Modal Dialogs," p. 266 improves on the example by collecting coefficients from the user through a dialog box.

This section further develops the example into a complete customized platform that first puts up a dialog box to ask for coefficients, then finds the roots, and finally displays the results along with a graph in a custom window.

```plaintext
/* FIND THE REAL ROOTS FOR A QUADRATIC POLYNOMIAL */

//First, put up a dialog to collect coefficients from the user:
myCoeffs=dialog(
    vlist("Find the roots for the equation",
        hlist(a=edit number(1),"*x^2 + ",
            b=edit number(2),"*x + ",
            c=edit number(1)," = 0"),
        Button("OK"),
        Button("Cancel")
    )
);

//Second, calculate the results:
// The quadratic formula is x=(-b + - sqrt(b^2 - 4ac))/2a.
// Use a list to store both the + and - results of the +- operation
x={expr((-b + sqrt(b^2 - 4*a*c))/(2*a)),
    expr ((-b - sqrt(b^2 - 4*a*c))/(2*a))};

//Plug the coefficients du jour into the quadratic formula:
substitute into(x,expr(a),mycoeffs["a"], expr(b),mycoeffs["b"],
    expr(c),mycoeffs["c"]);

//Third, test whether real roots were found and make an appropriate
display. If yes (e.g. with dialog's defaults), show roots and a graph:
results=expr(
    xmin=xx[1]-5;
    xmax=xx[2]+5;
    ymin=-20;
    ymax=20;
    myResult=new window("The roots of a quadratic function",
        vlistbox(
            textbox("The real roots for the equation "),
            textbox("||expr(po)|| = 0"),
            hlistbox(textbox("are x"),textbox(xxx)),
            textbox(" "),                    // to get a blank line
            graph box(framesize(200,200),xscale(xmin,xmax),yscale(ymin,ymax),
                linestyle(2),hline(0),
                linestyle(0),yfunction(polynomial,x),
                linestyle(3),pencolor(3),vline(xx[1]),vline(xx[2]),
                markersize(2),marker(0,xx[1],0), {xx[2], 0}))
    )
    )
);```

```
Constructing display trees

Display Trees

Chapter 7

Constructing display trees

)//If no (e.g. with a=3, b=4, c=5), put up an error window
//with a helpful graph:
error=expr(
  new window("Error",
    vlistbox(textbox(" " ),
    textbox(" Polynomial "||po||" has no real roots. "),
    textbox(" "),
    textbox("Examine a graph of the function to get an idea why."),
    graph box(framesize(200,200),xscale(-20,20),yscale(-20,20),
      linestyle(2),hline(0),
      linestyle(0),yfunction(polynomial,x)
  )
)
);

//Either way, the script needs to have some strings ready.
//Rewrite the polynomial with the coefficients specified:
polynomial=expr(a*x^2 + b*x + c);
substitute into(polynomial,expr(a),mycoeffs["a"], expr(b),mycoeffs["b"],
  expr(c),mycoeffs["c"]);
//Store this instance of the polynomial as a string:
po=char(evalexpr(polynomial));
//Store the solution list as a string:
xxx=char(evalexpr(x));
//Now it's ready for the test:
if(isMissing(xx[1]) | isMissing(xx[2]), error, results);

When you run this script, you first see a dialog like this:

Figure 7.19 Example: A Custom Platform Launch Dialog

And then you get a results window, with either the roots or an error message.
Constructing display trees

Figure 7.20 Example: The Custom Platform’s Report

Sheets

Sheet boxes let you create a grid of plots. **H Sheet Box** and **V Sheet Box** contain display boxes and arrange them in columns and rows. The general approach is to first consider what display boxes you want, and in what arrangement. Then, create either an **H** or **V Sheet Box** and send it a **Hold** message for each plot. Finally, create interior **H** or **V Sheet Boxes** and tell each one which plot it should hold.

Here is an example of creating a sheet with four plots: a bivariate plot, a distribution, a tree map, and a bubble plot.

First, open the data table and create a new window.

```julia
Open( "$SAMPLE_DATA/Big Class.jmp" );
New Window( "Example",

V Sheet Box("Example")

We'll use a **V Sheet Box** to organize the window into two columns.

V Sheet Box("Example")

We'll send it four **Hold** messages, one for each plot. The order matters, as you'll see a little later.

```julia
<<Hold(Bivariate( // Plot 1
    Y( :weight ),
    X( :height ),
    Fit Line()
  )),
<<Hold(Distribution( // Plot 2
    Continuous Distribution(
        Column( :height ),
        Horizontal Layout( 1 ),
        Outlier Box Plot( 0 )
  ))
```
Display Trees
Chapter 7
Constructing display trees

)}

<<Hold(Tree Map(Categories(:age))), // Plot 3
<<Hold(Bubble Plot( // Plot 4
  X(:height),
  Y(:weight),
  Sizes(:age),
  Coloring(:sex),
  Circle Size(6.226),
  All Labels(0))

))},

Finally, we'll add two H Sheet Boxes to the V Sheet Box and tell each one which plot it should hold. Each H Sheet Box holds two side-by-side plots. They are held by a V Sheet Box, so the H Sheet Boxes are displayed vertically.

H Sheet Box(
  Sheet Part("",
    Excerpt Box(1),
    Sheet Part("Distribution of height",
      Excerpt Box(2),
      Sheet Part("My Title Here",
        Excerpt Box(4),
        Sheet Part("",
          Excerpt Box(3),
          Sheet Part("",
            Excerpt Box(1))
        )
      )
    )
  )
)

);
Journals

It is relatively easy to put things into a journal window. It is more difficult to see how to manipulate the
journal itself. The following examples show some journal manipulations.

Suppose you start with a report, generated in this case from the Big Class.jmp data table.

\[
biv = \text{bivariate (y(weight), x(height))};
\]

\[
rbiv = biv << \text{report};
\]

To journal the report results, use the journal window message.

\[
rbiv << \text{journals window};
\]

To save the journal to a file,

\[
rbiv << \text{Save Journal("Macintosh HD:users:username:Documents:test.jrn")};
\]

To save the journal as HTML,

\[
rbiv << \text{Save HTML("Macintosh HD:users:username:Documents:test.htm")};
\]

(Note that the above examples use Macintosh standards for file names. Windows and Linux users
should adapt the examples to use file name conventions appropriate for your operating system or use
the POSIX operating system, which is respected on all platforms.)

When using a By variable in a script, the result is a list of references to the analysis results for each By
group. In order to journal all By member parts of the report, you need to use parent messages to get to
the top of the report.

For example, the following code creates a Bivariate report with a By group and then journals the entire
report:

\[
biv = \text{bivariate (y(weight), x(height), by(Sex))};
\]

\[
(\text{report(biv[1])} << \text{parent}) << \text{parent}) << \text{save journal("test.jrn")};
\]

Picture Display Type

JSL has a Picture data type, used to store pictures of JMP output or formulas. You can take a picture of
anything in a displaybox, or create a picture of a text formula as it would look in the Formula Editor.

To create picture data, send a Get Picture message to a displaybox.

\[
displaybox << \text{Get Picture};
\]

The function Expr As Picture evaluates its argument and creates a picture of the expression, using
the same formatting mechanism as JMP’s formula editor. If you have a literal expression as the argu-
ment, remember to enclose it in Expr() so that JMP just takes a picture of the result, rather than eval-
uate the expression.

For example,

\[
\text{New Window("Formula", Expr As Picture(expr(a*b*c+r/d+exp(x)))}));
\]

results in Figure 7.21.
Once you have the picture, there are two ways of using it.

1. Incorporate it into a new Display tree, using a `displayBox` constructor.
2. Write it to a file using `Save Picture`.

   ```
picture << Save Picture("path", type)
   ``

   where `type` can be `WMF`(Windows), `EMF`(Windows), `PICT`(Macintosh), `BMP`(Windows), `JPEG` or `JPG`, or `PNG`.

---

**Modal Dialogs**

JSL supports both modal and non-modal dialog boxes.

- Modal dialog boxes must be answered immediately. Clicking outside the dialog box produces an error beep. Script execution stops until the user responds to the dialog.
- Non-modal dialogs are similar to JMP reports. They do not have to be answered immediately. Dialogs for JMP's launch platforms are non-modal.

**Constructing modal dialog boxes**

This section introduces `Dialog`, a function to create a modal dialog box. When you submit a script with a modal dialog box, JMP draws the dialog box, waits for the user to make choices and click OK, and then stores a list of the variables with their values. Note that the dialog boxes are *modal*, meaning that the user has no choice but to respond to the dialog box—any attempt to click outside the dialog box produces an error beep, and script execution suspends until the user clicks OK or Cancel. You can construct non-modal dialog boxes (like all the launch platform dialog boxes in JMP) by using the `New Window` command.

A variant of `Dialog`—`Column Dialog`—is specifically intended to prompt users to choose columns from the current (topmost) data table. Column dialogs are also modal dialog boxes.

After introducing both forms, this section explores topics common to both types of dialog boxes.

A few tips for using dialog boxes in scripts:

1. Put all the dialog boxes near the beginning of the script, if possible. This way, all the user interaction can be accomplished at once, and then users can leave JMP to finish its work unattended.
2. Make sure your dialog boxes give the user enough information. Don't just present a number field;
How to access built-in dialog boxes

What if you don’t want to build your own dialog box, you just want your script to bring up one of JMP’s built-in dialogs? Perhaps you’d like to have the user locate a data table or launch a platform. To present one of JMP’s dialogs, simply omit required arguments from the JSL command for the task.

Some examples:
```
dt=Open();          // File->Open dialog box
dt<<Summary();      // Tables->Summary dialog box
```

For modal dialog boxes such as Open and Summary, further script execution waits until the modal dialog box is dismissed with OK, Cancel, etc.
```
myFit=Fit Y by X(); // Analyze->Fit Y by X dialog box
myChart=Chart();    // Graph->Chart dialog box
```

Whenever JMP encounters a script to launch a platform that lacks column assignments and column assignments are required for the platform, it produces the platform’s launch dialog box to get column assignments but continues with the script—that is, JMP does not wait for answers, as in the case of Open, Summary, and other modal dialog boxes. See the chapter “Scripting Platforms,” p. 173, for more about launching platforms.

Using the pick dialogs

You can prompt the user to pick a file or directory using the Pick File and Pick Directory commands. Both bring up a platform-specific dialog box that allows the user to navigate to a file (or directory) and store the path as a string.

```
path = Pick Directory (<"prompt message">);
path = Pick File(<"prompt message"> <,"initial directory"> <,filter list>);
```

If no arguments are specified, the standard Open File and Select Folder windows appear. For both Pick File and Pick Directory, the quote string Prompt Message defines the title of the windows.

You can also specify two other arguments for Pick File. The quoted string Initial Directory defines which folder the Open File window uses. If a directory is not defined, or if it is defined as an empty string, JMP’s default directory is used.

You can also define the filter used for the Open File window, forcing it to show only certain file types. This list must use the following syntax:
```
{"Label1|suffix1;suffix2;suffix3", "Label2|suffix4;suffix5"}
```

Each quoted string adds an entry to the File of Type menu in the File Open window. Label defines the text that is displayed for each menu option. The following list of suffixes defines the file types that are displayed if its corresponding label is selected. Note the use of * to list all files in the window.
```
Pick File(
    "Select JMP File",
    "$SAMPLE_DATA",
    {"JMP Files|jmp;jsl;jrn",
     "All Files|*"})
```
To obtain a list of file names in a specified directory, use the `Files In Directory` command:

```r
listOfFileNames = Files In Directory(path);
```

To obtain the full path names, use the concatenation operator:

```r
listOfFilePaths = path || Files In Directory(path);
```

The `Files in Directory` command accepts native and POSIX paths, as well as paths using path variables. It does not recurse by default, but does have an optional `recursive` argument.

```r
main_samples = Files In Directory("$SAMPLE_DATA");
{"Abrasion.jmp", "AdverseR.jmp", "Alcohol.jmp", ...}
all_samples = Files In Directory("$SAMPLE_DATA", recursive);
{..., "Templates/Polar Coordinates.jsl", "Therm.jmp", "Time Series/Air.jmp"...}
```

### General-purpose modal dialog boxes

Inside the `Dialog` function's parentheses, you can use any of its named arguments to create a user interface for setting values for variables. A bare bones dialog box might request a value for one variable.

```r
Dialog(  "Set this value", variable=EditNumber(42),  Button("OK"), Button("Cancel"));
```

Notice that the argument to `EditNumber` is the default value, 42. Also notice that a dialog box without at least one `Button` makes no sense, so JMP adds a button if you script a `Dialog` without one.
Figure 7.23  Sample Modal Dialog

If you click OK, the list \{variable=42,Button(1)\} is returned and the script continues.

If you click Cancel, the dialog is dismissed, it returns \{Button(-1)\}, and the script continues. See “Throwing and catching exceptions,” p. 403 in the “Advanced Concepts” chapter, if you need a way to cancel out of a script.

In addition to Edit Number, you can use Edit Text, Radio Buttons, Check Box, Combo Box, and List Box.

Figure 7.24  Sample List Dialog

List Boxes accept several kinds of arguments, including strings, names, literal lists, or globals containing lists. The following examples all result in the dialog box shown here.

```jmp
//---arguments as strings
dlg = Dialog(“Select Names”,
    vlist(          
        selection=listBox(“KATIE”, “LOUISE”, “JANE”, “JACLYN”, “LILLIE”, “KIRK”, “LAWRENCE”),
        button(“OK”)),
    //result is list of strings: {selection = {“LOUISE”, “JACLYN”},
    Button(1) }

//---arguments as names
dlg = Dialog(“Select Names”,
    vlist(          
        selection=listBox(KATIE, LOUISE, JANE, JACLYN, LILLIE, KIRK, LAWRENCE),
        button(“OK”)),
    //result is list of names: {selection = {LOUISE, JACLYN}, Button(1})

//---arguments as literal list
```
dlg = Dialog("Select Names",
  vlist(
    selection=listBox({KATIE, LOUISE, JANE, JACLYN, LILLIE, KIRK, LAWRENCE}),
    button("OK")))

//---arguments as global containing list
names = {KATIE, LOUISE, JANE, JACLYN, LILLIE, KIRK, LAWRENCE}; dlg =
  Dialog("Select Names", vlist(
    selection=listBox(names),
    button("OK")));

The next example is more elaborate and takes advantage of each type of dialog box element, all nicely arranged.

Dialog( HList(
  VList(
    "Radio Frequency Embolism Projection",
    Lineup(2,
      "Lower Spec Limit", lsl=EditNumber(230),
      "Upper Spec Limit", usl=EditNumber(340),
      "Threshold", threshold=EditNumber(275)),
    HList(
      VList(
        "Type of Radio",
        type=RadioButtons("RCA", "Matsushita", "Zenith", "Sony")),
      VList(
        "Type of Antenna",
        antenna=RadioButtons("Dish", "Helical", "Polarizing", "Radiant Array")),
      synch=CheckBox("Emission Synchronization",0),
      "Title for plot", title=EditText("My projection"),
      HList("Quality",
        quality=ComboBox("Fealty", "Loyalty", "Piety", "Obsequiousness")),
      VList(
        Button("OK"),
        Button("Cancel")))));
Figure 7.25  Sample Dialog Box Controls

![Sample Dialog Box Controls](image)

Clicking OK without changing any of the default choices returns this list:

$$\{isl = 230, usl = 340, threshold = 275, type = 1, antenna = 1, synch = 0, title = "My projection", quality = 1, Button(1)\}$$

The title of a dialog box can be set by adding a `Title()` statement to the `Dialog()` constructor. So, for example,

```julia
Dialog(
    Title("Hello World"),
    Button("OK"),
    Button("Cancel"));
```

produces a dialog box with “Hello World” in the title bar.

**Data column dialog boxes**

Column Dialog, a variant of the Dialog command, lets you prompt for column selections from the current data table or contextual data table, which must already be open.

For example:

```julia
dt=open("$SAMPLE_DATA/Big Class.JMP");
r=ColumnDialog(    
    Col Id=ColList("X, Treatment", Max Col(1)),
    Group =ColList("Group Factors"),
    Split =ColList("Y, Response"),
    w=ColList("Weight"),
    HList("Alpha",alpha=EditNumber(.05)));
```
This example returns a list similar to this one, depending on the user’s choices:

\[
\{\text{Col Id=\{name\}}, \text{Group=\{age\}}, \text{Split=\{weight\}}, w=\{}, \alpha=0.05, \text{Button(1)}\}
\]

For each destination list, a ColList clause must be a direct argument of ColumnDialog (not nested inside some other argument). An optional MaxCol(n) argument restricts the number of data columns that can be chosen to \(n\). The resulting list contains the “name” parenthesized list. Lists are always returned, although they can sometimes be empty lists. You may include as many as six ColList clauses.

Other dialog items permitted in a Dialog command are permitted in Column Dialog also, and have the same functionality. The OK, Cancel, and Remove buttons and the list of columns to choose from are both added automatically.

You can specify the minimum and maximum number of columns allowed in a column dialog box with the MaxCol parameter. You can also specify the modeling type of the columns that are allowed to be selected. You can set the width of the list using Select List Width(pixels) parameter. To set the width of the column list, use Width(pixels) inside the Col List() function.

For example, the following code generates a column dialog box that only allows the selection of exactly one numeric column.

\[
\text{rt_Dlg=ColumnDialog(}
\text{cv=ColList("Response To Test",MaxCol(1),MinCol(1),DataType(Numeric)))}
\]
Figure 7.27 Restricting Selection of Columns

The `DataType` choices are Numeric, Character, and RowState.

In addition, use the `Columns` specification to pre-fill some column selections. For example,

```python
dlg = ColumnDialog(
    xCols=ColList("X, Factors", Columns(:height)),
    yCols=ColList("Y, Response", Columns(:weight, :age))
);
```

assigns `height` to the X role and `weight` and `age` to the Y role.

Column arguments for platform launch scripts allow list arguments, so the following are all valid:

```python
Distribution(Y(height,weight))
Distribution(Y({height,weight}))  # equivalent
Distribution(Weight({}))         # empty specification causes usual launch dialog
```

**Details common to Dialog and Column Dialog**

**Arrangement**

JMP automatically creates a dialog box window of the appropriate size and shape and arranges items in the order specified.

Items can be grouped and aligned by using `HList` and `VList` containers. An `HList` top-aligns and spaces its contents in a horizontal row. `HListing` a pair of `VLists` produces a top-aligned, spaced pair of columns. A `VList` left-aligns and spaces its contents in a vertical column. `VListing` a pair of `HLists` produces a left-aligned, spaced pair of rows.

`Line Up` arranges items in the number of columns you specify. JMP automatically figures out the proper spacing.

```python
Dialog(
    VList(
        LineUp(2,
            "Set this value", variable=EditNumber(42),
```
Figure 7.28  Default Dialog Arrangements

Note that JMP does exert some control over OK and Cancel button positions to ensure that dialog boxes are consistent with what the operating system (Windows or Macintosh) expects. In certain cases, JMP will need to override your HList, VList, and LineUp settings for Button(OK) and Button(Cancel). Do not be alarmed if the result is slightly different from what you expect.

Assign dialogs to variables

If you assign the dialog function to a variable, JMP stores the list returned by the Dialog function in that variable. Calling that variable calls the list, not the Dialog.

```r
myValues=Dialog(HList(
    VList(
        "Lower Spec Limit", lsl=EditNumber(230)),
    VList(
        Button("OK"),
        Button("Cancel"))));
myValues; // returns the list of values
```

If you want to store the dialog function itself for later use, quote it with Expr.

```r
myDialog=expr(Dialog(HList(
    VList(
        "Lower Spec Limit", lsl=EditNumber(230)),
    VList(
        Button("OK"),
        Button("Cancel"))));
myDialog; // draws the dialog box to gather values
```

Unload results from dialogs

Both kinds of dialog boxes return lists. They do not set values directly.

```r
result=dialog(
    v list(
    line up (2,
    "Alpha (0-1)", a=edit number (0.05),
    "Sigma (0-5)", sd=edit number (1),
    "Effect (0-5)", eff=edit number (2),
    "Sample (2-100)", n=edit number (2)),
    h list(
        button("Cancel"),
    )
    )
```
When you're ready to use some of the values, you have to unload them from the list returned:

```plaintext
sd = result["sd"]; 
eff = result["eff"]; 
```

Use `EvalList` to evaluate an entire list of assignments all at once:

```plaintext
RemoveFrom(result,5); // since Button() is undefined 
EvalList(result); 
```

### Advanced example

An example for “Manipulating expressions,” p. 425 in the “Advanced Concepts” chapter, showed how to use JSL’s `SubstituteInto` operator to plug coefficients for a quadratic polynomial into the quadratic formula and then use the formula to calculate the roots of the polynomial. That example required specifying the coefficients as arguments to `SubstituteInto`.

You could instead collect the coefficients from the user through a Dialog:

```plaintext
myCoeffs=dialog(  
  vlist("Find the roots for the equation",  
        hlist(a=edit number(1),"*x^2 + ",  
              b=edit number(2),"*x + ",  
              c=edit number(1)," = 0"),  
        Button("OK"),  
        Button("Cancel"));
```

![Figure 7.29 Using a Dialog to Collect Information](image)

Now you just need to specify the appropriate members of the list `myCoeffs` as the arguments for `SubstituteInto`:

```plaintext
x={expr((-b + sqrt(b^2 - 4*a*c))/(2*a)),  
    expr ((-b - sqrt(b^2 - 4*a*c))/(2*a))}; 
substitute into(x,expr(a),mycoeffs["a"], expr(b), mycoeffs["b"], expr(c),  
    mycoeffs["c"]);
```

“Constructing display trees,” p. 240 discusses how to construct custom display windows. “Construct a custom platform,” p. 260 further develops this example into a complete customized platform that first presents this dialog box asking for coefficients, then finds the roots, and finally displays the roots along with a graph in a custom window.
Scripting the Script Editor

Even the script editor window is a display tree in JMP, which means you can write a JSL script to write and save another JSL script.

There is no New Script command. Instead, to open a new script window, you use the New Window command and then send it a message to tell it that it’s a script window:

```js
ww = New Window("Window Title", <<Script, "Initial Contents");
```

The last argument is optional. If you include a string, the new script window contains that string.

In the New Window example above, `ww` is a reference to the DisplayBox that is the entire window. To write to a script window, you first need to get a reference to the part of the display box you can write to, which is called a ScriptBox:

```js
ed = ww[scriptbox(1)];
```

Using the reference `ed`, you can add text, remove text, get the text that is already there.

```js
ed << get text();
"Initial Contents"
```

Use Set Text to set all the text in the script window. The following command removes the first line that you set to Initial Contents and changes it to `aaa=3;` followed by a return:

```js
ed << set text("aaa=3;\!N");
```

Use Append to add additional text to the end of the script window.

```js
ed << append text("bbb=1/10;");
ed << append text("\!Nccc=4/100;");
```

```js
ed << get line text(2);
ed << set line text(2, "bbb = 0.1;");
ed << get line count();
ed << get lines();
ed << reformat();
ed << run();
ww << close window(nosave);
```
You can run a script inside a graph, which draws inside the graph. You can do this with almost any graph from an analysis platform, or you can create your own new graphs. In both cases you are storing a JSL script inside a graph, and the script runs each time you display the graph.

For scripting 3-dimensional plots, see the chapter “Three-Dimensional Scenes,” p. 309.
## Contents

Adding Scripts to Graphs .................................................. 279
Adding a Legend to a Graph .............................................. 280
Creating New Graphs From Scratch .................................... 280
  Making changes to graphs .............................................. 281
Graphing Elements .......................................................... 282
  Plotting functions ...................................................... 282
  Getting the properties of a graphics frame ....................... 286
  Adding a Legend ......................................................... 286
Drawing lines, arrows, points, and shapes .......................... 287
  Lines ................................................................. 287
  Arrows .............................................................. 289
  Markers ............................................................. 290
  Pies and Arcs ......................................................... 291
  Regular Shapes: Circles, Rectangles, and Ovals ............... 293
  Irregular Shapes: Polygons and Contours ....................... 295
Adding text ................................................................. 297
Colors ........................................................................... 298
Transparency ................................................................. 300
Fill patterns ................................................................. 301
Line types .................................................................... 301
Drawing with pixels ......................................................... 301
Interactive graphs .......................................................... 302
  Handle ................................................................. 302
  MouseTrap ............................................................. 305
  Drag functions ......................................................... 306
Troubleshooting .............................................................. 307
Adding Scripts to Graphs

If you context-click (right-mouse-button) on the graphics frame, you can enter or paste JSL commands. The script usually will contain drawing commands that run in the context of the graphics frame. For example:

1. Open the Big Class.jmp data table.
2. Select Analyze > Fit Y by X.
3. Choose Height for X and Weight for Y, and click OK.
4. Right-click (Windows) or Control-click (Macintosh) inside the graph.
5. Select Customize from the context menu.
6. Click the plus button to add a new graphics script.
7. Type this text and click OK.
   
   text({55,160},"Hello World");

Now the graph has a text element at the graph’s x-coordinate 55 and y-coordinate 160.

Figure 8.1 Adding a Script to a Graph Interactively

By default, graphics scripts appear on top of the data. For example, add this line to the graph:

   Fill Color("Green"); Rect(57, 175, 65, 110, 1);

A solid green rectangle appears. The list of scripts shows the order in which each script is drawn, so the first item on the list is drawn first. If you want the rectangle behind everything, move it to the top. If you want it on top of everything, move it to the bottom. You can arrange all the scripts in a graph into the drawing order you prefer. Any new script is initially added directly after the item in the list that is selected.

**Tip:** To use a script that references a column name, use Column(colname) or a colon (:colname) to scope it properly.

**Hint:** To see the JSL for the above actions, select Script > Save Script to Script Window from the red triangle menu.
Adding a Legend to a Graph

Interactively, you add a legend using the **Row Legend** command. To accomplish this through JSL, send a **Row Legend** message to the display. In this message, specify which column you want to base the legend on, and whether the legend should affect colors and markers.

For example, using *Big Class*, submit the following JSL to turn on a legend based on the age column, setting both colors and markers by values in the age column.

```julia
biv = Bivariate( Y( :height), X( :weight) );
Report( biv )[Frame Box(1)] << Row Legend( "age", color(1), marker(1));
```

The `color()` and `marker()` arguments are optional. By default, they mimic the behavior of the dialog—colors are on by default, markers are off.

To use a continuous scale if your variable is nominal or ordinal, use `color(2)`.

Creating New Graphs From Scratch

Graphics scripts are set up within the **Graph Box** command within a **New Window** command.

```julia
New Window("window title", <Editable|Dialog>, Graph Box( named arguments,..., script));
```

There are two optional keywords after the window name. **Editable** treats the window like a Journal window (so report items can be dragged and dropped onto them). **Dialog** treats the window like a dialog window, so it shows up with a gray background on the Macintosh.

**Graph Box** takes named arguments for:

- `FrameSize(horizontal, vertical)`, // size in pixels
- `XScale(xmin, xmax), YScale(ymin, ymax)`, // range of x, y axes
- `X Axis(messages), Y Axis (messages)`, // the usual axis controls
- `Double Buffer` // for smooth animation
- `XName, YName` // names for x, y axes

Here is a script that plays a smoothed random walk around the frame. Because this uses random numbers, your display may differ.

```julia
New Window("Smoothed Random Walk",
Graph Box(FrameSize(200,200),XScale(-20,20),yScale(-20,20),
x = 0; y = 0; xi = 0; yi = 0;
For(i=0,i<2000,i++,
   xi = .9*xi + Random Normal()/10; yi = .9*yi + Random Normal()/10;
   xx = x + xi; yy = y + yi;
   xx=if(xx<-20,-20,xx>20,20,xx); yy=if(yy<-20,-20,yy>20,20,yy);
   line({x,y},{xx,yy}); x = xx; y = yy)));}
```
Making changes to graphs

You can also make changes to graphs through scripting. For example, create a window with a graph and get a reference to the report:

```julia
open("$SAMPLE_DATA/Big Class.jmp");
biv = Bivariate( Y( :weight ), X( :height ), Fit Line );
rbiv = biv<<report;
```

Using subscripting, you can send messages to any part of that report. Use Show Tree Structure to discover how to reference them. See “To examine a report’s display box tree,” p. 229 in the "DISPLAY TREES" chapter for details on the tree structure. See “Subscripts,” p. 231 in the "DISPLAY TREES" chapter on using subscripts. This line re-sizes the graph to 400 pixels by 400 pixels:

```julia
rbiv[frame box(1)]<<<<frame size(400,400);
```

To see a list of possible messages for any given display box object, use Show Properties. For example, here is a partial list of messages you can send to an axis:

```julia
rbiv[axis box(1)]
```

- **Axis Settings [Action]** (Bring up the Axis dialog to change various settings.)
- **Revert Axis [Action]** (Restore the settings that this axis had originally.)
- **Add Axis Label [Action]**
- **Remove Axis Label [Action]**
- **Save To Column Property [Action]** (Save the Axis settings as an Axis property in the data column associated with this axis.)
- **Set Width [Action]** [Scripting Only]
- **Axis [Subtable]** [Scripting Only]
  - **Scale [Enum]** {Linear, Log, Exp Prob, Weibull Prob, Logistic Prob, Frechet Prob, Lognormal Prob, Cube Root}
Min [Numeric]
Max [Numeric]
Inc [Numeric]
Tick Font [Action]
Interval [Enum] [Scripting Only] {Numeric, Year, Month, Week, Day, Hour, Minute, Second}
...

For example, to change the font of both axis labels (weight and height in the example above, which are both text edit boxes attached to the axis boxes) to 12-point, italic Arial Black:

```julia
rbiv[Text Edit Box( 1 )] << set font( "Arial Black" );
rbiv[Text Edit Box( 1 )] << set font style( "Italic" );
rbiv[Text Edit Box( 1 )] << set font size( 12 );
rbiv[Text Edit Box( 2 )] << set font( "Arial Black" );
rbiv[Text Edit Box( 2 )] << set font style( "Italic" );
rbiv[Text Edit Box( 2 )] << set font size( 12 );
```

---

**Graphing Elements**

Following are the commands you can use inside Graph Box statements. This chapter focuses on the JSL that is specific to graphing, but you can also use general script commands like For, While, and so on. The examples will give you some ideas.

### Plotting functions

A YFunction operator is used to draw smooth functions. The first argument is the expression to be plotted, and the second argument is the name of the X variable in the expression.

```julia
New Window("A",
    Graph Box(FrameSize(200,100),XScale(-10,10),yScale(-1,1),
        xName("x"),yName("Sine(x)"),
        YFunction(Sine(x),x)));
```

**Figure 8.3** Sine Wave

You can use For to overlap several sine waves:

```julia
New Window("A",
```
Graph Box(FrameSize(200,100),XScale(-10,10),yScale(-1,1),
For(i=1,i<=4,i+=.1,YFunction(Sine(x/i),x)));

Figure 8.4 Overlapping Sine Waves

Similarly, an XFunction is for drawing a plot where the symbol is varied on the Y variable.

New Window("B",
    Graph Box(FrameSize(100,200),XScale(-1,1),yScale(-10,10),
        For(i=1,i<=4,i+=.2,XFunction(Sine(y/i),y)));

Figure 8.5 Overlapping Sine Waves Along the X-Axis

ContourFunction is an analogous way to represent a three-dimensional function in a two-dimensional space. The final argument specifies the value(s) for the contour line(s), and it can be a value, an indexed range of values using ::, or a matrix of values.

New Window("Bird's eye view of the egg-carton function",
    Graph Box(FrameSize(300,300),XScale(-10,10),yScale(-10,10),
        pen color("black"); pen size(2);
        ContourFunction(Sine(y)+Cosine(x),x,y, (0::20)/5);
        pen color("red"); pen size(1);
        ContourFunction(Sine(y)+Cosine(x),x,y, (-20::0)/5)));
Normal Contour draws normal probability contours for *k* populations and two variables. The first argument is a scalar probability or a matrix of probability values for the contours, and subsequent arguments are matrices to specify means, standard deviations, and correlations. The mean and standard deviation matrices have dimension \(k \times 2\), and the correlation matrix should be \(k \times 1\), where the first row pertains to the first contour, the second row to the second contour, etc., and the first column is for \(x\) and the second column for \(y\). In other words:

\[
\text{Normal Contour}(\[\text{prob1, prob2, prob3, ...},
\text{xmean1 ymean1, xmean2 ymean2, xmean3 ymean3, ...},
\text{xsd1 ysd1, xsd2 ysd2, xsd3 ysd3, ...},
\text{xycorr1, xycorr2, xycorr3, ...}\]);
\]

For example, this script draws contours at probabilities 0.1, 0.5, 0.7, and 0.99 for two populations and two variables. The first population has \(x\) mean 0 and \(y\) mean 1, with standard deviation 0.3 along the \(x\) axis and 0.6 along the \(y\)-axis, and with correlation 0.5. The second has \(x\) mean 4 and \(y\) mean 6, with standard deviation 0.8 along the \(x\) axis and 0.4 along the \(y\)-axis, and with correlation 0.9.

```
new window("Normal contours",
   graph box(xscale(-5,10),yscale(-5,10),
   
```
normal contour([.1,.5,.7,.99],[0 1, 4 6],[.3 .6, .8 .4],[.5,.9]));

Figure 8.7 Normal Contour Function

Normal Contour is thus a generalized way to accomplish effects like Bivariate’s density ellipses, which are demonstrated to good effect with the Football sample data (just open the data table Football and run its stored Bivariate script).

Gradient Function

The syntax of the gradient function is

\[
\text{Gradient Function}(xname, yname, [zlow, zhigh], \text{zcolor}([\text{color, color}]), \\
<\text{XGrid}(\text{min}, \text{max}, \text{incr})>, <\text{YGrid}(\text{min}, \text{max}, \text{incr})>)
\]

This function fills a set of rectangles on a grid according to a color determined by the expression value as it crosses a range corresponding to a range of colors. To implement it, use the following syntax.

<table>
<thead>
<tr>
<th>GradientFunction()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>the expression to be contoured, which is a function in terms of the two variables that follow</td>
</tr>
<tr>
<td>xname, yname,</td>
<td>the two variable names used in the expression</td>
</tr>
<tr>
<td>[zlow zhigh],</td>
<td>the low and high expression values the gradient is scaled between</td>
</tr>
<tr>
<td>\text{ZColor}([\text{colorLow, colorHigh}])</td>
<td>the colors corresponding to the low value and the high value</td>
</tr>
<tr>
<td>&lt;\text{XGrid}(\text{min}, \text{max}, \text{incr}),&gt;</td>
<td>optional specification for the grid of values</td>
</tr>
<tr>
<td>&lt;\text{YGrid}(\text{min}, \text{max}, \text{incr}),&gt;</td>
<td></td>
</tr>
</tbody>
</table>

The \textbf{ZColor} values must be numeric codes, rather than names. You can use the color menu indices (0=black, 1=grey, 2=white, 3=red, 4=green, 5=blue etc.) found in “Colors,” p. 298.

The following example script uses the \textbf{Gradient} function, with the picture showing two frames of the animation.

\[
\text{phase} = 0.7;
\]
Scripting Graphs

Chapter 8

NewWindow("Gradient Function",
    a=Graph(FrameSize(400,400),XScale(-5,5),yScale(-5,5),DoubleBuffer,
    GradientFunction(phase*sine(x)*sine(y) + (1-phase)*cosine(x)*cosine(y),
    x,y,[-1 1],zcolor([0,2])));
    b = a[FrameBox(1)];
    for(i=1,i<=5,i++,
        for(phase=0,phase<1,phase+=0.05,b<<reshow;wait(0.01));
        for(phase=1,phase>0,phase-=0.05,b<<reshow;wait(0.01))
    );

Figure 8.8 Gradient Function

Getting the properties of a graphics frame

There are several functions that are useful for getting properties of an existing graphics frame:

- **H Size**  Returns the horizontal size of the graphics frame in pixels.
- **V Size**  Returns the vertical size of the graphics frame in pixels.
- **X Origin** Returns the \( x \)-value for the left edge of the graphics frame.
- **Y Origin** Returns the \( y \)-value for the bottom edge of the graphics frame.
- **X Range** Returns the distance from the left to right edge of the display box. For example, 
  \( \text{X Origin}() + \text{X Range}() \) is the right edge.
- **Y Range** Returns the distance from the bottom to top edges of a display box. For example, 
  \( \text{Y Origin}() + \text{Y Range}() \) is the top edge.

Adding a Legend

You can add a legend to a graph, using the **Row Legend** command. The following example uses the Fitness.jmp sample data file, and sets colors and markers based on the Age column, and adds a legend to the plot.
biv = Bivariate(Y(:Oxy), X(:Runtime)); //generate a scatterplot
Report(biv)[Frame Box(1)] << Row Legend (:Age, color(1), marker(1),
window(1));

The `marker()` parameter has binary arguments. Using a 1 is the same as checking the corresponding
box in the Row Legend dialog. Using a 0 is equivalent to leaving the box unchecked.

Color is a special case, as it has three settings:

- Color(0) turns the color legend on.
- Color(1) turns the color legend off.
- Color(2) sets the continuous scale option.

---

### Drawing lines, arrows, points, and shapes

#### Lines

Line draws lines from point to point.

```cpp
new window("Five-point star",
graph box(framesize(300, 300), xscale(-1.1, 1.1), yscale(-1.1, 1.1),
line(
    {cos(1*pi()/10), sin(1*pi()/10)},
    {cos(9*pi()/10), sin(9*pi()/10)},
    {cos(17*pi()/10), sin(17*pi()/10)},
    {cos(5*pi()/10), sin(5*pi()/10)},
    {cos(13*pi()/10), sin(13*pi()/10)},
    {cos(1*pi()/10), sin(1*pi()/10)}
));
```
Figure 8.9 Using Lines to Draw a Star

You can either specify the points in two-item lists as demonstrated above or as matrices of \(x\) and then \(y\) coordinates. Matrices are flattened by rows, so you can use either row or column vectors, as long as you have the same number of elements in each matrix. The following would be equivalent:

- \(\text{Line}([1,2], [3,0], [2,4]); // several \{x,y\} lists\)
- \(\text{Line}([1 3 2],[2 0 4]); // row vectors\)
- \(\text{Line}([1,3,2],[2,0,4]); // column vectors\)
- \(\text{Line}([1 3 2],[2,0,4]); // one of each\)

Thus, the star example could also be drawn this way. Note that it must use full \text{Matrix}(...) notation rather than \([\ldots]\) shorthand since the entries are expressions.

\[
\text{new window("Five-point star",}
\text{graph box(framesize(300, 300), xscale(-1.1, 1.1), yscale(-1.1, 1.1)},
\text{line(}
\text{matrix({ // the x coordinates}
\text{cos(1*\pi()/10), cos(9*\pi()/10), cos(17*\pi()/10),}
\text{cos(5*\pi()/10), cos(13*\pi()/10), cos(1*\pi()/10))}),
\text{matrix({ // the y coordinates}
\text{sin(1*\pi()/10), sin(9*\pi()/10), sin(17*\pi()/10),}
\text{sin(5*\pi()/10), sin(13*\pi()/10), sin(1*\pi()/10))})});
\]

\text{HLine} draws a horizontal line across the graph at the \(y\)-value you specify. Similarly \text{VLine} draws a vertical line down the graph at the \(x\)-value you specify. Both commands support drawing multiple lines by using a matrix of values in the \(y\) argument. These are illustrated in the example under "MouseTrap," p. 305.
Arrows

Similarly, `Arrow` draws an arrow from the first point to the second. The default arrowhead is scaled to 1 plus the square root of the length of the arrow. To set the length of the arrowhead, add an (optional) first argument, specifying the length of the arrowhead, in pixels.

As an example, the following script draws arrowheads with the default length.

```plaintext
New Window("Hurricane",
  Graph Box(FrameSize(100,100),XScale(-100,100),yScale(-100,100),
  for(r=35,r<100,r+=20,
    ainc = 2*pi()*3/r;
    for(a=0,a<2*pi(),a+=ainc, 
      x = r*cosine(a); y = r*sine(a);
      aa = a + ainc*45/r; rr = r - r/6;
      x2 = rr*cosine(aa);
      y2 = rr*sine(aa);
      arrow({x,y},{x2,y2});)));)
```

Figure 8.10 Drawing Arrows

This script compares drawing with a specified length (19 pixels) and drawing with the default arrow head size.

```plaintext
New Window( "Arrow Heads", Graph Box( 
  Frame Size( 300, 300 ), X Scale( 0, 100 ), Y Scale( 0, 220 ),
  x = 10; y1 = 10; y2 = y1 + 10;
  For( i = 1, i < 10, i++,
    Pen Color( "Red" );
    Arrow( {x, y1}, {x, y2} );
    y2 += 10; y1 += 100; y2 += 100;
    Pen Color( "Blue" );
    Arrow( 20, {x, y1}, {x, y2} );
  );
  x += 10; y1 -= 100; y2 -= 100;
  Text Color( "Red" );
  Text( {10, 80}, "Without Length Arg" );
```

```
Text Color( "Blue" );
Text( {10, 200}, "With Length Arg" );
Figure 8.11 Arrowhead Sizes

As with Line, you can either specify the points in two-item lists as demonstrated above or as matrices of \( x \) and then \( y \) coordinates.

**Markers**

Marker draws a marker of the type you specify (1–15) in the first argument at the point you specify in the second argument. Marker Size scales markers from 0–6 (dot–XXXL).

```
ymax=20;
New Window("The markers",
    Graph Box(FrameSize(300,400),XScale(-2,ymax-5),yScale(-2,ymax+3),
    for(j=1, j<7, j++,
        marker size(j);
    for(i=0,i<(ymax+1),i++,
        marker(i,{j*2,i});
        text({0,i},i);
        text({j*2,ymax+2},j)))));
```
Chapter 8
Scripting Graphs

Drawing lines, arrows, points, and shapes

Figure 8.12 Drawing Markers

You can also include a row state argument before, after, or instead of the marker ID argument. By using Combine States, you can set multiple row states inside Marker. Try substituting each of these lines in the graph script above:

```
marker(i, color state(i), {j*2, i});
marker(color state(i), i, {j*2, i});
marker(combine states(colorstate(i),markerstate(i),hiddenstate(i)),{j*2, i});
```

Again, points can also be specified as matrices of x and then y coordinates.

Pies and Arcs

Pie and Arc draw wedges and arc segments. The first four arguments are $x_1, y_1, x_2,$ and $y_2,$ the coordinates of the rectangle to inscribe. The last two arguments are the starting and ending angle in degrees, where 0 degrees is 12 o’clock and the arc or slice is drawn clockwise from start to finish.

```
New Window( "Pies and Arcs", Graph Box(
    framesize( 400, 400 ), X Scale( 0, 9 ), Y Scale( 0, 9 ),
    Fill Color( "Black" ), // top left
    Pie( 1.1, 7.9, 3.9, 5.1, 45, 270 ),
    Text( erased, {1.75, 6}, "1,8,4,5,45,270" ),
)
```

You can also include a row state argument before, after, or instead of the marker ID argument. By using Combine States, you can set multiple row states inside Marker. Try substituting each of these lines in the graph script above:

```
marker(i, color state(i), {j*2, i});
marker(color state(i), i, {j*2, i});
marker(combine states(colorstate(i),markerstate(i),hiddenstate(i)),{j*2, i});
```

Again, points can also be specified as matrices of x and then y coordinates.
Arc( 1, 8, 4, 5, 280, 35 ),
Fill Color( "Red" ), // top right
Pie( 7.9, 7.9, 5.1, 5.1, 270, 360 ),
Text( erased, {5.75, 6}, "8,8,5,5,270,360" ),
Arc( 8, 8, 5, 5, 370, 260 ),
Fill Color( "BlueCyan" ), // bottom left
Pie( 1.1, 1.1, 3.9, 3.9, 50, 360 ),
Text( erased, {1.75, 2}, "1,1,4,4,50,360" ),
Arc( 1, 1, 4, 4, 370, 40 ),
Fill Color( "Purple" ), // bottom right
Pie( 7.9, 1.1, 5.1, 3.9, 270, 45 ),
Text( erased, {5.75, 2}, "8,1,5,4,270,45" ),
Arc( 8, 1, 5, 4, 55, 260 )
);

Figure 8.13 Drawing Pies and Arcs
Regular Shapes: Circles, Rectangles, and Ovals

Circles

Circle draws a circle with the center point and radius given. Subsequent arguments specify additional radii.

```
New Window("Circles",
    Graph Box(framesize(200, 200),
    Circle({50, 50}, 10, 12, 25)));
```

Figure 8.14 Drawing Circles

Note that a circle is always a circle, even if you re-size the graph to a different aspect ratio. If you want your circle to change aspect ratios (in other words, cease being a circle) when the graph is resized, use an oval instead.

If you don’t want your circle to resize if the graph is resized, specify the radius in pixels instead:

```
New Window("Circles",
    Graph Box(framesize(200, 200),
    Circle({50, 50}, PixelRadius(10), PixelRadius(12), PixelRadius(25))));
```

Rectangles

Rect draws a rectangle from the diagonal coordinates you specify. The coordinates can be specified either as four arguments in order (left, top, right, bottom), or as a pair of lists ([left, top], [right, bottom]).

```
new window("Rectangles",
    graph box(framesize(200, 200),
    pen color(1); rect(0,40,60,0);
    pen color(3); rect(10,60,70,10);
    pen color(4); rect(50,90,90,50);
    pen color(5); rect(0,80,70,70)));
```

Rect has an optional fifth argument, fill. Specify a zero to get an unfilled rectangle, and a one to get a filled rectangle. The rectangle is filled with the current fill color. The default value for fill is 0.
Any negative fill argument produces an unfilled frame inset by one pixel:

```plaintext
ew window("Framed rectangle",
   graph box(framesize(200,200),
      rect(0,40,60,0, -1)))
```

Ovals

`Oval` draws an oval inside the rectangle given by its $x1, y1, x2, y2$ arguments:

```plaintext
new window("Ovals",
   graph box(framesize(200,200),
      pen color(1); oval(0,40,60,0);
      pen color(3); oval(10,60,70,10);
      pen color(4); oval(50,90,90,50);
      pen color(5); oval(0,80,70,70)))
```

`Oval` also uses the optional fifth argument, `fill`. Specify a zero to get an unfilled rectangle, and a one to get a filled oval. The oval is filled with the current fill color. The default value for `fill` is 0.

Figure 8.15 shows rectangles and ovals, drawn both filled and unfilled. Notice that filled rectangles do not have outlines, while ovals do. If you want a filled rectangle with an outline, you must draw the filled rectangle, and then draw an unfilled rectangle with the same coordinates.
Irregular Shapes: Polygons and Contours

Polygons

`Polygon` works similarly to `Line`, connecting points but also returning to the first point to close the polygon and filling the resulting area. You can specify the points as individual points in two-item lists (as shown for `Marker`, above) or as matrices of `x` and then `y` coordinates. Matrices are flattened by rows, so you can use either row or column vectors, as long as you have the same number of elements in each matrix. First set up the matrices of points, then call them inside `Polygon()`.

```plaintext
gCoordX=[25, 23.75, 22.5, 21.25, 21.25, 22.5, 23.75, 25.625, 27.5, 18.75, 12.5, 6.25, 2.5, 1.25, -1.25, 3.125, 6.25, 12.5, 18.75, 25, 29.375, 34.375, 37.5, 38.75, 40.625, 42.5, 43.125, 42.5, 41.25, 38.75, 43.75, 50, 56.25, 60.625, 63.75, 65.625, 62.5, 56.25, 50, 45, 37.5, 32.5, 28.75, 27.5, 26.25, 25.625, 25];
```
Scripting Graphs
Chapter 8

Drawing lines, arrows, points, and shapes

gCoordY=[-2, 2, 5, 10, 15, 20, 25, 30, 33, 34, 35, 37.5, 40, 41, 43.5, 41, 40, 39, 40, 42, 45, 50, 55, 60, 64, 60, 55, 50, 47, 42, 43.5, 43, 42, 40, 38, 36, 37, 36, 35, 30, 25, 20, 15, 10, 5, 2];
new window("The JMP man",
graph box(framesize(300,300), xscale(-10,80), yscale(-10,80),
p_pen color("black"); fill color("blue");
polygon(gCoordX, gCoordY);
Fill Color("black");
Circle({18, 58},9,"FILL")));;

Figure 8.16 Drawing a Polygon

A related command, In Polygon, tells whether a given point falls inside the polygon specified. This code checks some points from the JMP man pictured in Figure 8.16:

In Polygon(0,60, GcoordX, GCoordY); //returns 0
In Polygon(30,38, GcoordX, GCoordY); //returns 1

Or you can add In Polygon to the JMP man script. Run this script, then click various locations in the picture and watch the Log window.

new window("The JMP man",
graph box(framesize(300,300), xscale(-10,80), yscale(-10,80),
p_pen color("black"); fill color("black");
polygon(gCoordX, gCoordY);
mousetrap({},print(if(in polygon(x,y,gCoordX,gCoordY),"in","out")))));}
Contours

Contour draws contour lines using a grid of coordinates, all within the context of a graphics script. Its syntax is

```
Contour(xVector, yVector, zGridMatrix, zContour, <zColors>);
```

Given an \( n \) by \( m \) matrix \( z\text{GridMatrix} \) of values on some surface, defined across the \( n \) values of \( x\text{Vector} \) by the \( m \) values of \( y\text{Vector} \), this function draws the contour lines defined by the values in \( z\text{Contour} \) in the colors defined by \( z\text{Colors} \).

```plaintext
// testContour
x = (-10::10)/5;
y = (-12::12)/5;
grid = j(21,25,0);
z = [-.75, -.5, -.25,0,.25,.5,.75];
zcolor = [3,4,5,6,7,8,9];
for(i=1,i<=21,i++,for(j=1,j<=25,j++,grid[i,j]=sin((x[i])^2+(y[j])^2)));
show(grid);
newWindow("Hat",GraphBox(xScale(-2,2),yScale(-2.4,2.4),
    Contour(x,y,grid,z,zcolor)));
```

Figure 8.17 Drawing Contour Lines

Adding text

You can use Text to draw text at a given location. The point and text can be in any order and repeated. You can precede the point and text with an optional first argument, Center Justified, Right Justified, Erased, Boxed, Counterclockwise, or Clockwise. Erased is for “erasing” whatever would otherwise obscure the text in a graph—it paints a background-colored rectangle behind the text. In the example below, notice how the erased text appears inside a white box over the green Rect. The other effects are self-explanatory.

```plaintext
mytext=new window("Text",
    graph box(framesize(200,200), yscale(0,15), xscale(0,10),
```
text size(9); text color("blue");
text({5,1},"Left Justified");
text(Center Justified,{5,2},"Center Justified");
text(Right Justified,{5,3},"Right Justified");
fill color(4); rect(5,8,9,5,1);
text(Erased,{6,6},"Erased");
text(Boxed,{6,10},"Boxed");
text(Clockwise,{4,10},"Clockwise");
text(Counterclockwise,{3,5},"Counterclockwise");

Figure 8.18 Drawing Text in a Graph Box

There is a variant of the text function that draws a string inside the rectangle specified by four coordinates specified as arguments, wrapping as needed. The syntax is

    text( {left, top, right, bottom}, string)

Colors

Five commands control colors. Fill Color sets the color for solid areas, Pen Color for lines and points, Back Color for the background of text (similar to the box around the erased text above), Background Color for the graph's background color, and Font Color for added text.

Fill colors preempt pen colors for drawn shapes—you don't get both, as in some drawing packages (to get that effect, draw two shapes, one with fill and one without). A color can be chosen with a single numeric argument, or a color name in quotation marks, or an RGB value. The standard colors can be chosen with numbers 0–15 (both 0 and 15 are black) or by their names.
Chapter 8

Scripting Graphs

Drawing lines, arrows, points, and shapes

Table 8.1 Standard JMP colors

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Demonstration script to try</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Gray</td>
<td><code>mygraph=New Window(&quot;The JMP colors&quot;, Graph Box(FrameSize(300,300), XScale(0, ymax), yScale(0, ymax+2), for(i=0, i&lt;ymax, i++, If( colors[i + 1] == &quot;White&quot;, Fill Color( 65 ); Rect( 0, i + 1 + .5, 15, i + 1 - .5, 1 )); pen color(colors[i+1]); text color(colors[i+1]); hline(i+1); text({2, i + 1},i); text({5,i+1},colors[i+1])););</code></td>
</tr>
<tr>
<td>2</td>
<td>White</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Red</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Green</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Blue</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Orange</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>BlueGreen</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Purple</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Yellow</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Cyan</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Magenta</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>YellowGreen</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>BlueCyan</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Fuchsia</td>
<td></td>
</tr>
</tbody>
</table>

Larger numbers cycle through shading variations of the same color sequence. A script demonstrating this appears under “Colors and markers,” p. 165 in the “Data Tables” chapter. Values outside the range 0–84 are not accepted.

Table 8.2 How numbers map to colors

<table>
<thead>
<tr>
<th>Number</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>16–31</td>
<td>dark shades</td>
</tr>
<tr>
<td>32–47</td>
<td>light shades</td>
</tr>
<tr>
<td>48–63</td>
<td>very dark shades</td>
</tr>
<tr>
<td>64–79</td>
<td>very light shades</td>
</tr>
<tr>
<td>80–84</td>
<td>shades of gray, light to dark</td>
</tr>
</tbody>
</table>

If you prefer to use RGB values, type a list with the percentages for each color in red, green, blue order.

```
pen color({.38,.84,.67}); // a lovely teal
```

RGB Color and Color to RGB convert color values between JMP color numbers and the Hue-Lightness-Saturation system. For example, to find the RGB values for JMP color 3 (red):

```
Color to RGB(3);
{0.937254901960784, 0.227450980392157, 0.36078431372549}
```

Likewise, HLS Color and Color to HLS convert color values between JMP color numbers and the Hue-Lightness-Saturation system.

Finally, Heat Color returns the JMP color that corresponds to a value in any color theme that is supported by Cell Plot. The syntax is:

```
Heat Color(n, "<"theme")
```

The theme message is optional, and the default value is "Blue to Gray to Red".
Transparency

In a graphics environment (like a Frame Box), use the Transparency function to set the level of transparency. The argument, alpha, can be any number between zero and one. The value 0 means clear and drawing has no effect, while the value 1 means completely opaque and is the usual drawing mode. Intermediate values build semi-transparent color layers on top of what has already been drawn below it. The following example script illustrates transparency with rectangles.

```plaintext
New Window( "Transparency", 
    Graph Box(framesize( 200, 200 ),
        Pen Color( "gray" ); Fill Color( "gray" );
        Transparency( 0.25 );
        Rect( 0, 40, 60, 0, 1 );

        Pen Color( "red" ); Fill Color( "red" );
        Transparency( 0.5 );
        Rect( 10, 60, 70, 10, 1 );

        Pen Color( "green" ); Fill Color( "green" );
        Transparency( 0.75 );
        Rect( 50, 90, 90, 50, 1 );

        Pen Color( "blue" ); Fill Color( "blue" );
        Transparency( 1 );
        Rect( 0, 80, 70, 70, 1 );
    );
)
```

Figure 8.19 Transparency and Rectangles
Fill patterns

The Fill Pattern function has been deprecated and is now obsolete. It can be present in a script without causing errors, but will have no effect.

Line types

You can also control Line Style by number (0–4) or name (Solid, Dotted, Dashed, DashDot, DashDotDot).

linestyles = {"Solid", "Dotted", "Dashed", "DashDot", "DashDotDot"};
New Window( "The line styles",
    Graph Box(FrameSize( 200, 200 ),
        X Scale( -1, 5 ),
        Y Scale( -1, 5 ),
        For( i = 0, i < 5, i++,
            Line Style( i );
            H Line( i );
            Text( {0, i + .1}, i ); Text( {1, i + .1}, linestyles[i + 1] ))));

Figure 8.20  Line Styles

To control the thickness of lines, set a Pen Size and specify the line width in pixels. The default is 1 for single-pixel lines. For printing, think of Pen Size as a multiplier for the default line width, which varies according to your printing device.

pen size(2); //double-width lines

Drawing with pixels

You can also draw using pixel coordinates. First you set the Pixel Origin in terms of graph coordinates, and then use Pixel Move To or Pixel Line To commands in pixel coordinates relative to that origin. The main use for Pixel commands is for drawing custom markers that do not vary with

Figure 8.20  Line Styles
the size or scale of the graph. You can store a marker in a script and then call it within any graph. This example uses `Function` to store pixel commands in a script with its own parameters, \( x \) and \( y \).

```plaintext
ballotBox=function({x,y},
    pixelOrigin(x,y);PixelMoveTo(-5,-5);PixelLineTo(-5,5);
    PixelLineTo(5,-5);PixelLineTo(-5,-5);PixelLineTo(5,5);
    PixelLineTo(-5,5);PixelMoveTo(5,5);PixelLineTo(5,-5));
new window("Custom markers",
    graph box(framesize(200,200),
        ballotBox(10,10); ballotBox(15,90); ballotBox(20,50);
        ballotBox(80,50); ballotBox(60,70));
```

![Drawing Custom Markers](image)

**Figure 8.21** Drawing Custom Markers

---

**Interactive graphs**

`Handle` and `MouseTrap` are functions for making interactive graphs that respond to clicking and dragging. `Handle` lets you parametrize a graph by adding a handle-marker that can be dragged around with the mouse, executing the graph’s script at each new location. `MouseTrap` is similar but it takes its parameters from the coordinates of just a click, without using a draggable handle. The main difference is that `Handle` only catches mousedowns at the handle-marker’s location, but `MouseTrap` catches mousedowns at any location.

Another approach is to place buttons or slider controls outside the graph with `Button Box`, `Slider Box`, or `Global Box`.

**Handle**

`Handle` places a marker at the coordinates given by the initial values of the first two arguments and draws the graph using the initial values of the parameters. You can then click and drag the marker to move the handle to a new location. The first script is executed at each mousedown to update the graph
dynamically, according to the new coordinates of the handle. The second script (optional, and not used here) is executed at each mouseup, similarly; see the example for “MouseTrap,” p. 305.

// Normal Density
mu = 0; sigma = 1; rsqrt2pi = 1/sqrt(2*pi());
New Window("Normal Density",
Graph Box(FrameSize(500,300), XScale(-3,3), yScale(0,1), Double Buffer,
YFunction(Normal Density((x-mu)/sigma)/sigma, x);
Handle(mu,rsqrt2pi/sigma,mu=x;sigma=rsqrt2pi/y);
text({1,.7},"mu ",mu,{1,.65},"sigma ",sigma));

In the sample Scripts folder, you can find scripts for showing the Beta Density, Gamma Density, Weibull Density, and LogNormal Density. The output for the normal is show below, both at the initial parameter values, and after the handle has been moved sideways. A “multiple-exposure” screen shot is the best we can do here, so be sure to try this yourself!

Figure 8.22 Normal Density Example for Handle

To avoid errors, be sure to set the initial values of the handle’s coordinates, as in the first line of this example.

If you want to use some function of a handle’s coordinates, such as in the normal density example, you should adjust the arguments for Handle. Otherwise, the handle marker would run away from the mouse. For example:

YFunction(a*x^b);
handle(a,b,a=2*x,b=y)

Suppose you drag the marker from its initial location to (3,4). The parameter \(a\) is set to 6 and \(b\) to 4, the graph is redrawn as \(Y = 6x^4\), and the handle is now drawn at (6,4)—several units away from the mouse. To compensate, you would adjust the first argument to handle, e.g.

handle(a/2,b,a=2*x;b=y)

To generalize, suppose you define the Handle parameters as functions of the handle’s \((x, y)\) coordinates, e.g., \(a=f(x); b=g(y)\). If \(f(x)=x\) and \(g(y)=y\), then you would specify simply \(a, b\) as the first two
arguments. If not, you would solve $a = f(x)$ for $x$ and solve $b = g(y)$ for $y$ to get the appropriate arguments.

You can use other functions to constrain Handle. For example, here is an interactive graph to demonstrate power functions that uses \texttt{Round()} to prevent bad exponents and to keep the intercepts simple.

```plaintext
a=3; b=2;
New Window("Intercepts and powers",
 Graph Box(FrameSize(200,200),XScale(-10,10),yScale(-10,10),
 YFunction(round(b)+x^round(a),x);handle(a,b, a=x;b=y);
 text({a,b}," Move me");
 text({-9,9},"y=",round(b)," + x^",round(a)));
```

**Figure 8.23** Intercepts and Powers Example for Handle

Handle and \texttt{For} can be nested for complex graphs.

```plaintext
a=5; b=5;
New Window("powers",
 Graph Box(FrameSize(200,200),XScale(-10,10),yScale(-10,10),Double Buffer,
 for(i=0,i<1.5,i+=.2,
   pen color(1+10*i);
   text color(1+10*i);
   YFunction(i*x^round(a),x);
   Handle(a,b,a=x;b=y);
   h=9-10*i;
   text({-9,h},b,"*i*x^",round(a),", i=",i)));
```

You can use more than one handle in a graph:

```plaintext
amplitude = 1; freq = 1; phase = 0;
NewWindow("Sine Wave",
 Graph Box(FrameSize(500,300),XScale(-5,5),yScale(-5,5),Double Buffer,
 YFunction(amplitude*sin(x/freq+phase),x);
 Handle(freq,amplitude,freq=x;amplitude=y);
 Handle(phase,.5,phase=x);
```

Chapter 8
Scripting Graphs
Interactive graphs

MouseTrap

MouseTrap takes parameters for a graph from the coordinates of a mouse click. The first script is executed after each mousedown and the second script after each mouseup to update the graph dynamically, according to the new coordinates of the handle. As with Handle, it is important to set the initial values for the MouseTrap’s coordinates. If you include both MouseTraps and Handles in a graph, put the Handles before the MouseTraps so they have a chance to catch clicks before a MouseTrap does.

This example uses both MouseTrap and Handle to draw a three-dimensional function centered on the MouseTrap coordinates, where the single contour line takes its value from a Handle.

x0=0; y0=0; z0=0;
New Window("Viewing a 3D function in Flatland", 
Graph Box(FrameSize(300,300), XScale(-5,5), yScale(-5,5), DoubleBuffer, 
ContourFunction(exp(-(x-x0)^2)*exp(-(y-y0)^2)*(x-x0), x, y, z0/10);
handle(-4.5, z0, z0=round(y*10)/10); // get the z-cut vals from a handle
vline(-4.5); text size(9); text(Counterclockwise,{-4.6,-4},
"Drag to set the z-value for contour cut: z = \" || char(z0/10));
markersize(2); marker(2, {x0, y0});
mousetrap(x0=x; y0=y); //set the origin to the click-point
text({-4.25,-4.9},"Click any location to set the function's centerpoint.");

You might use MouseTrap to collect points in a data table, such as for visually interpolating points in a graph. Here’s an example illustrating a script that could be adapted and added to a data plot (such as a scatterplot from Fit Y by X) for that purpose:

dt = new Table("dat1");
Current Data Table(dt);
NewColumn("XX", Numeric);
NewColumn("YY", Numeric);
x=0; y=0;
add point = expr(
dt<<addRows(1);
row()=nrow();
:xx = x;
:yy = y);
NewWindow("Add Points"," 
Graph Box(FrameSize(500,300), XScale(-5,5), yScale(-5,5),
for each row(marker({xx, yy}));
MouseTrap({}, add point));

Notice that the first script argument is empty. At mousedown, nothing happens. The second script, add point, is executed at mouseup to add a data point. This means that if you click, drag, and release, the point that is added to your data set is the point where you let go of the mouse button, not the point where you pressed it down.
Drag functions

There are five Drag functions to perform similar functions to Handle and MouseTrap but with more than one point at a time. For \( n \) coordinates in matrices listed as the first two arguments:

- **Drag Marker** draws \( n \) markers.
- **Drag Line** draws a connected line with \( n \) vertices and \( n-1 \) segments.
- **Drag Rect** draws a filled rectangle using the first two coordinates, ignoring any further coordinates.
- **Drag Polygon** draws a filled polygon with \( n \) vertices.
- **Drag Text** draws a text item at the coordinates, or if there is a list of text items, draws the \( i \)th list item at the \( i \)th \((x,y)\) coordinate. If there are fewer list items than coordinate pairs, the last item is repeated for remaining points.

The syntax for these commands:

\[
\begin{align*}
dragMarker & \ (xMatrix, yMatrix, dragScript, mouseupScript) \\
dragLine & \ (xMatrix, yMatrix, dragScript, mouseupScript) \\
dragRect & \ (xMatrix, yMatrix, dragScript, mouseupScript) \\
dragPolygon & \ (xMatrix, yMatrix, dragScript, mouseupScript) \\
dragText & \ (xMatrix, yMatrix, "\text", dragScript, mouseupScript)
\end{align*}
\]

They all must have L-value arguments for the coordinates, in other words literal matrices or names of matrix values that will be modified if you click a vertex and drag it to a new position. The script arguments are optional, and behave the same as with Handle; however, there is no \( x \) nor \( y \) that is modified as in Handle.

The Drag operators are ways to display data that the user can adjust and then capture the adjusted values. Consider the earlier script to draw the JMP man. Drag Polygon makes it possible to draw an editable JMP man; using a matching Drag Marker statement makes the vertices more visible. And, similar to the Mouse Trap example, you can save the new coordinates to a data table. Notice how and :: operators avoid ambiguity among matrices and data table columns with the same names.

You could just as easily put storepoints in the fourth argument of Drag Polygon or Drag Marker, but that would create a data table after each drag, and you probably just want a single data table when you're done. Regardless, the values in \( gCoordX \) and \( gCoordY \) update with each drag.

```javascript
::i=1;
storepoints=expr(
    mydt= new Table("My coordinates_"||char(i)); i++;
    NewColumn("GCoordX",Numeric);
    NewColumn("GCoordY",Numeric);
    mydt<<add rows(nrow(GcoordX));
    :GCoordX<<values(::GcoordX);
    :GCoordY<<values(::GcoordY));

::GcoordX=[25, 23.75, 22.5, 21.25, 21.25, 22.5, 23.75, 25.625, 27.5, 18.75, 12.5, 6.25, 2.5, 1.25, -1.25, 3.125, 6.25, 12.5, 18.75, 25, 29.375, 34.375, 37.5, 38.75, 40.625, 42.5, 43.125, 45.25, 41.25, 38.75, 43.75, 50, 56.25, 60.625, 63.75, 65.625, 62.5, 56.25, 50, 47, 42, 37.5, 32.5, 28.75, 27.5, 26.25, 25.625, 25];
::GcoordY=[-2, 2, 5, 10, 15, 20, 25, 30, 33, 34, 35, 37.5, 40, 41, 43.5, 41, 40, 39, 40, 42, 45, 50, 55, 60, 64, 60, 55, 50, 45, 42, 43.5, 43, 42, 40, 38, 36, 37, 36, 35, 30, 25, 20, 15, 10, 5, 2];
```
new window(“Redraw the JMP Man!”,
    vlistbox(
        graph box(framesize(300,300), xscale(-10,80),yscale(-10,80),
            fill color(“blue”);
            dragPolygon(GcoordX, GCoordY);
            pen color(“gray”);
            dragMarker(GcoordX, GCoordY);
            Fill Color( {0, 0, 0});
            Circle({18, 58},9,"FILL"),
            buttonbox("Store current vertices in a new data table",storepoints))));
Perhaps you think the JMP Man needs to lose some weight. Here is how he looks before and after some judicious vertex-dragging. Clicking the button after re-reshaping the JMP Man executes the storepoints script to save his new, slender figure in a data table of coordinates.

Figure 8.24 Redraw the JMP Man

This example uses two operators discussed under “Constructing display trees,” p. 240 in the “Display Trees” chapter:

- Button Box, which creates controls outside the graph,
- V List Box, which glued the graph box and the button box together in the same graph window.

Troubleshooting

If your interactive graphs don’t work as expected, make sure that you supply initial values for the Handle or MouseTrap coordinates (and other globals as needed), and that the values make sense for the graph.
JSL includes commands for scripting three-dimensional scenes derived from OpenGL. Although not a complete OpenGL implementation, JSL’s 3-D scene commands enable complex, interactive plots to be constructed and viewed. The Surface Plot platform in JMP is built using JSL scene commands.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>About JSL 3-D Scenes</td>
<td>311</td>
</tr>
<tr>
<td>JSL 3-D Scene Basics</td>
<td>311</td>
</tr>
<tr>
<td>Scene Boxes</td>
<td>311</td>
</tr>
<tr>
<td>Setting the Viewing Space</td>
<td>314</td>
</tr>
<tr>
<td>Setting Up a Perspective Scene</td>
<td>315</td>
</tr>
<tr>
<td>Setting up an Orthographic Scene</td>
<td>316</td>
</tr>
<tr>
<td>Changing the View</td>
<td>316</td>
</tr>
<tr>
<td>The Translate Command</td>
<td>316</td>
</tr>
<tr>
<td>The Rotate Command</td>
<td>316</td>
</tr>
<tr>
<td>The Look At Command</td>
<td>318</td>
</tr>
<tr>
<td>The ArcBall</td>
<td>319</td>
</tr>
<tr>
<td>Graphics Primitives</td>
<td>321</td>
</tr>
<tr>
<td>Primitives Example</td>
<td>324</td>
</tr>
<tr>
<td>Controlling the Appearance of Primitives</td>
<td>326</td>
</tr>
<tr>
<td>Other uses of Begin and End</td>
<td>331</td>
</tr>
<tr>
<td>Drawing Spheres, Cylinders, and Disks</td>
<td>331</td>
</tr>
<tr>
<td>Construction</td>
<td>331</td>
</tr>
<tr>
<td>Lighting</td>
<td>332</td>
</tr>
<tr>
<td>Drawing Text</td>
<td>332</td>
</tr>
<tr>
<td>Using Text with Rotate and Translate</td>
<td>333</td>
</tr>
<tr>
<td>Using the Matrix Stack</td>
<td>334</td>
</tr>
<tr>
<td>Lighting and Normals</td>
<td>337</td>
</tr>
<tr>
<td>Creating Light Sources</td>
<td>337</td>
</tr>
<tr>
<td>Lighting Models</td>
<td>338</td>
</tr>
<tr>
<td>Normal Vectors</td>
<td>339</td>
</tr>
<tr>
<td>Shading Model</td>
<td>339</td>
</tr>
<tr>
<td>Material Properties</td>
<td>340</td>
</tr>
<tr>
<td>Alpha Blending</td>
<td>340</td>
</tr>
<tr>
<td>Fog</td>
<td>341</td>
</tr>
<tr>
<td>Example</td>
<td>341</td>
</tr>
<tr>
<td>Bézier Curves</td>
<td>342</td>
</tr>
<tr>
<td>One-Dimensional Evaluators</td>
<td>342</td>
</tr>
<tr>
<td>Two-Dimensional Evaluators</td>
<td>344</td>
</tr>
<tr>
<td>Using the mouse</td>
<td>345</td>
</tr>
<tr>
<td>Parameters</td>
<td>347</td>
</tr>
</tbody>
</table>
About JSL 3-D Scenes

JMP’s 3-D scene language is built on top of the OpenGL® API, extending, replacing, and leaving out various parts of the OpenGL API, and as such is not an implementation which is certified or licensed by Silicon Graphics, Inc. under the OpenGL API.

This chapter documents JMP’s JSL commands for creating 3-D scenes but is not a tutorial on OpenGL programming. If you are not familiar with OpenGL programming you may want supplemental material. If you are familiar with OpenGL programming, you still need to read this chapter because some items are non-standard.

JMP ships with sample files in the Scene3D subfolder of Sample Scripts which get you started and give you some ideas. Some of the example scripts are similar to some of the examples in this chapter; some are almost complete applications.

The web site opengl.org is a good jumping off point for information, as is your favorite search engine. JMP's Scene 3-D language does some work for you that the OpenGL API requires you to do for yourself. JMP makes text easy, gives you a built-in arcball controller, and makes sure the matrix operations that belong on the model view stack and projection stack go on their respective stacks. JMP uses its own display list manager so your scenes can be journaled and played back later, and provides a pick mechanism that calls back to your JSL code to tell you what object in your scene is under the mouse, with almost no extra effort on your part. At this time, JMP does not provide access to some features, like texturing.

OpenGL is a trademark of Silicon Graphics, Inc.

JSL 3-D Scene Basics

These commands are necessary to set up and configure a 3-D scene.

Scene Boxes

Like all displays in JMP (detailed in the “Display Trees” chapter), 3-D scenes must be placed in a display box (in this case, a Scene Box). This box is then placed in a window. Therefore, a simple 3-D scene script has the following form.

```jscript
myScene=Scene Box(300, 300); //create a 300 by 300 pixel scene box
...(commands to set up the scene)...
New Window (“3-D Scene”, myScene); //draw the scene in a window
...(commands that manipulate the scene)
```

The scene can be sent messages that construct elements in the scene. Typical messages alter the viewer’s vantage point, construct physical elements in the scene itself, or manipulate lights and textures. These messages are maintained in a display list and are manipulated in one of two ways:

- They are sent as messages to the scene, which immediately adds them to the scene’s internal display list.
They are sent as messages to a display list stored in a global variable, which is called by the scene’s display list later.

```javascript
myscene = SceneBox(300, 300);
dl = SceneDisplayList
```

As an example of commands sent to the scene’s display list directly, consider the following small script. Each of the commands is explained in detail later in the chapter.

```javascript
scene = SceneBox(400, 400); // make a scene box.
NewWindow("Example 1", scene); // put the scene in a window.
scene << Perspective(45, 3, 7); // define the camera
scene << Translate(0.0, 0.0, -4.5); // move to (0,0,-4.5) to draw
scene << color(1,0,0); // set the RGB color of the text
scene << Text(center, baseline, 0.2, "Hello, World."); // add text
scene << Update; // update the scene
```

The first two lines create a scene and place it in a window. The `Perspective` command defines the viewing angle and field depth. By sending it as a message to the scene, it is immediately added to the scene’s display list. Since the “Hello World” text is to be drawn at the origin (0, 0, 0), the `Translate` command is added to the display list to move the camera back a bit so that the origin is in the field of vision. The color is set to red with the `Color` command, the text is drawn, and the `Update` command causes the scene to be rendered (i.e., causes the display list that contains the commands to be drawn.)
Equivalently, the commands to construct the display can be accumulated in a display list stored in a global variable, which is then sent to the scene all at once. To define a global variable as a display list, assign it using the Scene Display List function. For example, to use the global greeting as a display list, issue the command

```
greeting = Scene Display List();
```

Display commands can then be sent as messages to greeting. An equivalent “Hello World” example using a display list follows.

```
// create a display list and send it commands
greeting = Scene Display List();
greeting << color(1,0,0); //set the RGB color of the text
  greeting << Text( center, baseline, 0.2, "Hello, World." ); //add text

  //draw the window and send it the stored display list
scene = Scene Box( 400, 400 ); // make a scene box.
New Window( "Example 1", scene ); // put the scene in a window.
scene << Perspective( 45, 3, 7 ); // define the camera
scene << Translate( 0.0, 0.0, -4.5 ); // move to (0,0,-4.5) to draw
scene << Call List(greeting); //send the display list to the scene
scene << Update; // update the scene
```

Note which commands were separated into the display list, and which were applied to the scene directly. Those that manipulate the camera (Translate and Rotate) are applied to the scene. Those that define the object (Color and Text) were relegated to the display list. This is done so that the display list can be called many times to replicate the object at different positions.
Setting the Viewing Space

3-D scenes can be rendered in two ways. *Orthographic* projections place the elements in a box, where coordinates are not changed to accommodate the perspective of the viewer. *Perspective* projections modify the display to simulate the position of the elements in relation to the position of the viewer. For example, two parallel lines (like railroad tracks) stay parallel in orthographic projections, but seem to connect at a distance in perspective projections.

As another example, imagine looking at a tube edge-on (like a telescope). In an orthographic projection, the tube would appear as a thin circle. In a perspective projection, the circle would have a thickness; the hole at the far end of the tube would appear smaller than the close hole, and the interior of the tube is visible.

Therefore, the viewable space of an orthographic projection is a rectangular shape, while that of a perspective projection is the frustum of a pyramid (that is, a pyramid whose top has been sliced off).
In general, perspective projections give a more realistic view of the world, since it mimics the way an eye or a camera sees. Orthographic projections are important when it is essential to preserve dimensions, such as an architectural CAD program.

**Setting Up a Perspective Scene**

To set up a perspective scene in JSL, send the `Perspective` command to a display list.

\[
\text{Perspective (angle, near, far)}
\]

where `angle` is the viewing angle, `near` is the distance to the near plane, and `far` is the distance to the far plane, as illustrated in the drawing above. A couple of things need to be remembered when defining the viewing space.

- Items outside the viewing space (for example, closer than the `near` plane or farther than the `far` plane) are not drawn. They are clipped off.
- The ratio of `far` to `near` needs to be small so that the rendering engine can effectively determine which items should be drawn “on top of” other items, simulating closeness of items. The `near` parameter must be greater than zero.

The “Hello World” example contains the line

\[
\text{scene \textless\textgreater Perspectiv}e(45, 3, 7); // define the camera}
\]

This defines a 45 degree viewing angle, with a near plane 3 units from the viewer and a far plane 7 units from the viewer.

The viewing angle functions in the same way as a wide angle or telephoto lens on a camera. Small viewing angles zoom into a drawing, while wide angles zoom out. In other words, a small viewing angle maps the screen space onto a small portion of the scene, resulting in apparently larger scene elements. A large viewing angle maps the screen space onto a large portion of the scene, resulting in apparently small screen elements. The size of scene elements can therefore be manipulated using the `angle` parameter of the `Perspective` function. The picture here shows the hello world script with perspective angles of 45 and 90 degrees.

**Hello, World.**

\[
\text{scene \textless\textgreater Perspective(} 45, 3, 7 );
\]

As an alternative to the `Perspective` command, you can define the actual viewing frustum with the `Frustum` command.

**Hello, World.**

\[
\text{scene \textless\textgreater Perspective(} 90, 3, 7 );
\]
Frustum\((left, right, bottom, top, near, far)\);
The frustum’s viewing volume is defined by \((left, bottom, near)\) and \((right, top, near)\) which specify the \((x, y, z)\) coordinates of the lower left and upper right corners of the near clipping plane; \(near\) and \(far\) give the distances from the viewpoint to the near and far clipping planes.

### Setting up an Orthographic Scene

Orthographic scenes are specified in ways similar to perspective scenes. Issue the command

\[
\text{Ortho}(left, right, bottom, top, near, far)
\]

which specifies the four corners of the near plane, the distance to the near plane, and the distance to the far plane.

If you are dealing with a simple 2-D environment, you can set up a two-dimensional orthographic scene with the command

\[
\text{Ortho2D}(left, right, bottom, top)
\]

which specifies the corners of the two-dimensional view.

### Changing the View

One of the advantages of creating an 3-D scene is the ease that they can be viewed from different angles and positions. The \texttt{Translate} and \texttt{Rotate} commands let you set the position from which you view the scene.

In addition, you can use the \texttt{ArcBall} command to allow the user to change the viewing angle interactively.

#### The Translate Command

You have actually seen the \texttt{Translate} command in earlier sample scripts. It sets the position from which the scene is viewed. The arguments give the amount to move from the current position in the \(x\), \(y\), and \(z\) direction.

\[
\text{Translate}(x, y, z)
\]

For example,

\[
\text{Translate}( 0.0, 0.0, -2 );
\]

moves the origin two units in the negative \(z\) direction.

Initially, the origin and camera were at the same place. Now, the camera can see the origin because the camera faces down the negative \(z\)-axis.

#### The Rotate Command

The \texttt{Rotate} command is used to modify the viewing angle of a scene. It has the following format.
Rotate \((\text{degrees}, \, x\text{Axis}, \, y\text{Axis}, \, z\text{Axis})\)

This rotates by degrees around the axis described by the vector \((x\text{Axis}, \, y\text{Axis}, \, z\text{Axis})\). For example, to rotate a model 90 degrees about the \(x\)-axis, use \texttt{Rotate( 90, 1, 0, 0 )}.

You can also specify the three axis values in a matrix, e.g. \texttt{Rotate( 90, [1, 0, 0] )}.

**Note:** The \texttt{Rotate} command uses degrees, in contrast to JMP’s trigonometric functions, which use radians.

\texttt{Translate} and \texttt{Rotate} are also used to position objects with respect to each other. The first \texttt{Translate} or \texttt{Rotate} can be thought of as positioning everything that follows with respect to the camera. Subsequent \texttt{Translate} and \texttt{Rotate} commands are used to position objects, such as spheres, cylinders, disks, and display lists in \texttt{Call List} and \texttt{ArcBall} commands. For example, suppose you have a display list named \texttt{table} and another named \texttt{chair}. Your scene might look like this:

```plaintext
// make a scene box...holds an OpenGL scene.
scene = SceneBox( 600, 600 );

// put the scene in a window.
NewWindow( "Example 1", scene );

for (i=1, i<360, i++,
scene << clear;

// the lens is 45 degrees, near is 1 units from the camera, far is 10.
scene << Perspective( 45, 1, 10 );
scene << Translate( 0.0, 0.0, -2 );

scene << Rotate(i,1,0,0);
scene << Rotate(i*3, 0, 1, 0);
scene << Rotate (i*3/2, 0, 0, 1);
scene << Color(0, 1, 0); //green for cylinder
scene << Cylinder(0.5, 0.5, 0.5, 40,10);
scene << Color(0, 0, 0); //black for sphere
```

The following example uses the \texttt{Rotate} command inside a \texttt{for} loop to continuously change the viewing angle of a scene. It draws a cylinder that swings around a central point. This central point is shown by a small sphere.

```plaintext
// make a scene box...holds an OpenGL scene.
scene = SceneBox( 600, 600 );

// put the scene in a window.
NewWindow( "Example 1", scene );

for (i=1, i<360, i++,
scene << clear;

// the lens is 45 degrees, near is 1 units from the camera, far is 10.
scene << Perspective( 45, 1, 10 );
scene << Translate( 0.0, 0.0, -2 );

scene << Rotate(i,1,0,0);
scene << Rotate(i*3, 0, 1, 0);
scene << Rotate (i*3/2, 0, 0, 1);
scene << Color(0, 1, 0); //green for cylinder
scene << Cylinder(0.5, 0.5, 0.5, 40,10);
scene << Color(0, 0, 0); //black for sphere
```
Note the use of the `Update` command at the end of the scene messages. This command tells JMP to make the displayed screen agree with the current state of the display list. It is important to clear the list at the beginning (so the list doesn’t contain the old angles as well as the current) and update the scene after each change.

### The Look At Command

The `Look At` command is an alternative way to set the camera view.

```plaintext
Look At( eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ )
```

The `Look At` command puts the camera at the `eye` coordinates and points it toward the `center` coordinates. The `up` vector describes how the camera is rotated on its line of sight. Because the model is typically constructed at the origin, a JMP scene should have either a `Look At` or a `Translate` command near its beginning to move the camera away from the origin.

First, clear the scene box of any commands from the previous frame.

```plaintext
scene<<clear;
```
Then use one of these projections:

```lang-plaintext
scene <<perspective(45,2,10);
scene <<frustum(-.5,.5,-.5,.5,1,10);
scene <<ortho2d(-2,2,-2,2);
```

**Note:** If you use the `ortho2d` projection, you should not also set the camera position using either `Translate` or `Look At`.

Finally, use either `Translate` or `Look At` to set the camera position:

```lang-plaintext
scene <<Translate(0.0, 0.0, -4.5);
/* the camera faces down the negative Z axis.
   move it back so 0,0,0 is in view. */
scene <<Look At( /*eye*/ 3,3,3, /*center*/ 0,0,0, /*up*/ 1,0,0 );
/*this is much easier. */
```

Once the scene and camera position are set, add your model.

### The ArcBall

Sometimes you want a scene to rotate based on the movements of the mouse. The Surface Plot platform in JMP is an example of a 3-D scene that rotates based on mouse movements.

An ArcBall creates a sphere around the 3-D scene and allows the user to click on the sphere’s surface and drag it around, thus causing the scene to rotate.

Use an ArcBall instead of a `CallList` command to place the scene in an ArcBall. Scenes that are attached to an ArcBall automatically respond to clicks and drags of the mouse. Custom programming is not needed. However, rotations made in the arcball are not saved. (Technically, the ArcBall is surrounded by an implicit Push Matrix and Pop Matrix block, so the movements are gone after it returns. See “Using the Matrix Stack,” p. 334 for details of pushing and popping.)

As an example, examine the script from “Primitives Example,” p. 324. Change the single line

```lang-plaintext
scene << CallList(shape); //send the display list to the scene
```

so that it reads

```lang-plaintext
scene << ArcBall(shape,2); //send the display list to an arcball
```

This displays the script with an associated arcball with diameter 2. When you run the script and the window appears, Right-click (Control-click on the Macintosh) and select **Show ArcBall > Always** from the menu that appears.

**Note:** ArcBall comes from an article by Shoemake (1994) found in *Graphics Gems IV*, published by Academic Press.
This sets the display so that the ArcBall is always showing. Click and drag on the ArcBall to rotate the scene. The popup menu with Background Color, Use Hardware Acceleration, and Show ArcBall is always available, whether the scene is displayed through a platform, in a journal, or through JSL.

**Note:** The ArcBall does not have to be showing to react to mouse commands. It is shown here for display purposes only.

You also can set the display state of the ArcBall in JSL using the `Show ArcBall` command.

```jscript
scene << Show Arcball (state)
```

where `state` is During Drag, Always, or Never.
Graphics Primitives

All scenes in JSL are built with a small number of graphics primitives. These fundamental elements function as the building blocks for complicated scenes.

Every graphics primitive involves specifying vertices. In some cases, the vertices are simply drawn as points. In others, the vertices are connected to form polygons. To draw a primitive, you must specify the type of primitive and the coordinates and properties of the vertices involved. In JSL, this specification is accomplished through the Begin and End statements.

```jscript
scene<<Begin(primitive type);
...(commands specifying vertices and their properties)...
scene<<End();
```

To specify the coordinates of the vertices, use the vertex command.

```jscript
scene<<Begin(primitive type);
scene<<Vertex(x, y, z);
...
scene<<End();
```

The options for primitive type are the following. In these examples, assume that $v_0$, $v_1$, etc. have been specified between a Begin/End pair, similar to the following.

```jscript
scene<<Begin(primitive type);
scene<<Vertex(x0, y0, z0)//specify vertex v0
scene<<Vertex(x1, y1, z1)//specify vertex v1
...
scene<<Vertex(xn, yn, zn)//specify vertex vn
```
primitive type=POINTS
Draws a point at each of the vertices.

primitive type=LINES
Draws a series of (unconnected) line segments. Segments are drawn between v0 and v1, between v2 and v3, and so on. If \( n \) is odd, the last vertex is ignored.

primitive type=POLYGON
Draws a polygon using the points v0, ..., vn as vertices. Three vertices must exist, or nothing is drawn. In addition, the polygon specified must not intersect itself and must be convex. If the vertices don’t satisfy these conditions, the results are unpredictable.

primitive type=TRIANGLES
Draws a series of (disconnected) triangles using vertices v0, v1, v2, then v3, v4, v5, and so on. If the number of vertices isn’t an exact multiple of 3, the final one or two vertices are ignored.

primitive type=LINE_STRIP
Draws a line segment from v0 to v1, then from v1 to v2, and so on. Therefore, \( n \) vertices specify \( n-1 \) line segments. Nothing is drawn unless there is more than one vertex. There are no restrictions on the vertices describing a line strip; the lines can intersect arbitrarily.
9 Scripting Three Dimensional Scenes

Chapter 9

Three-Dimensional Scenes

Graphics Primitives

323

primitive type=LINE_LOOP

Same as LINE_STRIP, except that a final line segment is drawn from the last vertex to the first, completing a loop.

primitive type=QUADS

Draws a series of quadrilaterals (four-sided polygons) using vertices \( v_0, v_1, v_2, v_3 \), then \( v_4, v_5, v_6, v_7 \), and so on. If the number of vertices isn’t a multiple of 4, the final one, two, or three vertices are ignored.

primitive type=QUAD_STRIP

Draws a series of quadrilaterals (four-sided polygons) beginning with \( v_0, v_1, v_3, v_2 \), then \( v_2, v_3, v_5, v_4 \), then \( v_4, v_5, v_7, v_6 \), and so on. The number of vertices must be at least 4 before anything is drawn, and if odd, the final vertex is ignored.

primitive type=TRIANGLE_STRIP

Draws a series of triangles (three-sided polygons) using vertices \( v_0, v_1, v_2 \), then \( v_2, v_1, v_3 \) (note the order), then \( v_2, v_3, v_4 \), and so on. The ordering is to ensure that the triangles are all drawn with the same orientation so that the strip can correctly form part of a surface. There must be at least three vertices for anything to be drawn.
primitive type = TRIANGLE_FAN

Same as TRIANGLE_STRIP, except that the vertices are $v_0$, $v_1$, $v_2$, then $v_0$, $v_2$, $v_3$, then $v_0$, $v_3$, $v_4$, and so on.

Primitives Example

The following short example illustrates the use of a graphics primitive.

```cpp
// create a display list and send it commands
shape = Scene Display List();
shape << Color(1,0,0); //set the RGB color of the text
shape << Begin(POLYGON);
shape << Vertex(0, 0, 0);
shape << Vertex(0, 3, 0);
shape << Vertex(3, 3, 0);
shape << Vertex(5, 2, 0);
shape << Vertex(4, 0, 0);
shape << Vertex(2, -1, 0);
shape << End();

//draw the window and send it the stored display list
scene = Scene Box( 400, 400 ); // make a scene box.
New Window( "Primitive", scene ); // put the scene in a window.
scene << Perspective( 90, 3, 7 ); // define the camera
scene << Translate( 0.0, 0.0, -5 ); // move to (0,0,-5) to draw
scene << Call List(shape); //send the display list to the scene
scene << Update; // update the scene
```

The first section of the script creates a display list named `shape`. Inside this display list, a polygon is defined using six vertices.

The second section of the script creates a scene box and a new window. It then uses the `Call List` function to put the list in the display.
Note that all the $z$-coordinates are zero, which makes sure the polygon lies in a plane. Polygons that don't lie in a plane can cause unpredictable results.

Experiment with the line

```plaintext
shape <<Begin(POLYGON);
```

by changing it to some of the other primitive types. For example, changing it to

```plaintext
shape <<Begin(TRIANGLES);
```

results in a different picture.
Controlling the Appearance of Primitives

JSL has several commands that let you tailor-make the appearance of primitive drawing objects. You can also specify the widths of lines and their stippling pattern (that is, whether they are dashed, dotted, and so on.)

Size and Width

To set the point size of rendered objects, use the Point Size command.

```
Point Size (n)
```

where \( n \) is the number of pixels. Note that this may not be the actual number of pixels rendered, depending on other settings such as anti-aliasing and your hardware configuration.

Set the line width using the Line Width command

```
Line Width(n)
```

where \( n \) is the number of pixels. The argument \( n \) must be larger than zero and is, by default, one.

Stippling Pattern

To make stippled lines, use the Line Stipple command.

```
Line Stipple(factor, pattern)
```

Factor is a stretching factor. Pattern is a 16-bit integer that turns pixels on or off. Use Enable(LINE_STIPPLE) to turn the effect on.

To construct a line stippling pattern, write a 16-bit binary number that represents the stippling pattern that you desire. Note that the pattern should read from right to left, so your representation may seem backward to the way it is rendered. Convert the binary number to an integer and use this as the pattern parameter.

For example, imagine you want the dotted line pattern 0000000011111111. This is equal to 255 in decimal notation, so use the command Line Stipple(1, 255).

The factor parameter expands each binary digit to two digits. In the example above, Line Stipple(2, 255) would result in 00000000000000000000111111111111111.

As an example, the following script draws three lines, each of different widths (the Line Width commands) and stippling patterns.
// make a scene box...holds an OpenGL scene.
scene = SceneBox(200, 200);

// put the scene in a window.
NewWindow("Stipples", scene);

scene << Ortho(-2,2,-2,2,-1,1);

scene << color(0,0,0);
    //set the RGB color of the text
scene << Enable(LINE_STIPPLE);

scene << Line Width(2);
scene << Line Stipple(1, 255);

scene << Begin(LINES);
scene << Vertex(-2, -1, 0);
scene << Vertex(2, -1, 0);
scene << End();

scene << Line Width(4);
scene << Line Stipple(1, 32767);

scene << Begin(LINES);
scene << Vertex(-2, 0, 0);
scene << Vertex(2, 0, 0);
scene << End();

scene << Line Width(6);
scene << Line Stipple(3, 51);

scene << Begin(LINES);
scene << Vertex(-2, 1, 0);
scene << Vertex(2, 1, 0);
scene << End();
scene << Update;
Note: Stipple patterns “crawl” on rotating models because they are in screen pixels, not model units, and lines in the model change length on the screen even though nothing changes in model units. To see an example of this effect, replace the scene<<Call List(d1) line in the script above with scene<<ArcBall(d1,2) and spin the resulting display by Shift-dragging and releasing the mouse.

Fill Pattern

Polygons are rendered with both a front and a back, and the drawing mode of each side is customizable. This allows the user to see the difference between the back and front of the polygon.

To set the drawing mode of a polygon, use the Polygon Mode command.

Polygon Mode (face, mode)

where face can be FRONT, BACK, or FRONT_AND_BACK, and mode can be POINT, LINE, or FILL.

As an example, the following script creates a display list that defines a triangle. This display list is used three times in conjunction with Translate, Rotate, and Color commands to draw triangles in three positions. In addition, the Polygon Mode command changes the drawing mode of each triangle. Note there is no explicit call to the FILL mode, since it is the default.

The following table dissects the script, showing how the Translate and Rotate commands accumulate to manipulate a single display list.

Table 9.1 Translate and Rotate Commands

<table>
<thead>
<tr>
<th>Code from above script</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>shape = Scene Display List();</td>
<td>Creates a display list</td>
</tr>
<tr>
<td>shape &lt;&lt; Begin(TRIANGLES);</td>
<td>Creates a display list named shape that holds vertices for the triangles. All the z vertices are zero since this is a two dimensional scene</td>
</tr>
<tr>
<td>shape &lt;&lt; Vertex(0, 0, 0);</td>
<td></td>
</tr>
<tr>
<td>shape &lt;&lt; Vertex(-1, 2, 0);</td>
<td></td>
</tr>
<tr>
<td>shape &lt;&lt; Vertex(1, 2, 0);</td>
<td></td>
</tr>
<tr>
<td>shape &lt;&lt; End();</td>
<td></td>
</tr>
<tr>
<td>scene = Scene Box(200, 200);</td>
<td></td>
</tr>
<tr>
<td>New Window(&quot;Fill Modes&quot;, scene);</td>
<td>Put the scene in a display box, and create a new window.</td>
</tr>
<tr>
<td>scene &lt;&lt; Ortho2d(-2,2,-2,2);</td>
<td></td>
</tr>
</tbody>
</table>
Table 9.1  Translate and Rotate Commands

Code from above script

```cpp
scene << Color(1, 0, 0);
scene << Call List(shape);
scene << Update;

// update the scene to see the triangle
```

```
scene << Rotate (90, 0, 0, 1);
scene << Translate (-0.5, 0, 0);
scene << Color(0, 0.5, 0.5);
scene << Polygon
    Mode(FRONT_AND_BACK, LINE);
scene << Call List(shape);

// update the scene to see the triangle
```

```
scene << Rotate (90, 0, 0, 1);
scene << Translate (-0.5, 0, 0);
scene << Color(0, 0.5, 0.5);
scene << Polygon
    Mode(FRONT_AND_BACK, LINE);
scene << Call List(shape);

// update the scene to see the triangle
```

**comments**

Draw the first triangle in red.

Draw the second triangle in teal. Note that we first rotate the triangle...

...then translate it.
Some developers use the fill mode in concert with the line mode to draw a filled polygon with a differently-colored border. However, due to the way the figures are rendered, they sometimes do not line up correctly. The Polygon Offset command is used to correct for this so-called “stitching” problem.

Polyon Offset \((\text{factor}, \text{units})\)

To enable offsetting, use `Enable(POLYGON_OFFSET_FILL)`, `Enable(POLYGON_OFFSET_LINE)`, or `Enable(POLYGON_OFFSET_POINT)`, depending on the desired mode. The actual offset values are calculated as \(m*\text{(factor)}+r*\text{(units)}\), where \(m\) is the maximum depth slope of the polygon and \(r\) is the smallest value guaranteed to produce a resolvable difference in window coordinate depth values. Start with Polygon Offset\((1,1)\) if you need this.

An example of Polygon Offset is in the Surface Plot platform, when a surface and a mesh are displayed on top of each other, or a surface and contours displayed on top of each other. In either case, the surface would interfere with the lines if the lines were not moved closer or the surface moved farther from the viewer.

### Table 9.1 Translate and Rotate Commands

**Code from above script**

```plaintext
scene << Rotate (90, 0, 0, 1);
scene << Translate (-0.5, -1, 0);
scene << Color(0, 0, 0);
scene << Point Size(5); //large points so they are visible
scene << Polygon
    Mode(FRONT_AND_BACK, POINT);
scene << Call List(shape);

// update the scene to see the triangle
```

**Comments**

Draw the third triangle as black points. First rotate...

...then translate to get the final picture.

```plaintext
...then translate to get the final picture.
```
Other uses of Begin and End

Although vertices are typically specified between begin and end statements, there are other commands that are valid. These commands are discussed in other sections of this chapter.

- **Vertex**: adds a vertex to the list
- **Color**: changes the current color
- **Normal**: sets the normal vector coordinates
- **Edge Flag**: controls drawing of edges
- **Material**: sets material properties
- **Eval Coord** and **Eval Point**: generate coordinates
- **Call List**: executes a display list.

Drawing Spheres, Cylinders, and Disks

There are several pre-defined commands that allow for quick rendering of spheres, cylinders, and disks. The advantage of these commands is not only their ease-of-use, but that they have special lighting properties (their “normals”) built in.

Construction

The following commands are used to construct cylinders, disks, partial disks, and spheres.

**Cylinders**

`Cylinder( baseRadius, topRadius, height, slices, stacks )`

- **baseRadius**: the radius of the cylinder’s base. Similarly, **topRadius** is the radius of the top. **height** is the height of the cylinder.
- **Slices**: can be 10 for a reasonably-accurate cylindrical shape. Using `QuadricNormals(Smooth)` helps the appearance.
- **Stacks**: sets the number of vertices available for lighting reflections. Use a larger value for **Stacks** for accurate “hot-spots”.

**Disks**

The following command draws a paper-thin disk with an **innerRadius** hole in the middle.

`Disk( innerRadius, outerRadius, slices, loops )`

Like `Cylinder`, **slices** controls the accuracy of the curve and **loops** will make more vertices (for lighting accuracy).

`Partial Disk( innerRadius, outerRadius, slices, loops, startAngle, sweepAngle )`
The Partial Disk command works like Disk, but with a slice of the disk removed. Specify the part of the disk that is showing using \textit{startAngle} and \textit{sweepAngle}.

\textbf{Spheres}

The following command draws a sphere with the specified \textit{radius}.

\begin{verbatim}
Sphere( radius, slices, stacks )
\end{verbatim}

The \textit{slices} can be thought of as longitudes and \textit{stacks} as latitudes. About 10 of each make a nicely-drawn sphere.

\textbf{Lighting}

It is not necessary to make specific calculations of normal vectors (as is the case for customized surfaces) for spheres, disks, and cylinders. However, you can use the following commands to tailor the automatic lighting.

\textbf{Quadric Normals} tells what kind of normal should be automatically generated. The parameter \textit{mode} can be \texttt{None}, \texttt{Flat}, or \texttt{Smooth}. \texttt{Flat} makes faceted surfaces. \texttt{Smooth} makes the normals at each vertex be the average of the adjacent polygons.

\textbf{Quadric Orientation} determines which way the normals point. The parameter \textit{mode} can be \texttt{Inside} or \texttt{Outside}.

\textbf{Quadric Draw Style} specifies the drawing mode. The parameter \textit{mode} can be \texttt{Fill}, \texttt{Line}, \texttt{Silhouette} or \texttt{Point}.

JMP uses the values you set for \textbf{Quadric Normals}, \textbf{Quadric Orientation}, and \textbf{Quadric Draw Style} for subsequently generated cylinders, disks, and spheres.

\textbf{Note}: Other OpenGL documentation refers to quadric objects. JMP only has one, and always uses it.

\textbf{Drawing Text}

As shown in the “Hello World” example above, text is added to a scene using the \texttt{Text} command.

\begin{verbatim}
Text( horz, vert, size, string, <billboard>)
\end{verbatim}

- \textit{horz} can be \texttt{Left}, \texttt{Center}, or \texttt{Right} justification.
- \textit{vert} can be \texttt{Top}, \texttt{Middle}, \texttt{Baseline}, or \texttt{Bottom} justification.
- \textit{size} represents the height of a capital letter M in model coordinates.
• `string` is the text to draw.
• `billboard` is an optional argument that causes the text to rotate with the model. Text with this option always faces the viewer.

The font will always be the JMP Text font. You can change the text font from the preferences menu, but because of the way JMP caches fonts for scenes, changes may not take effect until JMP is restarted.

**Note:** Text is not part of the standard OpenGL definition.

### Using Text with Rotate and Translate

The following example uses the text command in conjunction with the `Translate` and `Rotate` commands.

```cpp
/* make a scene box...holds an OpenGL scene */
scene = SceneBox( 600, 600 );

/* put the scene in a window */
NewWindow( "Example 2", scene );

scene <<= Perspective( 45, 3, 7 );
    /* the "lens" is 45 degrees, near is 3 units from the camera, far is 7 */
scene <<= Translate( 0.0, 0.0, -4.5 );
    /* move the world so 0,0,0 is visible in the camera */
scene <<= Rotate( 30, 0, 1, 0 );
    /* rotate the first text about the Y (vertical on screen) axis */
scene <<= Color( 1, 0, 0 );
    /* pure red */
scene <<= Text( center, baseline, .2, "First Red String" );
scene <<= Translate( 0.0, 0.0, -2.0 );
    /* the next string will be even further away from the camera */
scene <<= Rotate( 30, 0, 1, 0 );
    /* rotate the second text about the Y (vertical on screen) axis */
scene <<= Color( 0, 1, 0 );
    /* pure green */
scene <<= Text( center, baseline, .2, "Second Green very long string" );
scene <<= Update;
    /* update the displaybox in the window using the current display list */
```

Note the green string is extending backwards beyond the far clipping plane. Change the 7 to 10 in the `Perspective` command to see the complete string.
Using the Matrix Stack

JMP 3-D scenes use a matrix stack to keep track of the current transform. The stack is initialized to the identity matrix, and each time a translate, rotate, or scale command is given, the top matrix on the stack is changed.

Confusion Alert: Unlike many OpenGL implementations, JMP does not use a transposed matrix.

The JSL example below uses Push Matrix and Pop Matrix to position pieces of the toy top and then return to the origin. This is faster than using the Translate command a second time in reverse.

```
toyTop = SceneDisplayList();
toyTop<<PushMatrix;
toyTop<<Translate(0,0,.1);
toyTop<<Color(1,0,0); // red
    toyTop<<Cylinder(1,.2,.2,25,5);
        /* baseRadius, topRadius, height, slices, stacks */
toyTop<<PopMatrix;
toyTop<<PushMatrix;
toyTop<<Translate(0,0,-.1);
toyTop<<Rotate(180,1,0,0);
toyTop<<Color(0,1,0); // green
    toyTop<<Cylinder(1,.2,.2,25,5);
toyTop<<PopMatrix;
toyTop<<Color(0,0,1); // blue
    toyTop<<Sphere(.5,30,30);
        /* radius, slices, stacks */
toyTop<<Color(1,1,1); // yellow
    toyTop<<PartialDisk(1,1.2,25,2,0,270);
        /* innerRadius, outerRadius, slices, rings, startAngle, sweepAngle */
toyTop<<PushMatrix;
toyTop<<Translate(0,0,-.1);
toyTop<<Color(1,0,1); // magenta
```
toyTop<<Cylinder(1,1,.2,25,3);
/* baseRadius, topRadius, height, slices, stacks */
toyTop<<PopMatrix;

/* make a scene box...holds an OpenGL scene */
scene = SceneBox( 600, 600 );

/* put the scene in a window */
NewWindow( "Example 3", scene );
scene << Perspective( 45, 3, 7 );
scene << Translate( 0.0, 0.0, -4.5 );
scene << Rotate( -85, 1, 0, 0 );
scene << Rotate( 65, 0, 0, 1 );
scene << CallList( toyTop );

/* update the displaybox */
scene << Update;

There are some cases where you want to replace the current matrix on the stack. For these cases, use the Load Matrix command.

Load Matrix(m)

where m is a 4x4 JMP matrix that is loaded onto the current matrix stack.

Similar is the Mult Matrix command

Mult Matrix(m)

When the Mult Matrix command is issued, the matrix on the top of the current matrix stack is multiplied by m.

The following matrices perform some simple commands.

Translation:

\[
\begin{bmatrix}
1 & 0 & 0 & x \\
0 & 1 & 0 & y \\
0 & 0 & 1 & z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

In the following rotation matrices, c = cos(angle) and s = sin(angle).

Rotation about x-axis:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & c & -s & 0 \\
0 & s & c & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Rotation about y-axis:
Using the Matrix Stack

Rotation about z-axis:

\[
\begin{bmatrix}
c & -s & 0 & 0 \\
s & c & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

As an example, here are two equivalent (except for the translation being opposite) ways to translate and rotate a display list.

// first way uses matrix

```cpp
gl<<Push Matrix;
xt = identity(4); // translate this one left by .75
xt[1,4]=-0.75;
xr = Identity(4); // rotate this one, cos needs radians, not degrees
xr[2,2]=cos(3.14159*a/180);
xr[2,3]=-sin(3.14159*a/180);
xr[3,2]=sin(3.14159*a/180);
xr[3,3]=cos(3.14159*a/180);
yr = Identity(4);
yr[1,1]=cos(3.14159*a/180);
yr[1,3]=sin(3.14159*a/180);
yr[3,1]=-sin(3.14159*a/180);
yr[3,3]=cos(3.14159*a/180);
zr = identity(4);
zr[1,1]=cos(3.14159*a/180);
zr[1,2]=-sin(3.14159*a/180);
zr[2,1]=sin(3.14159*a/180);
zr[2,2]=cos(3.14159*a/180);
gl<<Mult Matrix(xt*xr*yr*zr); // order of multiplication matters with matrices
```

// second way uses functions

```cpp
// Push Matrix;
gl<<arcball(d1,1);
gl<<Pop Matrix;
```

It is not possible to read back the current transform matrix, because the matrix only exists while the display list is drawing, not while your JSL script is creating it. If you must know its content, create it in JSL and use Load Matrix to put it on the stack.
Lighting and Normals

The following methods allow you to add lighting, materials, and normal vectors to your shapes. Using these methods, models can appear shiny or light-absorbing.

Creating Light Sources

Light sources are specifications of a color, position, and direction. JSL allows for up to eight lights (numbered 0 to 7) defined by the Light command, where \( n \) is the number of the light.

\[
\text{Light}( n, \text{parameter}, \text{value}, \ldots \text{value} )
\]

**Note:** To turn each light on, issue an Enable (Lighting) and an Enable (light \( n \)) command, where \( n \) is the light number. Then, move the light to a position in the scene with a Light(\( n\), POSITION, \( x \), \( y \), \( z \)) command.

The value of parameter can be any one of those shown in Table 9.2. The table shows default values for each parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMBIENT</td>
<td>(0, 0, 0, 1)</td>
<td>Ambient RGBA intensity</td>
</tr>
<tr>
<td>DIFFUSE</td>
<td>(1, 1, 1, 1)</td>
<td>diffuse RGBA intensity</td>
</tr>
<tr>
<td>SPECULAR</td>
<td>(1, 1, 1, 1)</td>
<td>specular RGBA intensity</td>
</tr>
<tr>
<td>POSITION</td>
<td>(0, 0, 1, 0)</td>
<td>((x, y, z, w)) position</td>
</tr>
<tr>
<td>SPOT_DIRECTION</td>
<td>(0, 0, -1)</td>
<td>((x, y, z)) direction of spotlight</td>
</tr>
<tr>
<td>SPOT_EXPONENT</td>
<td>0</td>
<td>spotlight exponent</td>
</tr>
<tr>
<td>SPOT_CUTOFF</td>
<td>180</td>
<td>spotlight cutoff angle</td>
</tr>
<tr>
<td>CONSTANT_ATTENUATION</td>
<td>1</td>
<td>constant attenuation factor</td>
</tr>
<tr>
<td>LINEAR_ATTENUATION</td>
<td>0</td>
<td>linear attenuation factor</td>
</tr>
<tr>
<td>QUADRATIC_ATTENUATION</td>
<td>0</td>
<td>quadratic attenuation factor</td>
</tr>
</tbody>
</table>

**Note:** The default values for DIFFUSE and SPECULAR in this table only apply to Light 0. For other lights, the default value is (0, 0, 0, 1) for both parameters.

The first three parameters (AMBIENT, DIFFUSE, and SPECULAR) are used to color the light. DIFFUSE is the parameter that is most closely associated with the physical color of the light. AMBIENT refers to the property of the light when it functions as a background light. SPECULAR alters the way a light is reflected off a surface.

Specify the position of the light using the POSITION parameter. Non-zero values of the fourth \((w)\) coordinate position the light in homogenous object coordinates.

Light in the real-world decreases in intensity as distance from the light increases. Since a directional light is infinitely far away, it doesn't make sense to attenuate its intensity as a function of distance. However, JSL attenuates a light source by multiplying the contribution of the source by an attenuation factor.
where $c = \text{CONSTANT\_ATTENUATION}$, $l = \text{LINEAR\_ATTENUATION}$, and $q = \text{QUADRATIC\_ATTENUATION}$.

To create a spotlight, limit the shape of the light to a cone. Use the \text{SPOT\_CUTOFF} parameter to define the side of the cone, as shown in the following illustration.

In addition to the cutoff angle, you can control the intensity and direction of the light distribution in the cone. \text{SPOT\_DIRECTION} specifies the direction for the spotlight to point; \text{SPOT\_EXponent} influences how concentrated the light is.

**Lighting Models**

Lighting models are specified with the \text{Light Model} command.

\begin{verbatim}
Light Model( parameter, value,...,value )
\end{verbatim}

Light models specify three attributes of lights.

- The global ambient light intensity
- Whether the viewpoint is local or is an infinite distance away
- Whether lighting calculations should be performed differently for the front and back faces of objects.

Table 9.2 “Light Parameters and Default Values,” p. 337 shows the three valid parameters for the \text{Light Model} command.

Table 9.3 Light Model Parameters and Default Values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{LIGHT_MODEL_AMBIENT}</td>
<td>(0.2, 0.2, 0.2, 1)</td>
<td>Ambient RGBA intensity of the entire scene</td>
</tr>
<tr>
<td>\text{LIGHT_MODEL_LOCAL_VIEWER}</td>
<td>0 (false)</td>
<td>how specular reflection angles are computed</td>
</tr>
<tr>
<td>\text{LIGHT_MODEL_TWO_SIDE}</td>
<td>0 (false)</td>
<td>non-zero values imply two-sided lighting</td>
</tr>
</tbody>
</table>
Normal Vectors

Normal vectors point in a direction perpendicular to a surface. For a plane, all normals are the same. For a more complicated surface, normals are more complicated. JSL allows you to specify the normal vector for each vertex. These normals specify the orientation of the surface in space, necessary for lighting calculations. Accurate normals assure accurate lighting.

The normal vector is of length 1 and is perpendicular to the vertex. Typically, a vertex is shared between several polygons and a smooth shaded effect is desired, so the perpendicular at the vertex is calculated as a (possibly weighted) average of the polygon's normals. It is important to calculate the "outward" normal for polygons unless two-sided shading is enabled because only the outer face of the polygon is illuminated. With a scaled polygon, the normal's length will not be 1 after scaling and the lighting will be wrong.

Normal vectors are set at the same time the surface is constructed, and are specified with the `Normal` command. Use the `Enable(NORMALIZE)` command to have the normals re-normalized to 1 each time the scene is drawn.

Shading Model

The shading model of a polygon is set using the `Shade Model` command.

\[
\text{Shade Model (mode)}
\]

where mode can be `SMOOTH` (the default) or `FLAT`. `SMOOTH` shading interpolates the colors of the primitive from one vertex to the next. `FLAT` mode duplicates the color of one vertex across the entire primitive.

The following script changes the color at each of a triangle's vertices. The `FILL` shade model interpolates the color of the interior automatically.

```plaintext
// make a scene box...holds an OpenGL scene.
scene = SceneBox( 200, 200 );

// put the scene in a window.
NewWindow( "Shade Model", scene );
scene << clear;
scene << Ortho2D (-1,1,-1,1);

scene << Shade Model(SMOOTH);
scene << Polygon Mode (FRONT_AND_BACK, FILL);
scene << Begin(TRIANGLES);
scene << color(0, 0, 0);  //black
scene << Vertex(-1, -1, 0);
scene << Color(0, 1, 1);//cyan
scene << Vertex(0, 1, 0);
scene << Color (1, 1, 1);  //white
scene << Vertex(1, -1, 0);
scene << End();

scene << Update;
```
Material Properties

To set the material properties of a surface, use the Material command.

\[
\text{Material( face, parameter, value, ..., value )}
\]

`face` can be Front, Back, or Front_and_back. (Note that the material properties can be set separately for the front and back faces of a polygon.)

Table 9.4 shows the parameters and default values for Material parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMBIENT</td>
<td>(0.2, 0.2, 0.2, 1.0)</td>
<td>Ambient color of material</td>
</tr>
<tr>
<td>DIFFUSE</td>
<td>(0.8, 0.8, 0.8, 1.0)</td>
<td>Diffuse color of material</td>
</tr>
<tr>
<td>AMBIENT_AND_DIFFUSE</td>
<td>Both AMBIENT and DIFFUSE</td>
<td>Both AMBIENT and DIFFUSE</td>
</tr>
<tr>
<td>SPECULAR</td>
<td>(0.0, 0.0, 0.0, 1.0)</td>
<td>Specular color of material</td>
</tr>
<tr>
<td>SHININESS</td>
<td>0</td>
<td>Specular exponent that can range from 0 to 128.</td>
</tr>
<tr>
<td>EMISSION</td>
<td>(0, 0, 0, 1)</td>
<td>Emissive color of material</td>
</tr>
</tbody>
</table>

Alpha Blending

The BlendFunc command allows for alpha blending. To use it, send a BlendFunc message to a scene, e.g.,

```plaintext
scene << BlendFunc(SRC_ALPHA, ONE_MINUS_SRC_ALPHA)
```

SRC_ALPHA and ONE_MINUS_SRC_ALPHA are OpenGL constants that tell BlendFunc to use alpha to blend against the existing display buffer. Disabling z-buffer testing or rendering primitives from back to front may be needed for some applications. By default, the z-buffer tests prevent anything from drawing behind a transparent polygon after it is drawn.

Complete details of all the constants available to BlendFunc—many of which are not useful to the JSL programmer—are available in the OpenGL documentation at opengl.org.
Fog

Fog enables figures to fade into the distance, making for more realistic models. All types of geometric figures can be fogged. To turn fog on, enable the FOG parameter.

Example

The following example uses several of the concepts presented in this section, including lighting, fog, and normalization. It draws a spinning cylinder that is affected by two lights.

```plaintext
scene = SceneBox( 300, 300 ); // make a scene box
New Window( "Cylinder", scene ); // put the scene in a window.
for (i=1, i<360, i++)
    scene << Clear;
    // the lens is 45 degrees, near is 3 units from the camera, far is 7.
    scene << Perspective( 50, 1, 10 );
    // move the world so 0,0,0 is visible in the camera
    scene << Translate( 0.0, 0.0, -2 );
    scene<<Enable(Lighting);
    scene<<Enable(Light0);
    scene<<Enable(Light1);
    scene<<Light(Light0,POSITION,1,1,1,1); //near viewer
    scene<<Light(Light0,DIFFUSE,1,0,0,1); //red light
    scene<<Light(Light1,POSITION,-1,-1,-1,1);//behind object
    scene<<Light(Light1,DIFFUSE,.5,.5,1,1); //blue-gray light
    scene<<Enable(Fog);
    scene<<Enable(NORMALIZE);
    scene << Rotate(i,1,0,0);
    scene << Rotate(i*3, 0, 1, 0);
    scene << Rotate (i*3/2, 0, 0, 1);
    scene << Cylinder(0.5, 0.5, 0.5, 40,10);
    scene << Update;
)```
Bézier Curves

A complete discussion of Bézier curves is beyond the scope of this book. JSL has several commands for defining and drawing curves and their associated meshes.

One-Dimensional Evaluators

To define a one-dimensional map, use the Map1 command.

Map1(target, u1, u2, stride, order, matrix)

The target parameter defines what the control points represent. Values of the target parameter are shown in Table 9.5. Note that you must use the Enable command to enable the parameter.
Table 9.5 Map1 Target Parameters and Default Values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAP1_VERTEX_3</td>
<td>(x, y, z) vertex coordinates</td>
</tr>
<tr>
<td>MAP1_VERTEX_4</td>
<td>(x, y, z, w) vertex coordinates</td>
</tr>
<tr>
<td>MAP1_INDEX</td>
<td>color index</td>
</tr>
<tr>
<td>MAP1_COLOR_4</td>
<td>R &lt; G &lt; B &lt; A</td>
</tr>
<tr>
<td>MAP1_NORMAL</td>
<td>normal coordinates</td>
</tr>
<tr>
<td>MAP1_TEXTURE_COORD_1</td>
<td>s texture coordinates</td>
</tr>
<tr>
<td>MAP1_TEXTURE_COORD_2</td>
<td>s, t texture coordinates</td>
</tr>
<tr>
<td>MAP1_TEXTURE_COORD_3</td>
<td>s, t, r texture coordinates</td>
</tr>
<tr>
<td>MAP1_TEXTURE_COORD_4</td>
<td>s, t, r, q texture coordinates</td>
</tr>
</tbody>
</table>

The second two parameters (u1 and u2) define the range for the map. The stride value is the number of values in each block of storage (i.e., the offset between the beginning of one control point and the beginning of the next control point). The order should equal the degree of the curve plus one. The matrix holds the control points.

As an example, Map1(MAP1_VERTEX_3, 0, 1, 3, 4, <4x3 matrix>) is typical for setting the two end points and two control points to define a Bézier line.

The EvalCoord1 command evaluates a defined and enabled one-dimensional map.

`EvalCoord1(u)`

where u is the value of the domain coordinate.

You use the Mapgrid1 and EvalMesh1 commands to define and apply an evenly-spaced mesh.

`MapGrid1(un, u1, u2)`

sets up the mesh with un divisions spanning the range u1 to u2. Code is simplified by using the range 0 to 1.

`EvalMesh1(mode, i1, i2)`

actually generates the mesh from i1 to i2. The mode can be either POINT or LINE. The EvalMesh1 command makes its own Begin/End clause.

Alternatively, once you have set up MapGrid1, you can use EvalPoint1(i) to generate one vertex at a time. EvalCoord1 and EvalPoint1 commands should be surrounded by explicit Begin/End clauses.

The following example script demonstrates a one-dimensional outlier. A random set of control points draws a smooth curve. Only the first and last points are on the curve. Using NPOINTS=4 results in a cubic Bézier spline.

```
boxwide=500;
boxhigh=400;
gridsize=100; // bigger for finer divisions

NPOINTS = 4;
/* We suggest you use only values between 2 and 8 (inclusively). Numbers beyond these may be interpreted differently, depending on implementation. This value is the degree+1 of the fitted curve */
```
points = J(NPOINTS, 3, 0);
    // create an array of x,y,z triples
for( x = 1, x <= NPOINTS, x++,
    points[x, 1] = (x-1)/(NPOINTS-1) - .5;
    // x from -.5 to +.5
    points[x, 2] = randomuniform() - .5;
    // y is random in same range
    points[x, 3] = 0;
/* z is always zero, which causes the curve to stay in a plane */
);

spline = SceneBox(boxwide,boxhigh);
spline << ortho( -.6, .6, -.6, .6, -2, 2 );
    /* data from -.5 to .5 in x and y; this is a little larger */
spline<<Enable(MAP1_VERTEX_3);
spline<<MapGrid1(gridsize, 0, 1);
spline<<Color(.2,.2,1); // blue curve
spline<<MapI( MAP1_VERTEX_3, 0, 1, 3, NPOINTS, points );
spline<<LineWidth(2); // not-so-skinny curve
spline<<EvalMesh1(LINE, 0, gridsize ); // also try LINE, POINT
spline<<Color(.2, 1, .2);
spline<<PointSize(4); // big fat green points
    // show the points and label them
for( i=1, i <= NPOINTS, i++,
    spline<<Begin(POINTS);
    spline<<Vertex(points[i,1], points[i,2], points[i,3]);
    spline<<End;
    spline<<Push Matrix;
    spline<<Translate(points[i,1], points[i,2], points[i,3]);
    spline<<Text(center, bottom, .05,char(i));
    spline<<Pop Matrix;
);

New Window("Spline", spline);
http://www.tinaja.com/glib/bezconn.pdf offers an explanation of connecting cubic segments so that both the slope and the rate of change match at the connection point. This example does not illustrate doing so; there is only one segment here.

Two-Dimensional Evaluators
Two-dimensional evaluators follow their one dimensional counterparts, and are used in a similar way.
Map2(target, u1, u2, ustride, uorder, v1, v2, vstride, vorder, matrix)
Eval Coord2(u, v)
Values for the target parameter are the same as those shown in Table 9.5 “Map1 Target Parameters and Default Values,” p. 343 with Map1 replaced with Map2 appropriately. The u1, u2, v1, and v2 values specify the range of the two-dimensional mesh.

As an example, Map2(MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, <16x3 matrix>) is typical for setting the 16 points that define a Bézier surface.

Use the Mapgrid2 and EvalMesh2 commands to define and apply an evenly-spaced mesh.

MapGrid2(un, u1, u2, vn, v1, v2)

sets up the mesh with un and vn divisions spanning the range u1 to u2 and v1 to v2. Code is simplified by using ranges that span 0 to 1.

EvalMesh2(mode, i1, i2, j1, j2)

actually generates the mesh from i1 to i2 and j1 to j2. The mode can be POINT, LINE, or FILL. The EvalMesh2 command makes its own Begin/End clause.

Alternatively, once you have set up MapGrid2, you can use EvalPoint2(i, j) to generate one vertex at a time. EvalCoord2 and EvalPoint2 commands should be surrounded by explicit Begin/End clauses.

### Using the mouse

Mouse activity is supported through two feedback functions. The Patch Editor.jsl sample script uses these functions to support the dragging and dropping of points. Part of that script, the call-back function for mouse activity, is explained below. To run the script, open PatchEditor.jsl in the Scene3D folder inside the Sample Scripts folder.

```javascript
topClick2d = Function( {x, y, m, k},
  dragfunc( x, boxhigh - y, m, 1, 2 );
1;
);
frontClick2d = Function( {x, y, m, k},
  dragfunc( x, boxhigh - y, m, 1, 3 );
1;
);
rightClick2d = Function( {x, y, m, k},
  dragfunc( x, boxhigh - y, m, 2, 3 );
1;
);

Click3d = Function( {x, y, m, k, hitlist},
  If( m == 1,
    If( N Items( hitlist ) > 0,
      CurrentPoint = hitlist[1][3], /* first matrix in the list is the closest; 3rd element of matrix is ID*/
      CurrentPoint = 0
    );
    makePatch();
  );
0; /* only cares about initial mouse down. return 1 if drag, release is needed, but then arcball won't happen */
```
There are a pair of 2D functions and a pair of 3D functions. The first function is for clicking (mouse button is down) and the second is for tracking (mouse button is up). 2D functions are called before 3D functions.

When a 2D function is called, the arguments are \( X, Y, M, K \).

- \( X \) and \( Y \) are the coordinates of the mouse.
- \( M \) shows the state of the mouse and button. \( M=0 \) says the mouse button is up. \( M=1 \) says the button was just pressed. \( M=2 \) says the button is down and the mouse is moving. \( M=3 \) says the button was just released.
- \( K \) is related to the keys Shift, Alt, and control. \( K=1 \) for the Shift key, \( K=2 \) for the Control (command) key, and \( K=3 \) for the Alt (Option) key.

The 3D function is called similarly. The arguments are \( X, Y, M, K, hitlist \), where \( hitlist \) is a list of matrices

\[
[znear, zfar, id1, id2, id3, ...]
\]

\( znear, zfar \) is the Z distance from the camera of the near and far edge of the object. The matrices are sorted from near to far by the midpoint of \( znear, zfar \). The \( id \)s in the list are the \textit{pushname}, \textit{loadname}, and \textit{popname} values you just put in the display list.

The drag functions use a return value to tell if mouse processing should continue. That’s the trailing “1” you see in the functions. Anything else stops the mouse tracking. This is needed because the 2D and 3D functions do not run in parallel. You might want the 2D to return 0 and the 3D to return 1 so the tracking would happen in 3D rather than 2D.

**Pick Commands**

This \texttt{SceneBox} callback gets 2D mouse coordinates, then uses pick to determine the “named” object under the mouse. For example, \( hitlist \) is a 5x5 pixel pick box around \( x, y \); up to 1000 items returned, but just the leaf names. The format of the return is determined by the last parameter (1 returns a simple array, 0 returns a sorted (by depth) list of arrays).

\[
\text{Track2d} = \text{function(} \{x, y, m, k\},
\text{hitlist} = \text{theSceneBox} \ll \text{pick(} x, y, 5, 5, 1000, 1 \};
\text{if} ( \text{nrow(hitlist)} > 0, // something IS in the pick box
\text{... hitlist[1..n]} // are names you put in the display list
\});
\]
Contrast this with a call back `Track3d` function, where the pick rectangle is always 1x1 and picking only happens when the mouse moves. This is almost always what you want, but points are hard to pick because the 1x1 pick area is the same small size as the point. This function lets you pick without a mouse move.

The `Track3d` function always provides a depth-sorted list of arrays; each array may describe multiple names in a hierarchy (pushname/popname construct a hierarchy of objects). The sorting can be very slow when thousands of objects are selected. The final argument (1, above) controls whether the pick function will replace the sorted list of arrays with a simple array. The simple array will only contain the "leaf" names, not higher level names.

### Parameters

Parameters allow you to specify special modes and settings. To enable a parameter, use the `Enable(parameter)` command. To disable a parameter, use the `Disable(parameter)` command. Available parameters are shown in Table 9.6.

**Table 9.6 Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA_TEST</td>
<td>LIGHT5</td>
</tr>
<tr>
<td>AUTO_NORMAL</td>
<td>LIGHT6</td>
</tr>
<tr>
<td>BLEND</td>
<td>LIGHT7</td>
</tr>
<tr>
<td>CLIP_PLANE0</td>
<td>LIGHTING</td>
</tr>
<tr>
<td>CLIP_PLANE1</td>
<td>LINE_SMOOTH</td>
</tr>
<tr>
<td>CLIP_PLANE2</td>
<td>LINE_STIPPLE</td>
</tr>
<tr>
<td>CLIP_PLANE3</td>
<td>MAP1_COLOR_4</td>
</tr>
<tr>
<td>CLIP_PLANE4</td>
<td>MAP1_INDEX</td>
</tr>
<tr>
<td>CLIP_PLANE5</td>
<td>MAP1_NORMAL</td>
</tr>
<tr>
<td>COLOR_LOGIC_OP</td>
<td>MAP1_TEXTURE_COORD_1</td>
</tr>
<tr>
<td>COLOR_MATERIAL</td>
<td>MAP1_TEXTURE_COORD_2</td>
</tr>
<tr>
<td>CULL_FACE</td>
<td>MAP1_TEXTURE_COORD_3</td>
</tr>
<tr>
<td>DEPTH_TEST</td>
<td>MAP1_TEXTURE_COORD_4</td>
</tr>
<tr>
<td>DITHER</td>
<td>MAP1_VERTEX_3</td>
</tr>
<tr>
<td>FOG</td>
<td>MAP1_VERTEX_4</td>
</tr>
<tr>
<td>LIGHT0</td>
<td>MAP2_COLOR_4</td>
</tr>
<tr>
<td>LIGHT1</td>
<td>MAP2_INDEX</td>
</tr>
<tr>
<td>LIGHT2</td>
<td>MAP2_NORMAL</td>
</tr>
<tr>
<td>LIGHT3</td>
<td>MAP2_TEXTURE_COORD_1</td>
</tr>
<tr>
<td>LIGHT4</td>
<td>MAP2_TEXTURE_COORD_2</td>
</tr>
</tbody>
</table>
Matrix algebra is a compact notation to represent operations on matrices, which are two-dimensional arrays of numbers. JSL supports matrices as a data type, and implements many matrix operations. Most statistical methods are expressed in compact matrix notation, and you will find using matrices in JSL to be a powerful and compact way to program statistical expressions.

For example, least squares regression can be expressed compactly as an expression involving matrix multiplication and inversion:

\[ b = (X'X)^{-1}X'y \]

This could be implemented by a JSL expression that looks just like it:

\[ b = \text{Inv}(X' \times X) \times X'y; \]

**Notation**

This chapter follows the standard notational practice of representing a matrix with an uppercase bold variable (e.g., \( A \)). Often lowercase bold letters represent matrices that are vectors (e.g., \( x \)).
## Contents

- **Basics** ........................................................................................................ 351
- Definitions ........................................................................................................ 351
- Constructing matrices ..................................................................................... 351
- Numeric operators. .......................................................................................... 353
- Manipulating values ........................................................................................ 355
- Subscripts .......................................................................................................... 356
- Matrix operators. .............................................................................................. 360
- Solving linear systems ..................................................................................... 360
- Further construction functions ........................................................................ 363
- Decompositions and normalizations ............................................................... 365
- Build your own matrix operators .................................................................... 369
- Matrices with the rest of JMP. ......................................................................... 369
  - Matrices and data tables .............................................................................. 369
  - Matrices and reports ................................................................................... 370
- Statistical examples. ......................................................................................... 370
  - Regression example ..................................................................................... 370
  - ANOVA example .......................................................................................... 371

- **Basics** ........................................................................................................ 351
- Definitions ........................................................................................................ 351
- Constructing matrices ..................................................................................... 351
- Numeric operators. .......................................................................................... 353
- Manipulating values ........................................................................................ 355
- Subscripts .......................................................................................................... 356
- Matrix operators. .............................................................................................. 360
- Solving linear systems ..................................................................................... 360
- Further construction functions ........................................................................ 363
- Decompositions and normalizations ............................................................... 365
- Build your own matrix operators .................................................................... 369
- Matrices with the rest of JMP. ......................................................................... 369
  - Matrices and data tables .............................................................................. 369
  - Matrices and reports ................................................................................... 370
- Statistical examples. ......................................................................................... 370
  - Regression example ..................................................................................... 370
  - ANOVA example .......................................................................................... 371
Basics

Definitions

A matrix is a rectangular array of rows and columns of numbers. JSL supports matrix values and a number of matrix operators. Statistical methods are often implemented in matrix expressions.

Scalar is used in this chapter to refer to a simple non-matrix numeric value. If a matrix has only one row or one column, then it is commonly called a row vector or column vector, respectively, or sometimes just vector.

Some of this chapter reintroduces operators and functions that you already know from their use in scalar arithmetic. Pure numeric functions work elementwise on matrices, though there are exceptions, notably multiply and divide, which are not elementwise. The square root of a matrix is the matrix of square roots of the elements, but there are also several kinds of matrix roots available from other functions, Cholesky and Eigen and SVD.

Constructing matrices

Use brackets to specify a matrix literal, with the values separated by blanks (or other “white” characters) and the rows separated by either commas or semicolons. For example, a 4-row by 3-column matrix would be specified like this:

\[
A = \begin{bmatrix}
1 & 2 & 3, \\
4 & 5 & 6, \\
7 & 8 & 9, \\
10 & 11 & 12
\end{bmatrix}
\]

The numbers inside may have decimal parts, may be signed, and may be in scientific notation. They are internally stored in double precision floating point. Since white spaces are ignored in JSL, you can use as many spaces, tabs, and returns as you want to make matrices easier for you to read.

\[
r = [10\ 12\ 14]; \quad //\ row\ vector \\
c = [11,\ 13,\ 15]; \quad //\ column\ vector \\
b = [20]; \quad //\ 1-by-1\ matrix \\
e = []; \quad //\ empty\ matrix
\]

You can specify just signs to get matrices with +1 or –1 elements:

\[
B = [+\ +\ +, \\
+\ -\ -, \\
+\ -\ +]; \\
[1\ 1\ 1, \\
1\ -1\ -1, \\
1\ -1\ 1]
\]

Matrices can only contain numbers, unlike lists, which can contain expressions and other lists.
Constructing matrices from expressions

If you have numbers of numeric expressions stored in lists, you can convert it to a matrix by using the Matrix function. Just specify it as a list of expressions, or if you have several rows, a list of lists. e.g.:

\[
A = \text{matrix}([[1,2,3], [4,5,6], [7,8,9], [10,11,12]]);
\]

When specifying matrices in this way, elements can be expressions that resolve to numbers:

\[
\text{first} = [1,2,3]; \text{second} = [4,5,6]; \text{third} = [7,2*4,3^2];
\]

\[
A = \text{matrix}([\text{first}, \text{second}, \text{third},
\begin{bmatrix}
2*5, & \text{floor}(\exp(0)+\exp(1)+\exp(2)), & \text{floor}(\pi()^4) \\
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
10 & 11 & 12
\end{bmatrix}]);
\]

Special matrix constructors

The Identity function constructs an identity matrix of the dimension you specify. An identity matrix is a square matrix of zeros except for a diagonal of ones. The only argument specifies the dimension.

\[
\text{Identity}(3);
\]

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

The J function constructs a matrix with the number of rows and columns you specify as the first two arguments, whose elements are all the third argument, e.g.

\[
J(3,4,5);
\]

\[
\begin{bmatrix}
5 & 5 & 5 & 5 \\
5 & 5 & 5 & 5 \\
5 & 5 & 5 & 5
\end{bmatrix}
\]

The Index function generates a row vector of integers from the first argument to the last argument.

Double colon :: is an infix operator to do the same thing:

\[
6::10;
\]

\[
\begin{bmatrix}
6 & 7 & 8 & 9 & 10
\end{bmatrix}
\]

Index(1,5);

\[
\begin{bmatrix}
1 & 2 & 3 & 4 & 5
\end{bmatrix}
\]

The optional increment argument changes the default increment of +1.

index (0.1, 0.4, 0.1)

produces

\[
[0.1, 0.2, 0.3, 0.4]
\]

The increment can also be negative, so index(6, 0, -2) produces

\[
[6, 4, 2, 0]
\]

The default value of the increment is 1, or -1 if the first argument is higher than the second.

Note that :: preceding an argument (used as a prefix operator rather than an infix operator) is a scoping operator to force its argument to be interpreted as a JSL global variable, not a data table column, as discussed under “Step,” p. 100 in the “Programming Functions” chapter. To use :: for Index, you must place it between two arguments.
The Shape function reshapes an existing matrix across rows to be the specified dimensions. For example the following changes the 3x4 matrix \( a \) into a 12x1 matrix.

\[
a = \begin{bmatrix} 1 & 1 & 2 & 2 \ 2 & 3 & 3 & 4 \ 4 & 4 & 4 & 4 \end{bmatrix};
\]
\[
\text{shape}(a, 12, 1)
\]
\[
\begin{bmatrix} 1,1,1,2,2,2,3,3,3,4,4,4 \end{bmatrix}
\]

Deleting rows and columns

Deleting rows and columns is accomplished by these assignments:

\[
A[k, 0] = [] ; \quad \text{// to delete the kth row}
\]
\[
A[0, k] = [] ; \quad \text{// to delete the kth column}
\]

Inquiry functions

NCol and NRow return the number of columns and rows in a matrix (or data table), respectively:

\[
\text{NCol}([1 2 3, 4 5 6]) ; \quad \text{// returns 3, for 3 columns}
\]
\[
\text{NRow}([1 2 3, 4 5 6]) ; \quad \text{// returns 2, for 2 rows}
\]

To determine if a value is a matrix, use the Is Matrix function, which returns a 1 if the argument evaluates to a matrix:

\[
\text{if(IsMatrix}(A), \text{clauseForMatrix}, \text{elseClause}) ;
\]

Numeric operators

Basic arithmetic

You can add, subtract, multiply, and divide matrices, but be aware that the standard multiply operator becomes a matrix multiplier, rather than an elementwise multiplier.

\[
A = \begin{bmatrix} 1 & 2 & 3, 4 & 5 & 6, 7 & 8 & 9, 10 & 11 & 12 \end{bmatrix};
\]
\[
B = \begin{bmatrix} 0 & 1 & 2, 2 & 1 & 0, 0 & 1 & 1, 2 & 0 & 0 \end{bmatrix};
\]
\[
C = \begin{bmatrix} 1 & 2 & 3, 4 & 3 & 2 & 1, 0 & 1 & 0, 1 \end{bmatrix};
\]
\[
D = \begin{bmatrix} 0 & 1 & 2, 2 & 1 & 0, 1 & 2 & 0 \end{bmatrix};
\]
\[
\begin{bmatrix} 1 & 3 & 5, \\ 6 & 6 & 6, \\ 7 & 9 & 10, \\ 12 & 11 & 12 \end{bmatrix}
\]
\[
R = A + B ; \quad \text{// matrix addition}
\]
\[
\begin{bmatrix} 1 & 1 & 1, \\ 2 & 4 & 6, \\ 7 & 7 & 8, \\ 8 & 11 & 12 \end{bmatrix}
\]
\[
R = A - B ; \quad \text{// matrix subtraction}
\]

The usual \(^*\) and \(/\) infix operators, and the equivalent Multiply and Divide functions, do matrix multiplication and matrix division when given two matrix arguments. Given a matrix and a scalar, they perform elementwise multiplication and division. You can also use the Matrix Mult function to indicate matrix multiplication.
To force elementwise multiplication, use \( \ast \), or the equivalent `EMult` function. Recall that while multiplication of scalars is commutative (\( ab = ba \)), multiplication of matrices is not.

To force elementwise division, use \( \div \), or the equivalent `EDiv` function.

\[
R = A \ast C; // \text{ matrix multiplication,} \\
// \text{ inner product of rows of } A \text{ with columns of } C \\
\begin{bmatrix}
9 & 11 & 7 & 9 \\
24 & 29 & 22 & 27 \\
39 & 47 & 37 & 45 \\
54 & 65 & 52 & 63
\end{bmatrix}
\]

\[
R = A \div D; // \text{ matrix division,} \\
// \text{ equivalent to } A \ast \text{Inverse}(D) \\
\begin{bmatrix}
1.5 & 0.5 & 0 \\
3 & 2 & 0 \\
4.5 & 3.5 & 0 \\
6 & 5 & 0
\end{bmatrix}
\]

\[
R = A \ast \ast B; // \text{ elementwise multiplication} \\
\begin{bmatrix}
0 & 2 & 6 \\
8 & 5 & 0 \\
0 & 8 & 9 \\
20 & 0 & 0
\end{bmatrix}
\]

\[
R = C \ast 2; // \text{ scalar multiplication} \\
\begin{bmatrix}
2 & 4 & 6 & 8 \\
8 & 6 & 4 & 2 \\
0 & 2 & 0 & 2
\end{bmatrix}
\]

\[
R = C \div 2; // \text{ scalar division} \\
\begin{bmatrix}
0.5 & 1 & 1.5 & 2 \\
2 & 1.5 & 1 & 0.5 \\
0.5 & 0 & 0.5
\end{bmatrix}
\]

\[
R = A \div \div B; // \text{ elementwise division} \\
\begin{bmatrix}
. & 2 & 1.5 \\
2 & 5 & . \\
. & 8 & 9 \\
5 & . & .
\end{bmatrix}
\]

**Scalar numeric library functions**

The other arithmetic operators and functions work elementwise on matrices:

\[
B = \text{Sqrt}(A); // \text{ elementwise square root} \\
\begin{bmatrix}
1 & 1.414213562373095 & 1.732050807568877 \\
2 & 2.23606797749979 & 2.449489742783178 \\
2.645751311064591 & 2.82842712474619 & 3 \\
3.16227766016838 & 3.31662479035544 & 3.464101615137754
\end{bmatrix}
\]

Most of the pure numeric functions can be applied to matrices, resulting in a matrix of results. You can mix conformable matrix arguments with scalar arguments. Among the functions that work this way are:

- Sqrt, Root, Log, Exp, ^ Power, Log10
- Abs, Mod, Floor, Ceiling, Round, Modulo
- Sine, Cosine, Tangent, ArcSine, other trig functions...
- Normal Distribution, other probability functions...
Manipulating values

Concatenation

Concat combines two matrices side by side to form a larger matrix. The number of rows must agree. Double vertical bar (||) is an infix operator equivalent for horizontal concatenation.

\[
\text{Identity}(2)||\text{J}(2,3,4); \\
\begin{bmatrix}
1 & 0 & 4 & 4 & 4 \\
0 & 1 & 4 & 4 & 4
\end{bmatrix}
\]

\[
B=[1,1]; B || \text{Concat}(\text{Identity}(2),\text{J}(2,3,4)); \\
\begin{bmatrix}
1 & 1 & 0 & 4 & 4 & 4 \\
1 & 0 & 1 & 4 & 4 & 4
\end{bmatrix}
\]

VConcat stacks two matrices on top of each other to form a larger matrix. The number of columns must agree. Vertical-bar-slash (|/) is an infix operator equivalent for vertical concatenation.

\[
\text{Identity}(2) |/ \text{J}(3,2,1); // or VConcat(\text{Identity}(2),\text{J}(3,2,1)); \\
\begin{bmatrix}
1 & 0 \\
0 & 1 \\
1 & 1 \\
1 & 1 \\
1 & 1
\end{bmatrix}
\]

Both Concat and VConcat support concatenating to empty matrices, scalars, and lists, e.g.:

\[
a=[]; a || [1]; // yields [1] \\
a || [2]; // yields [2] \\
a || [3 4 5]; // yields [3 4 5]
\]

There are two in place concatenation operators ||= and |/=.

Diagonals

\[
\text{Diag} \text{ creates a diagonal matrix from a square matrix or a vector. A diagonal matrix is a square matrix whose nondiagonal elements are zero.}
\]

\[
D=[ 1 -1 1 ]; \\
\text{Diag}(D); \\
\begin{bmatrix}
1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

\[
\text{Diag} ([1,2,3,4]); \\
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 \\
0 & 0 & 3 & 0 \\
0 & 0 & 0 & 4
\end{bmatrix}
\]

\[
\text{Diag} ([1, 2, 3, 4], [5]); \\
\begin{bmatrix}
1 & 2 & 0 \\
3 & 4 & 0 \\
0 & 0 & 5
\end{bmatrix}
\]

VecDiag creates a vector from the diagonal elements of a matrix.

\[
v=\text{vecdiag}( \\
\begin{bmatrix}
1 & 0 & 0 & 1, \\
5 & 3 & 7 & -1
\end{bmatrix}
)
VecQuadratic is used to calculate the hats in regression that go into the standard errors of prediction or the Mahalanobis or $T^2$ statistics for outlier distances. The first argument is a symmetric matrix, usually an inverse covariance matrix; the second argument is a rectangular matrix with the same number of columns as the symmetric matrix argument.

It is equivalent to calculating \( \text{VecDiag}(X^\top X)^{-1} \).

\text{Trace} returns the sum of the diagonal elements for a square matrix:

\[
D = \begin{bmatrix} 0 & 1 & 2 \\ 2 & 1 & 0 \\ 1 & 2 & 0 \end{bmatrix};
\]

\text{trace}(D); // returns 1

\text{Transpose} transposes the rows and columns of a matrix. Back-quote (’’) is a postfix operator equivalent for \text{Transpose}, like the common prime or superscript-T notation (\( A' \) or \( A^\top \)).

\[
A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix};
\]

\( A' \);

\[
\begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}
\]

\text{Transpose}([1 2 3 4]);

\[
\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}
\]

\text{Matrices and data tables}

Messages to data tables or data table columns can either get data table values as matrices or write values from matrices into data tables. See the section “Converting between matrices and data tables,” p. 135 in the “Data Tables” chapter.

\text{Subscripts}

Use the subscript operator to pick out elements or submatrices from matrices. \text{Subscript} is usually written as a bracket notation after the matrix to be subscripted, with arguments for row and column.

\text{Single element}

The expression \( A[i,j] \) extracts the element in row \( i \), column \( j \), returning a scalar number. The equivalent functional form is \text{Subscript}(A,i,j).

\[
P = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix};
\]

\( P[2,3]; // \text{returns 6} \)
\( \text{Subscript}(P,2,3); // \text{equivalent to } P[2,3] \)

\text{Matrix or list subscripts}

You can also give lists or matrices of subscript values, and the result will be a matrix of the selected rows and columns. The following expressions select the 2nd and 3rd rows with the 2nd and 1st columns.
Matrices

Chapter 10

Basics

P[[2,3],[2,1]]; // matrix subscripts
P{{2,3},{2,1}}; // list subscripts

both resulting in:

[  5  4 ,
   8  7 ]

Single subscripts

A single subscript addresses matrices as if all the rows were turned into columns connected end-to-end
in a single column. This makes the double subscript A[i,j] the same as the single subscript
A[(i-1)*ncol(A)+j].

Here is an assortment of ways to get the matrix [10,14,18] using single subscripts:

Q=[2 4 6,8 10 12,14 16 18];
Q[[5,7,9]];
Q[[5 7 9]];
ii=[5,7,9]; Q[ii];
ii={5,7,9}; Q[ii];
Subscript(Q,ii);

This script returns the values 1–9 from the matrix P in order:

for(i=1,i<=3,i++,
   for(j=1,j<=3,j++,
      show(P[i,j])));

Selecting whole rows or columns

A subscript argument of zero will select all rows or columns. For example:

P[0,2];      // results in column 2: [2,5,8]
P[0,[3,2]];  // results in columns 3 and 2: [3 2,6,5,9,8]
P[3,0];      // results in row 3: [7,8,9]
P[[2,3],0];  // results in rows 2 and 3: [4 5,6,7,8,9]

Assignment through subscripts

You can insert new values into matrices using subscripts. The subscripts can be single indices, matrices
or lists of indices, or the zero index representing all rows or columns. The number of selected rows and
columns for the insertion must either match the dimension of the inserted argument, or the argument
can be a scalar inserted repeatedly into the indexed positions.

P[2,3]=99;              // insert a 99 into row 2, column 3
p[[1 2],[2 3]]=[66 77,88 99];
   // result P is [1 66,77,4 88,99,7 8,9]
P[0,2] = [11,22,33];    // inserts the three values into column 2
p[3,0] = [100 102,104]; // inserts the three values into column 3
p[2,0]=99;              // replaces the second row with all 99s

Operator assignment

You can use operator-assignment (such as +=) on matrices or subscripts of matrices. For example, the
following statement adds 1 to the i–j th element of the matrix:
P[i,j] += 1;

Operator assignments to subscripted matrices can map very generally. For example to multiply each negative element of a matrix by −1:

\[ A[\text{Loc}(A<0)] *= -1; \] // another way of doing A=Abs(A)

Ranges of rows or columns

It is common to need to work with a range of subscripts. The Index operator, mentioned previously and written with a double-colon, is used to create matrices of ranges.

\[ 4::7; \] // creates the matrix \[ 4 5 6 7 \]
\[ \text{Index}(4,7); \] // the equivalent function notation for 4::7
\[ T[1::3,10::14]; \] // refers to rows 1 to 3, columns 10 to 14

Loc

Loc creates a matrix of positions that locate where \( A \) is true, that is, nonzero and nonmissing.

\[ A = [0 \ 1 \ 0 \ 3 \ 4 \ 0]; \]
\[ I = \text{Loc}(A); \] // returns \( I \) as \( [2 \ 4 \ 5] \), the nonzero positions
\[ B = [2 \ 0 \ 0 \ 0 \ 5 \ 6]; \]
\[ I = \text{Loc}(A < B); \] // returns the indices where true, \( [1 \ 5 \ 6] \)

The following script replaces negative values in \( A \) with zero:

\[ A[\text{Loc}(A<0)]=0; \]

Loc Min and Loc Max

Loc Min and Loc Max return the position of the minimum and maximum elements of a matrix. Elements of a matrix are numbered consecutively, starting in the first row, first column, and proceeding left to right. For example,

\[ A = [1 \ 2 \ 2 \ 4 \ 4 \ 1 \ 1 \ 1]; \]
\[ B = [6 \ 12 \ 9]; \]
\[ \text{Show}( \text{Loc Max}( A )); \]
\[ \text{Show}( \text{Loc Min}( B )); \]

results in the output

\[ \text{Loc Max}(::A):5 \]
\[ \text{Loc Min}(::B):1 \]

Loc Sorted

Loc Sorted creates a matrix of positions in \( A \) that have values less than or equal to given values in \( B \). For example,

\[ A = [2 \ 4 \ 6 \ 8]; \]
\[ B = [2 \ 5 \ 8 \ 9]; \]
\[ I = \text{Loc Sorted}(A, B); \]

results in an \( I \) of

\[ [1, 2, 4, 4] \]
indexing the values 2, 4, 8, and 8 of A. The returned values are always 1 or greater. Loc Sorted is mainly used in conjunction with interpolation routines that need to do fast lookups for boundary values. Note that A must be sorted in ascending order.

**Comparisons, range checks, and logical operators**

JMP’s comparison, range check, and logical operators also work with matrices and produce matrices of elementwise boolean results, following all the usual rules described in “Comparison and Logical Operators,” p. 74 in the “JSL Operators” chapter. You can compare a matrix with a conformable matrix, or a matrix with a scalar.

```
A< B;  // less than
A<=B; // less or equal
A>B;  // greater than
A>=B; // greater or equal
A==B; // equal to
A!=B; // not equal to
A<B<C; // continued comparison (range check)
A|B;   // logical OR
A&B;   // logical AND
```

You can use the Any or All operators to summarize matrix comparison results. Any returns a 1 if any element is nonzero. All returns a 1 if all elements are nonzero.

```
[2 2]==[1 2]      // returns [0 1], therefore:
All([2 2]==[1 2]) // returns 0
Any([2 2]==[1 2]) // returns 1
```

Min or Max return the minimum or maximum element from the matrix or matrices given as arguments. Using the matrices A and B defined earlier:

```
A=[1 2 3, 4 5 6, 7 8 9, 10 11 12];
B=[0 1 2, 2 1 0, 0 1 1, 2 0 0];
min(A); // returns 1
max(A); // returns 12
min(A,B); // returns 0
```

**Ranking and sorting**

Rank returns a column vector of ranks for the elements of a row or column vector. These ranks can be used to produce a sorted column vector. For example:

```
E=[1 -2 3 -4 0 5 1 8 -7];
R=Rank(E); // returns [9,4,2,5,7,1,3,6,8]
sortedE=E[R]; // returns [-7,-4,-2,0,1,1,3,5,8]
```

Since single subscripts result in column vectors, if you want a sorted row vector, you will need to transpose the result, e.g. sortedE=E[r].

Ranking Tie(vector) returns a vector of ranks of the values of vector, with ranks for ties averaged. Similarly, Ranking(vector) returns a vector of ranks of the values of vector, low to high as 1 to n, but the ties are ranked arbitrarily.

You can also use Sort Ascending and Sort Descending to obtain a copy of the matrix supplied sorted as specified, as described for lists.
Summarizing Columns

There are several functions that return a row vector based on summary values for each column. For example, given this matrix:

\[
mymatrix = \begin{bmatrix} 11 & 22 \\ 33 & 44 \\ 55 & 66 \end{bmatrix}
\]

the following row vectors are returned:

- \( \text{V Max(mymatrix)} \):
  \[
  \begin{bmatrix} 55 \\ 66 \end{bmatrix}
  \]

- \( \text{V Min(mymatrix)} \):
  \[
  \begin{bmatrix} 11 \\ 22 \end{bmatrix}
  \]

- \( \text{V Mean(mymatrix)} \):
  \[
  \begin{bmatrix} 33 \\ 44 \end{bmatrix}
  \]

- \( \text{V Sum(mymatrix)} \):
  \[
  \begin{bmatrix} 99 \\ 132 \end{bmatrix}
  \]

- \( \text{V Std(mymatrix)} \):
  \[
  \begin{bmatrix} 22 \\ 22 \end{bmatrix}
  \]
The \texttt{GInverse} function accepts any matrix, including non-square ones, and uses singular-value decomposition to calculate the Moore-Penrose generalized inverse. This is useful in inverting a matrix that is not full rank. For example, a solution of the system
\begin{align*}
x + 2y + 2z &= 6 \\
2x + 4y + 4z &= 12 \\
x + y + z &= 1
\end{align*}
is found using the script
\begin{verbatim}
A=[1 2 2, 2 4 4, 1 1 1];
B=[6,12,1];
Show(GInverse(A)*B);
\end{verbatim}

\textbf{Solve}

\texttt{Solve} solves a linear system, that is it finds the vector \(x\) for the square, nonsingular matrix \(A\) and vector \(b\) you specify such that \(A\,x = b\). The matrix \(A\) and vector \(b\) must have the same number of rows.
\begin{verbatim}
A=[1 -4 2, 3 3 2, 0 4 -1];
b=[1, 2, 1];
x=solve(A,b);
// returns [-16.9999999999999, 4.99999999999998, 18.9999999999999]
A*x;
// returns [1, 2, 0.999999999999997]
\end{verbatim}
\texttt{Solve}(A,b) is the same as \texttt{Inverse}(A)*b, but \texttt{Solve} is more efficient and is preferred.

Note that results may be rounded differently on different machines.

\textbf{Sweep}

The \texttt{Sweep} function is a way to invert parts of a square matrix, identified by pivot indices, in such a way that if you sequence through all the pivot indices, you end up with the matrix inverse. Normally the matrix must be positive definite (or negative definite) so that the diagonal pivots never go to zero.

Suppose matrix \(E\) is composed of smaller matrix partitions, \(A\), \(B\), \(C\), and \(D\), to look like this:
\[
E = \begin{bmatrix}
A & B \\
C & D
\end{bmatrix}
\]
\[
E = (A||B) \\
/ (C||D);
\]

Then the command:
\begin{verbatim}
Sweep(E, [...]); // where [...] indicates partition A
\end{verbatim}
produces the matrix result equivalent to:
\[
\begin{bmatrix}
A^{-1} & A^{-1}B \\
-C^{-1} & D - C^{-1}B
\end{bmatrix}
\]
\[
(\text{inv}(A) || \text{inv}(A)*B) \\
/ \\
(-\text{C*inv}(A) || \text{D-C*inv}(A)*B)
\]
The submatrix in the A position becomes the inverse. The submatrix in the B position becomes the solution to \( Ax = B \). The submatrix in the C position becomes the solution to \( xA = C \).

One reason why Sweep is particularly useful is that it is sequential and reversible. That is, 

\[ A = \text{Sweep}(A, \{i, j\}) \] is the same as 

\[ A = \text{Sweep}(\text{Sweep}(A, i), j) \] : it is sequential. And 

\[ A = \text{Sweep}(\text{Sweep}(A, i), i) \] restores \( A \) to its original values: it is reversible.

If you have a cross-product matrix partitioned as follows:

\[
C = \begin{bmatrix}
X'X & X'y \\
'yX & y'y
\end{bmatrix}
\]

Then after sweeping through the indices of \( X'X \), the result is:

\[
\begin{bmatrix}
(X'X)^{-1} & (X'X)^{-1}X'y \\
'yX(X'X)^{-1} & y'y - X'y(X'X)^{-1}X'y
\end{bmatrix}
\]

The partitions are recognizable to statisticians as the least squares estimates for the model \( Y = Xb + e \) in the upper right, the sum of squared errors in the lower right, and a matrix proportional to the covariance of the estimates in the upper left. This property, when combined with the properties of being sequential and reversible, make the Sweep function very useful in doing the partial solutions needed for stepwise regression.

The index argument is a vector that lists the rows (or equivalently the columns) on which you want to sweep the matrix. For example, if \( E \) is a \( 4 \times 4 \) matrix, to sweep on all three rows to get \( E^{-1} \) requires these commands:

\[
E = \begin{bmatrix} 5 & 4 & 1 & 1 \\ 4 & 5 & 1 & 1 \\ 1 & 1 & 4 & 2 \\ 1 & 1 & 2 & 4 \end{bmatrix};
\]

\[
\text{sweep}(E, [1, 2, 3, 4]);
\]

\[
\begin{bmatrix}
0.56 & -0.44 & -0.02 & -0.02 \\
-0.44 & 0.56 & -0.02 & -0.02 \\
-0.02 & -0.02 & 0.34 & -0.16 \\
-0.02 & -0.02 & -0.16 & 0.34
\end{bmatrix}
\]

\[
\text{inverse}(E); \hspace{1em} \text{// notice that these are the same}
\]

\[
\begin{bmatrix}
0.56 & -0.44 & -0.02 & -0.02 \\
-0.44 & 0.56 & -0.02 & -0.02 \\
-0.02 & -0.02 & 0.34 & -0.16 \\
-0.02 & -0.02 & -0.16 & 0.34
\end{bmatrix}
\]


The Sweep operator does not check whether the matrix given is positive definite. If the matrix is not positive definite, then it will work as long as no zero pivot diagonals are encountered. If zero (or near-zero) pivot diagonals are encountered on a full sweep, then the result will be a g2 generalized inverse if the zero pivot row and column are zeroed.

Sweep is further demonstrated in the “ANOVA example,” p. 371.
Determinant

\text{Det} \text{ returns the determinant of a square matrix. The determinant of a } 2 \times 2 \text{ matrix is the difference of the diagonal products, as demonstrated below. Determinants for } n \times n \text{ matrices are defined recursively as a weighted sum of determinants for } (n-1) \times (n-1) \text{ matrices. The determinant must be nonzero for the matrix to have an inverse.}

\[
\begin{vmatrix}
1 & 2 \\
3 & 5
\end{vmatrix}
= (1 \cdot 5) - (3 \cdot 2) = -1
\]

\text{F}=[1 \ 2, 3 \ 5];
\text{Det(F); // returns -1}

Further construction functions

Design matrices

\text{Design} \text{ creates a matrix of design columns for a vector, one column for each unique value in the vector, with indicator values for that value. The design columns have elements 0 and 1. For instance, } x \text{ below has values 1, 2, and 3, then the design matrix has a column for 1's, a column for 2's, and a column for 3's. Each row of the matrix has a 1 in the column representing that row's value. So, the first row (1) has a 1 in the 1s column (the first column) and 0s elsewhere; the second row (2) has a 1 in the 2's column and 0s elsewhere; and so on. A design matrix has as many columns as the vector has unique values.}

\text{x}=[1, 2, 3, 2, 1];
\text{Design(x);}
\text{[ 1 0 0,}
\text{ 0 1 0,}
\text{ 0 0 1,}
\text{ 0 1 0,}
\text{ 1 0 0]}

A variation is \text{DesignNom}, which removes the last column and subtracts it from the others. Thus, the elements of \text{DesignNom} \text{ matrices are 0, 1, and -1, and the } \text{DesignNom} \text{ matrix has one fewer columns than the vector has unique values. This operator is used to make full-rank versions of design matrices for effects.}

\text{DesignNom(x);}
\text{[ 1 0,}
\text{ 0 1,}
\text{ -1 -1,}
\text{ 0 1,}
\text{ 1 0]}

\text{DesignNom} \text{ is further demonstrated in the “ANOVA example,” p. 371.}

In order to facilitate ordinal factor coding, a third Design function, \text{DesignOrd()}, which functions like \text{Design()} but produces a full-rank coding such that the row for the low level is all zero, and succeeding levels pack one more 1 across the row of the design matrix.

\text{Design, DesignNom, and DesignOrd} \text{ support a second argument that specifies the levels to be looked up and their order. This feature allows design matrices to be created one row at a time.}
result, as with the one-argument version, is a design matrix of 0’s and 1’s, each column indicating the
level. In the case of DesignNom, -1’s appear in all columns indicating the last level.

\[
\begin{align*}
  r &= \text{Design(values, levels);} \quad \text{//create a design matrix of indicator columns} \\
  r &= \text{DesignNom(values, levels);} \quad \text{//create a full-rank design matrix of} \\
  & \quad \text{indicator columns}
\end{align*}
\]

The levels argument is a list or matrix of levels to be looked up. The values argument is a numeric or
character value, matrix of numeric values, or list of numeric or character values.

The result will have the same number of rows as there are elements in the values argument. If there is
only one value, the result has one row. Also, the result always has the same number of columns as there
are items in the levels argument, or in the case of DesignNom, one less.

If a value is not found, the whole row will be zero.

\[
\begin{align*}
  \text{Design}(20, [10 \ 20 \ 30]); & \\
  & [0 \ 1 \ 0] \\
  \text{Design}(30, [10 \ 20 \ 30]); & \\
  & [0 \ 0 \ 1] \\
  \text{DesignNom}(20, [10 \ 20 \ 30]); & \\
  & [0 \ 1] \\
  \text{DesignNom}(30, [10 \ 20 \ 30]); & \\
  & [-1 \ -1] \\
  \text{DesignOrd}(20, [10 \ 20 \ 30]); & \\
  & [1 \ 0] \\
  \text{Design}([20, 10, 30, 20], [10 \ 20 \ 30]) & \\
  & \begin{bmatrix}
    0 & 1 & 0 \\
    1 & 0 & 0 \\
    0 & 0 & 1 \\
    0 & 1 & 0
  \end{bmatrix} \\
  \text{Design}({"b", "a", "c", "b"}, {"a", "b", "c"}); & \\
  & \begin{bmatrix}
    0 & 1 & 0 \\
    1 & 0 & 0 \\
    0 & 0 & 1 \\
    0 & 1 & 0
  \end{bmatrix}
\end{align*}
\]

**Direct products**

**Direct Product** finds the direct product (or Kronecker product) of two square matrices or scalars.

The direct product of an \( m \times m \) matrix and an \( n \times n \) matrix is the \( mn \times mn \) matrix whose elements are
the products of terms, one from \( A \) and one from \( B \). The direct product of two incidence matrices can
be useful in determining the existence of certain classes of design. For example:

\[
\begin{align*}
  G &= [1 \ 2, \\
      & 3 \ 5]; \\
  H &= [2 \ 3, \\
      & 5 \ 7]; \\
  \text{Direct product}(G, H); & \\
  & \begin{bmatrix}
    2 & 3 & 4 & 6, \\
    5 & 7 & 10 & 14, \\
    6 & 9 & 10 & 15, \\
    15 & 21 & 25 & 35
  \end{bmatrix}
\end{align*}
\]

**H Direct Product** finds the horizontal direct product, which is the row-by-row direct product of
two matrices with the same number of rows or scalars. For example:
Matrix operators

```math
HDirectProduct(G,H);
\[ \begin{bmatrix}
  2 & 3 & 4 & 6 \\
  15 & 21 & 25 & 35
\end{bmatrix} \]
```

*HDirect Product* is useful in constructing the design matrix columns for interactions.

```math
XA = Design Nom(A);
XB = Design Nom(B);
XAB = HDirectProduct(XA,XB);
X = J(NRow(A),1)||XA||XB||XAB;
```

**Decompositions and normalizations**

**Eigenvalues**

*Eigen* does eigenvalue decomposition for the nonsingular, symmetric matrix you specify, returning a list of matrices. The first matrix in the returned list is a vector of eigenvalues. The second matrix is a matrix of column eigenvectors.

```math
A = \begin{bmatrix}
  1 & -1 & -1 & -1 \\
  -1 & 1 & -1 & -1 \\
  -1 & -1 & 1 & -1 \\
  -1 & -1 & -1 & 1
\end{bmatrix};
```

```math
Eigen(A);
\{[2, 1.999999999999999, -2],
  [-0.70710678118655, -0.28867513459481, -0.40824829046386, 0.5,
   0.707106781186547, -0.28867513459481, -0.40824829046386, 0.5,
   -1.281975124e-16, -0.28867513459481, 0.816496580927726, 0.5,
   0, 0.866025403784439, 0 0.5]\}
```

Since the function returns a list of matrices, you might want to assign it to a list of two global variables, so that the vector of eigenvalues is assigned to one global and the matrix of eigenvectors to another:

```math
{evls, evcs} = Eigen(A);
```

Eigenvalue decomposition finds, for some \( n \times n \) matrix \( A \), all scalars \( \lambda \) (lambda) and vectors \( x \) such that the equation \( Ax = \lambda x \) has a nonzero solution \( x \). The \( \lambda \)'s are called eigenvalues, and the corresponding \( x \)'s are called eigenvectors. This is equivalent to solving \( (A - \lambda I)x = 0 \). You can reconstruct \( A \) from eigenvalues and eigenvectors by a statement such as:

```math
newA = evcs*diag(evls)*evcs';
```

The eigenvector matrices are orthonormal, such that the inverse is the transpose: \( E' E = EE' = I \).

Eigenvalues are uniquely determined only if the eigenvalues are unique. Zero eigenvalues correspond to singular matrices. Inverses can be obtained by inverting the eigenvalues and reconstituting with the eigenvectors. Moore-Penrose (see “GInverse,” p. 360) generalized inverses can be formed by inverting the non-zero eigenvalues and reconstituting, though in practice you have to use judgement as to whether a very small eigenvalue is effectively zero.

The eigenvalue decomposition allows one to view any square-matrix multiplication as a rotation (multiplication by an orthonormal matrix), a scaling (multiplication by a diagonal matrix), and a reverse rotation (multiplication by the orthonormal inverse, which is the transpose), or in notation:

```math
A*x = E'*diag(M)*E*x;  // E rotates, diag(M) scales, E' reverse-rotates
```
Eigenvalue decompositions are used in many statistical techniques, notably in principal components and canonical correlation, where the transformation associated with the largest eigenvalues are transformations that maximize variances.

**Cholesky decomposition**

Cholesky does Cholesky decomposition, where a positive definite matrix \( A \) is reexpressed as the product of a real nonsingular lower triangular matrix \( L \) and its transpose: \( A = LL' \).

```plaintext
E = [ 5 4 1 1, 4 5 1 1, 1 1 4 2, 1 1 2 4 ];
L = cholesky(E);  
[ 2.23606797749979 0 0 0,  
  1.78854381999832 1.34164078649874 0 0,  
  0.447213595499958 0.149071198499986 1.9436506316151 0,  
  0.447213595499958 0.149071198499986 0.914659120760047 1.714985851425088 ]
```

To verify the results:

```plaintext
L*L';
[ 5.000000000000000 4 1 1,  
  4 5 1 1,  
  1 1 4 2,  
  1 1 2 4 ]
```

Cholesky is often useful for reconstituting expressions into a more manageable form. For example, since eigenvalues are only available for symmetric matrices in JMP, then the eigenvalues of the product \( AB \) could not be obtained directly, because though \( A \) and \( B \) may be symmetric, the product is not. However, you can usually rephrase the problem in terms of eigenvalues of \( L'BL \) where \( L \) is the Cholesky root of \( A \), which has the same eigenvalues. Another use is in reordering matrices within \( \text{Trace} \) expressions. Expressions like \( \text{Trace}(A^*B^*A') \) may involve huge operations counts if \( A \) has lots of rows, but if \( B \) is small and can be factored into \( LL' \) by Cholesky, then it can be reformulated to \( \text{Trace}(A^*L^*L'*A') \) which is equal to \( \text{Trace}(L^*A^*AL) \), which is a much smaller number of operations, being just the sum of squares of the elements of \( AL \).

You can use the function \( \text{Chol Update()} \) to automatically update a Cholesky decomposition. If \( L \) is the cholesky root of an \( m \times n \) matrix \( A \), then after calling \( \text{cholUpdate}(L, C, V) \), \( L \) will be replaced with the cholesky root of \( A+V*C*V' \) where \( C \) is an \( m \times m \) symmetric matrix and \( V \) is an \( n \times m \) matrix.

Example:

```plaintext
// The inner product of a design matrix
exS = [16 1 0 11 -1 12, 1 11 -1 1 1 -1 10, 11 1 -1 -1 11 -1 9, -1 -1 -1 -1 1, 12 1 0 9 -1 12];  
// Conduct the Cholesky decomposition
exAchol = Cholesky( exS );
  
// Two column vectors to be applied to change the design matrix
exV = [1 0 0, 0 1, 0 0, 0 0, 0 1];  
/* The first column vector is added to one of the rows in the design matrix  
The second column vector is subtracted from one of the rows in the design matrix */
exC = [1 0, 0 -1];
```
// Update the Cholesky decomposition manually
exAnew = exS + exV * exC * exV';
exAcholnew = Cholesky(exAnew);

// Update the Cholesky decomposition more efficiently
exAcholnew_test = Chol Update(exAchol, exV, exC);

// Results are the same
Show(exAcholnew_test);
Show(exAcholnew);

Singular value decomposition

SVD finds the singular value decomposition of a matrix; that is, for a matrix A, SVD returns a list of three matrices U, M, and V such that \( U \cdot \text{diag}(M) \cdot V' = A \). Usually A is taller than wide, so that M is more compact, without extra zero diagonals. Singular value decomposition reexpresses A in the form \( U \cdot S \cdot V' \), where U and V’ are orthogonal vectors (perpendicular, statistically independent vectors) and S is an \( n \times n \) diagonal matrix containing the non-negative square roots of the eigenvalues of \( A'\cdot A \), the singular values of A. Singular value decomposition is the basis of correspondence analysis.

\[
A = \begin{bmatrix}
1 & 2 & 1 & 0 \\
2 & 3 & 0 & 1 \\
1 & 0 & 1 & 5 \\
0 & 1 & 5 & 1
\end{bmatrix}
\]

\{U, M, V\} = svd(A);
newA = U * diag(M) * V';

\[
\begin{bmatrix}
1 & 2 & 1 & 1.6895265e-15 \\
2 & 3 & 1.2632499e-15 & 1 \\
1 & 4.5864769e-16 & 1 & 5 \\
5.955833e-16 & 1 & 5 & 1
\end{bmatrix}
\]

Orthonormalization

Ortho does orthonormalization of the columns of a matrix using the Gram-Schmidt method. That is, it orthogonalizes the columns and then divides the vectors by their magnitudes to normalize them. The column vectors of orthogonal matrices are unit-length and are mutually perpendicular (their dot products are zero). For example:

\[
B = \text{ortho}([1 \ -1 \ 1 \ 0 \ 0 \ 1]);
\]

\[
\begin{bmatrix}
0.408248290463863 & -0.70710678118655 & 0.408248290463863 & 0.707106781186547 & -0.81649658092773 & 0.145090223e-16
\end{bmatrix}
\]

You can verify that these vectors are orthogonal by multiplying B by its transpose, which should yield the identity matrix.

\[
C = B' \cdot B;
\]

\[
\begin{bmatrix}
1 & -3.205621207e-16 & -3.205621207e-16
\end{bmatrix}
\]

By default, vectors are normalized. That is, they are divided by their magnitudes, which scales them to have length 1. Include the option Scaled(0) to turn scaling off:

\[
\text{ortho}([1 \ -1 \ 1 \ 0 \ 0 \ 1], \text{scaled}(0));
\]
Matrices
Chapter 10
Matrix operators

\[
\begin{bmatrix}
0.408248290463863 & -0.35355339059327,
0.408248290463863 & 0.353553390593274,
-0.81649658092773 & 1.572545112e-16
\end{bmatrix}
\]

Include the option `Centered(1)` to create vectors whose elements sum to zero. This is useful when constructing a matrix of contrasts.

```plaintext
result = ortho([1 -1,1 0,0 1], centered(1));
[ 0.408248290463863 -0.70710678118655,
 0.408248290463863 0.707106781186547,
-0.81649658092773 3.145090223e-16]
```

You can verify that the elements of each column sum to zero by premultiplying by a vector of ones to sum the columns:

```plaintext
J(1,3)*result;
[1.110223025e-16 2.034867198e-16]
```

Orthogonal polynomials

`OrthoPoly` returns orthogonal polynomials for a vector of indices representing spacings up to the order given. Orthogonal polynomials can be useful for fitting polynomial models where otherwise some regression coefficients might be highly correlated.

```plaintext
OrthoPoly([1 2 3],2);
[ -0.70710678118655 0.408248290463862,
 0 -0.81649658092773,
 0.707106781186547 0.408248290463864]
```

The order must be less than the dimension of the vector. A Scale option produces vectors of unit length, as described above for `Ortho`.

QR decomposition

`QR` factorization is useful for numerically stable matrix work. `QR` returns a list of two matrices. The typical usage is

```plaintext
{Q, R} = QR(X);
```

where Q and R hold the results. For an \( m \times n \) matrix, `QR` creates an orthogonal \( m \times m \) matrix Q and an upper triangular \( m \times n \) matrix R such that \( X = Q^R \).

Updating Inverse Matrices

If you have an inverse of an \( X'X \) matrix and want to recalculate it to add or drop a row of \( X \), then the operator `InvUpdt(S,x,w)` uses an updating formula to recalculate it efficiently. This is used in drop-1 influence diagnostics, and also in candidate design evaluation. The third argument is 1 to add a row or -1 to delete a row.

It is equivalent to calculating \( S^{-w}S^X(1+w*X'*S^X)*X'*S \) where \( w \) is the 1 or -1 for adding or deleting a row.
Build your own matrix operators

Recall from “Macros,” p. 423 in the “Advanced Concepts” chapter, that you can store your own operations in macros. So, for example, you can make your own matrix operation Norm to find the magnitude of a vector:

\[
\text{norm}=\text{function}({\mathbf{x}},\sqrt{\mathbf{x}^\top \mathbf{x}});
\]

You could similarly make Normalize to divide a vector by its magnitude:

\[
\text{normalize}=\text{function}({\mathbf{x}},\mathbf{x}/\sqrt{\mathbf{x}^\top \mathbf{x}});
\]

Matrices with the rest of JMP

Matrices and data tables

You can use several methods to move information between a matrix structure and a JMP data table. This way you can use JMP's capabilities for matrix algebra to perform your own calculations on numbers already stored in JMP data tables, and you can in turn save results back to data tables.

Moving data from a data table to a matrix

If \( \text{dt} \) is a variable containing a reference to a data table, then \text{Get As Matrix} will create a matrix \( \mathbf{A} \) with the values from all numeric columns in the data table:

\[
\text{dt}=\text{Open}("\$SAMPLE\_DATA/Big Class.jmp");
\]

\[
\mathbf{A} = \text{dt}<<\text{GetAsMatrix};
\]

If \( \text{col} \) is a variable containing a reference to a data column in a data table, then \text{Get As Matrix} will create a matrix \( \mathbf{A} \) as a column vector of values from the column.

\[
\text{col}=\text{Column}("Height");
\]

\[
\mathbf{A} = \text{col}<<\text{GetAsMatrix};
\]

Of course there are many other ways to do this, such as using For loops:

\[
\text{n=}\text{nrow(dt)};
\]

\[
\text{a} = \text{j(}n,1,0); \\
\text{for} (i=1,i<=n,i++,a[i]=\text{col}[i]);
\]

There are also commands to get certain rows of a data table. \text{GetAsMatrix} supports column list arguments, either names or character strings.

\[
\text{dt=}\text{current data table}();
\]

\[
\text{x=dt}<<\text{Get As Matrix}(\{"height", "weight"\});
\]

or

\[
\text{x=dt}<<\text{Get As Matrix}([\text{height, weight}]);
\]

\[
\text{dt}<<\text{Get Selected Rows} \text{ returns a matrix of the currently selected rows in the data table. Note that this can be an empty matrix.}
\]

\[
\text{dt}<<\text{Get Rows Where (expression) returns a matrix of row numbers where expression is true.}
\]
Matrices and reports

You can extract matrices of values from reports in windows. See “Manipulating displays,” p. 227 in the “Display Trees” chapter to learn how to refer to elements of reports, such as TableBoxes and ColumnBoxes.

If `tableBox` is a variable containing a reference to a table box, then `Get As Matrix` will create a matrix `A` with the values from all numeric columns in the table:

\[
A = \text{tableBox}<<\text{GetAsMatrix};
\]

If `colBox` is a variable containing a reference to a numeric column box in a report table, then `Get As Matrix` will create a matrix `A` as a column vector of values from the column.

\[
A = \text{colBox}<<\text{GetAsMatrix};
\]

For example, to obtain the table of parameter estimates in a bivariate report, you could submit this:

```julia
biv = Bivariate(X(height),Y(weight),Fit Line);
col = (biv<<report)["Linear Fit", "Parameter Estimates", ColumnBox("Estimate")];
beta = col<<GetAsMatrix;
[-127.145248610915,3.711354893859555]
```

Statistical examples

These examples demonstrate how to use JSL’s matrix algebra capabilities to compute statistics.

Regression example

Suppose that you want to implement your own regression calculation, rather than use the facilities built into JMP. Because of the compact matrix notation, it might only take a few lines of code:

```julia
Y = [98,112.5,84,102.5,102.5,50.5,90,77,112,150,128,133,85,112];
X = [65.3,69,56.5,62.8,63.5,51.3,64.3,56.3,66.5,72,64.8,67,57.5,66.5];
X = J(nrow(X),1) || X; // put in an intercept column of 1s
beta = Inv(X' * X) * Y; // the least square estimates
resid = Y-X*beta; // the residuals, Y - predicted
sse = resid' * resid; // sum of squared errors
show(beta,sse);
```
This could easily be expanded into a script that gets its data from a data table, calculates additional results, and shows the results in a report window:

```plaintext
// open the data table
bigClass = open("$SAMPLE_DATA/Big Class.jmp");

// get data into matrices
x = (Column("Age")<<getValues) || (Column("Height")<<getValues);
x = j(nrow(x),1,1)||x;
y = Column("Weight")<<getValues;

// regression calculations
xpxi = Inv(x`*x);
beta = xpxi*x`*y;       // parameter estimates
resid = y-x*beta;       // residuals
sse = resid`*resid;     // sum of squared errors
dfe = nrow(x)-ncol(x);  // degrees of freedom
mse = sse/dfe;          // mean square error, error variance estimate

// additional calculations on estimates
stdb = sqrt(vecDiag(xpxi)*mse); // standard errors of estimates
alpha = .05;
qt = Students t Quantile(1-alpha/2,dfe);
betau95 = beta+qt*stdb;         // upper 95% confidence limits
betal95 = beta-qt*stdb;         // lower 95% confidence limits
tratio = beta:/stdb;            // Student's T ratios
probt = (1-TDistribution(abs(tratio),dfe))*2; // p-values

// present results
newWindow("Big Class Regression",
         tableBox(
             StringColBox("Term",{"Intercept","Age","Height"}),
             NumberColBox("Estimate", beta),
             NumberColBox("Std Error", stdb),
             NumberColBox("TRatio", tratio),
             NumberColBox("Prob>|t|", probt),
             NumberColBox("Lower95%", betal95),
             NumberColBox("Upper95%", betau95)));
```

### ANOVA example

You can also implement your own one-way ANOVA. This example steps through this in some detail with a simple problem involving a three-level factor, perhaps indicating Low, Medium, and High doses, and a response measurement. Thus it solves the general linear model:

\[ Y = \alpha + bX + \varepsilon \]

where \( Y \) is a vector of responses, \( \alpha \) is the intercept term, \( b \) is a vector of coefficients, \( X \) is a design matrix for the factor, and \( \varepsilon \) is an error term.

```plaintext
factor=[1,2,3,1,2,3,1,2,3];
y=[1,2,3,4,3,2,5,4,3];
```
First you need to build a design matrix for the factor.

\[
designNom(factor);
\]

\[
\begin{bmatrix}
1 & 0 \\
0 & 1 \\
-1 & -1 \\
1 & 0 \\
0 & 1 \\
-1 & -1 \\
1 & 0 \\
0 & 1 \\
-1 & -1
\end{bmatrix}
\]

You also need to add a column of 1s to the design matrix, for the intercept term. You can do this by concatenating \( J \) and \( \text{Design Nom} \):

\[
x = J(9,1,1) || designNom(factor);
\]

\[
\begin{bmatrix}
1 & 1 & 0 \\
1 & 0 & 1 \\
1 & -1 & -1 \\
1 & 1 & 0 \\
1 & 0 & 1 \\
1 & -1 & -1 \\
\end{bmatrix}
\]

Now, to solve the normal equation, you need to construct a matrix \( M \) with partitions like this:

\[
\begin{bmatrix}
X'X & X'y \\
y'X & y'y
\end{bmatrix}
\]

You can do this in one step by concatenating the pieces:

\[
M=(X'x || x'y)
\]

\[
(y'x || y'y);
\]

\[
\begin{bmatrix}
9 & 0 & 0 & 27 \\
0 & 6 & 3 & 2 \\
0 & 3 & 6 & 1 \\
27 & 2 & 1 & 93
\end{bmatrix}
\]

Now you sweep \( M \) over all the columns in \( X'X \) for the full fit model, over the first column only for the intercept-only model, and over the second and third columns for the model without an intercept:

\[
\text{FullFit}=\text{sweep}(M,[1,2,3]);
\]

\[
\text{InterceptOnly}=\text{sweep}(M,[1]);
\]

\[
\text{FactorOnly}=\text{sweep}(M,[2,3]);
\]

Recall that some of the standard ANOVA results are calculated by comparing the results of these three models. This example focuses on the full fit model, which produces this swept matrix:

\[
\begin{bmatrix}
0.1111111111111111 & 0 & 0 & 3 \\
0 & 0.2222222222222222 & -0.1111111111111111 & 0.3333333333333333 \\
0 & -0.1111111111111111 & 0.2222222222222222 & 0 \\
-3 & -0.3333333333333333 & 0 & 11.333333333333333
\end{bmatrix}
\]

Read the model coefficients from the upper right partition of the matrix (notice that the lower left partition is the same except that the signs are reversed): 3, 0.333, 0. Thus the coefficient for the intercept
term is 3, the coefficient for the first level of the factor is 0.333, the coefficient for the second level is 0, and by virtue of the use of Design Nom, the coefficient for the third level is the difference, –0.333. The lower right partition of the matrix holds the sum of squares, 11.333.

You could modify this into a generalized ANOVA script by replacing some of the explicit values in the script with parameters.

These results match those from the Fit Model platform:

Here is how to construct this display. First build a data table, according to the methods described in the “Data Tables” chapter:

```julia
factor=[1,2,3,1,2,3,1,2,3];
Y=[1,2,3,4,3,2,5,4,3];
dt= new table("foo");
dt<<new column("y", Set Values(::y));
dt<<new column("factor", Nominal, Values(::factor));
```

Then run a model, as described in “Launching platforms,” p. 186 in the “Scripting Platforms” chapter.

```julia
Fit Model(Y(:y),Effects(:factor),Personality(Standard Least Squares),Run Model(y<<{Plot Actual by Predicted(0),Plot Residual by Predicted(0),Plot Effect Leverage(0)}));
```

Finally use JSL techniques for navigating displays, as discussed in “DisplayBox object references,” p. 230 in the “Display Trees” chapter.

```julia
ranova=report(Fit Least Squares[1]);
ranova["Analysis of Variance",columnbox("Sum of Squares")]<<select;
ranova["Parameter Estimates",columnbox("Estimate")]<<select;
```
This chapter discusses scripting features that are particularly useful for production settings, such as datafeeds for capturing real-time data from a laboratory instrument, scheduling scripted actions, and controlling JMP externally by OLE automation.

Some general JSL commands that might be of particular interest for use in a production setting include Caption, Speak, Print, Write, and Mail, which are described in the earlier chapter, “Functions that communicate with users,” p. 86 in the “JSL Operators” chapter.
## Contents

- Real-time data capture .................................................. 377
  - Create a datafeed object ........................................... 377
  - Manage a datafeed with messages ............................... 378
- Using sockets in JSL ..................................................... 381
  - Socket-related commands ........................................... 382
  - Messages for sockets ................................................ 383
- Database access (Windows and Linux only) ......................... 384
- Scheduling actions ....................................................... 385
- Working with SAS ......................................................... 387
  - Making a SAS DATA step ............................................ 387
  - SAS Variable Names ................................................ 387
  - Connecting to a SAS Metadata Server ....................... 387
- OLE Automation .......................................................... 391
  - Automating JMP through Visual Basic ................... 391
  - Automating JMP through Visual C++ .................... 398
Real-time data capture

A datafeed is a real-time method to read data continuously, such as from a laboratory measurement device connected to a serial port. A datafeed object sets up a concurrent thread with a queue for input lines that arrive in real time and are processed with background events. You set up scripts to interpret the lines and push the data to data tables, or do whatever else your process requires.

For example, submit this to get records from com1: and list them in the log.

```javascript
feed = Open DataFeed(
    Connect( Port("com1:"), Baud(9600), DataBits(7)),
    Set Script(print(feed<<getLine)));
```

A datafeed window shows status and offers controls:

Create a datafeed object

To create a DataFeed object, use the `Open DataFeed` command with arguments specifying details about the connection:

```javascript
feed = Open DataFeed( options );
```

No arguments are required. You can simply create the datafeed object and then send messages to it—to connect to a port, set up a script to process the data coming in, and so on. However, you would typically set up the basic operation of the data feed in the `Open DataFeed` command and subsequently send messages as needed to manage the data feed. Any of the options below work both as options inside `Open DataFeed` or as messages sent to a datafeed object.

It’s a good idea to store a reference to the object in a global variable, such as `feed` above, so that you can easily send messages to the object. You can store a reference to an existing object by using a subscript; for instance, to store a reference to the second data feed created:

```javascript
feed2 = Data Feed[2];
```

Options

(Windows only) To connect to a live data source, use `Connect` and specify details for the port. Each setting takes only one argument; in this syntax summary the symbol | between argument choices means “or.” The `Port` specification is needed if you want to connect; otherwise the object still works, but is not connected to a data feed. The last three items, DTR_DSR, RTS_CTS, and XON_XOFF, are bool- eans to specify which control characters are sent back and forth to indicate when the datafeed is ready to get data. Typically you would turn on at most one of them.

```javascript
feed = Open Data Feed(
```
This command creates a scriptable data feed object and stores a reference to it in the global variable `feed`. The `Connect` argument starts up a thread to watch a communications port and collect lines. The thread collects characters until it has a line, then appends it to the line queue and schedules an event to call the script.

```
Connect(
    Port( "com1:"|"com2:"|"lpt1:"|...),
    Baud( 9600|4800)...),
    Data Bits( 8|7 ),
    Parity( None|Odd|Even ),
    Stop Bits( 1|0|2 ),
    DTR_DSR( 0|1 ), // DataTerminalReady
    RTS_CTS( 0|1 ), // RequestToSend/ClearToSend
    XON_XOFF( 1|0 ) // TransmitterOn/TransmitterOff
);  
```

Set `Script` attaches a script to the datafeed object. This script is triggered by the `On DataFeed` handler whenever a line of data arrives. The argument for `Set Script` is simply the script to run, or a global containing a script.

```
feed=Open Data Feed(set script(myScript));
feed=Open Data Feed(set script( print(feed<<getLine) ));
```

A datafeed script typically uses `Get Line` to get a copy of one line and then does something with that line. Often it parses the line for data and adds it to some data table.

### Manage a datafeed with messages

A datafeed object responds to several messages, including `Connect` and `Set Script`, which are detailed above as arguments for `Open Datafeed` but can also be sent as messages to a data feed object that already exists:

```
feed<<connect(port(com1:), baud(4800), databits(7), parity(odd), stopbits(2));
feed<<set script(myScript);
```

The following messages could also be used as arguments to `On Data Feed`, but it would be more common to send them as messages to a data feed object that is already present.

You can send lines to a datafeed from a script. This is a quick way to test a datafeed. Include a text argument or a global that stores text:

```
feed<<Queue Line("14");
feed<<Queue Line(myValue);
```

Here's a test script to queue five lines of data:

```
feed<<queue line("11");
feed<<queue line("22");
```
To get the first line currently waiting in the queue, use a **Get Line** (singular) message. When you get a line, it is removed from the queue. Five lines were queued with the test script above, and **Get Line** returns the first line and removes it from the queue:

```plaintext
feed<<Get Line;
"11"
```

To empty all lines from the queue into a list, use **Get Lines** (plural). This returns the next four lines from the test script in list `{ }` format.

```plaintext
myList = feed<<GetLines;
{"22", "33", "44", "55"}
```

To stop and later restart the processing of queued lines, either click the **Stop** and **Restart** buttons in the datafeed window, or send the equivalent messages:

```plaintext
feed<<Stop;
feed<<Restart;
```

To close the datafeed and its window:

```plaintext
feed<<Close;
```

To disconnect from the live data source:

```plaintext
feed<<Disconnect;
```
Data-reading example

Here is a typical datafeed script. It expects to find a string 3 characters long starting in column 11. If it does, it uses it as a number and then adds a row to the data table in the column "thickness."

```plaintext
feed = Open Datafeed();
myScript = Expr(
    line = feed << Get Line;
    If( Length( line ) >= 14,
        x = Num( SubString( line, 11, 3 ) );
        If( !Is Missing( x ),
            Current Data Table() << Add Row( {thickness = x} )
        );
    );
);

Assign the script to the data feed object by using Set Script:

```

feed<<Set Script(myScript);
```

Control chart example

Here is a sample script that sets up a new data table and starts a control chart based on the data feed.

```plaintext
// make a data table with one column
dt = New Table("Gap Width");
dc = dt<<New Column("gap", Numeric, Best);

// set up control chart properties
dt<<Set Property("Control Limits", XBar(Avg(20), LCL(19.8), UCL(20.2)));
dt<<Set Property("Sigma", 0.1);

// make the data feed
feed = Open Datafeed();
feedScript = expr(
    line = feed<<get line;
    z = Num(line);
    Show(line, z); // if logging or debugging
    if (!IsMissing(z), dt<<AddRow({:gap = z}));
);

feed<<SetScript(feedScript);

// start the control chart
Control Chart(Sample Size(5), K Sigma(3), Chart Col(gap, XBar(Connect Points(1), Show Points(1), Show Center Line(1), Show Control Limits(1)),
    R(Connect Points(1), Show Points(1), Show Center Line(1), Show Control Limits(1))));

// Either start the feed from the device or test-feed some data
// to see it work (comment out one of the lines):
// feed<<connect(Port("com1:"), Baud(9600));
for(i=1;i<20,i++, feed<<Queue Line(Char(20+RandomUniform()*.1)));
```
Store the script in a data table

You can further automate the production setting by placing a datafeed script such as the one above in an On Open data table property. A property with this name is run automatically each time the table is opened (unless you set a preference to suppress execution). If you save such a data table as a Template, opening the template will run the datafeed script and save data to a new data table file. Also see “Control Chart,” p. 217 in the “Scripting Platforms” chapter.

Table 11.1 DataFeed messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Data</td>
<td>feed = Open DataFeed(commands)</td>
<td>Creates a data feed object. Any of the following can be used as commands inside Open DataFeed or sent as messages to an existing data feed object. Data Feed is a synonym.</td>
</tr>
<tr>
<td>Data Feed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set Script</td>
<td>Set Script(script)</td>
<td>Assigns the script that is run each time a line of data is received.</td>
</tr>
<tr>
<td>Show Window</td>
<td>Show Window(bool)</td>
<td>Specifies whether to show a status window for the data feed.</td>
</tr>
<tr>
<td>Get Line</td>
<td>feed&lt;&lt;Get Line</td>
<td>Returns and removes one line from the datafeed queue.</td>
</tr>
<tr>
<td>Get Lines</td>
<td>feed&lt;&lt;Get Lines</td>
<td>Returns as a list and removes all lines from the datafeed queue.</td>
</tr>
<tr>
<td>Queue Line</td>
<td>feed&lt;&lt;Queue Line(string)</td>
<td>Sends one line to the end of the datafeed queue.</td>
</tr>
<tr>
<td>Stop</td>
<td>feed&lt;&lt;Stop</td>
<td>Stops processing queued lines.</td>
</tr>
<tr>
<td>Restart</td>
<td>feed&lt;&lt;Restart</td>
<td>Restarts processing queued lines.</td>
</tr>
<tr>
<td>Close</td>
<td>feed&lt;&lt;Close</td>
<td>Closes the datafeed object and its window.</td>
</tr>
<tr>
<td>Connect</td>
<td>feed&lt;&lt;Connect(</td>
<td>Sets up port settings for the connection to the device. The symbol</td>
</tr>
<tr>
<td></td>
<td>Port(&quot;com1:&quot;</td>
<td>&quot;lpt1:&quot;</td>
</tr>
<tr>
<td>Disconnect</td>
<td>feed&lt;&lt;Disconnect</td>
<td>(Windows only) Disconnects the device from the datafeed queue but leaves the datafeed object active.</td>
</tr>
</tbody>
</table>

Using sockets in JSL

Another tool that may be useful in establishing a live datafeed is JSL Sockets. You can create two types of sockets using JSL:
Stream  Stream sockets create a reliable connection between JMP and another computer. The other computer may be running JMP, or it may be a vending machine, data collector, printer, or other device that is capable of socket communication. Some devices implement their interface as an HTTP web server.

Datagram  Datagram sockets create a less reliable connection between JMP and another computer. Datagrams are connectionless and information may arrive multiple times, not at all, and out of order. A datagram connection doesn't include all the overhead of a stream connection that does guarantee reliability. Since datagrams are connectionless, the destination address must be supplied each time (and for the same socket that can be different each time).

Once a socket is created, it can do two things: wait for a connection from another socket, or make a connection to another socket. Here is a simple example program that makes a connection to another computer's web server to get some data:

```julia
  tCall = socket();
  tCall << connect("www.jmp.com","80");
  tCall << send(CharToBlob("GET / HTTP/1.0~0d~0a~0d~0a"));
  tMessage = tCall << Recv(1000);
  text = BlobToChar( tMessage[3] );
  show(text);
  tCall << close();
```

The first line creates a socket and gives it a reference name (`tCall`). By default, a stream socket is created. You can designate the type of socket to create with an optional argument: `socket(stream)` or `socket(dgram)`.

The second line connects the `tCall` socket to port 80 (which is generally the HTTP port) of the JMP web site.

The third line sends a `GET` request to the JMP web server; this message tells the JMP web server to send the JMP home page back. The `/` that follows the word `GET` should be the path to the page to be opened. A `/` opens the root page.

The fourth line receives up to 1000 bytes from the JMP web server and stores a list of information in `tMessage`. Each socket call returns a list. The first element of the list is the name of the call, and the second is a text message which might be `ok` or a longer diagnostic message. Additional elements, if present, are specific to the call. In this case, the third element in the list is the data received.

The fifth line converts the binary information received into a character string. `tMessage[3]` is the third item in the list returned by `Recv`; it is the data from the JMP web server.

The sixth line displays the data in the log.

The last line closes the socket. The web server has already closed the far end, so this socket either needs reconnecting or proper disposal (`close`).

The Sample Scripts folder contains several examples of scripts using sockets.

### Socket-related commands

Before creating and using a socket, you may need to retrieve information about the end you want to connect to. `GetAddrInfo()` and `GetHostName()` each takes an address argument and an optional port argument and returns a list of information. For example:
Sometimes there may be more than one answer, in which case the sublist may be repeated one or more times. These functions can be quite slow; you probably should not try to build a data table of every web site name with it. For IPV6 compatibility you should generally use names like “www.sas.com” rather than the numerical form of an address.

Messages for sockets

Once you’ve created a socket with socket(), there are many messages you can send to it.

- **connect** Connects to a listening socket. Returns a list: {"connect", "ok"} if the connection was successful; or an error if not (for example, {"connect", "CONNREFUSED: The attempt to connect was forcefully rejected."}).
- **close** Closes the connection when you are finished with it. Returns a list (for example, {"Close", "ok"}).
- **send** Sends a STREAM message to the other end of the socket.
- **sendto** Sends a DGRAM message to the other end of the socket.
- **recv** Receives a STREAM message. The data comes back in a list, along with some other information. *Recv* takes a required numeric argument that specifies the number of bytes to accept.
- **recvfrom** Receives a DGRAM message.
- **ioctl** Controls the socket’s blocking behavior. By default, JMP sockets block if no data is available; the socket will not return control to the JSL program until data is available. This makes scripts easy to write, but not particularly robust if the remote end of the connection fails to supply the data. A socket that is set for non-blocking behavior always returns immediately, either with an ok return code and some data, or with a “WOULDBLOCK: ...” return code, which means if it were a blocking socket, it would have to wait (block progress of the next JSL statement) until data became available.

**Important:** Background operations that use a JSL callback avoid this issue; a socket used in a background *recv, recvfrom, or accept* is set to non-blocking and is polled during *wait* statements and when JMP is otherwise idle.

- **ioctl** returns a list; for example, {"ioctl", "ok"}, or {"ioctl", "NOTCONN: The socket is not connected."} if the socket has not been bound (see *bind*, below) or connected already.
- **bind** Tells the server socket what address the client socket will listen on. *Bind* associates a port on the local machine with the socket. This is required before a socket can *Listen* (see below). *Bind* is not usually used on sockets that will connect; the operating system will pick an unused port for you. *Bind* is needed for a server because anyone that wants to connect to the server needs to
know what port is being used. A common port is 80, the HTTP port. Bind returns a list; for example, {"bind", "ok"}, or {"bind", "ADDRNOTAVAIL: The specified address is not available from the local machine. "} if you try binding to a name that is not on your machine. Another socket can connect to this socket if it knows your machine name and the number.

listen tells the server socket to listen for connections. A listening socket is listening for connections from other sockets. You only need to put the socket into listen mode once. Accept (see below) is used to accept a connection from another socket. Listen returns a list; for example, {"listen", "ok"}, or {"listen", "INVAL: The socket is (or is not, depending on context) already bound to an address. or, Listen was not invoked prior to accept. or, Invalid host address. or, The socket has not been bound with Bind. "} if your bind call did not succeed.

accept tells the server socket to accept a connection and return a new connected socket. Accept returns a list stating what happened, and if successful, a new socket that is connected to the socket at the other end. For example, the returned list might be {"Accept", "ok", "localhost", socket()}. In this case localhost is the name of the machine that just connected, and the fourth parameter is a socket you can use to send or recv a message.

getpeername asks about the other end of a connection. GetPeerName returns a list with information about the other end's socket: {"getpeername", "ok", "127.0.0.1", "4087"}. If this is a server socket, you can discover the address and port of the client that connected. If this is a client socket, you re-discover the name and port of the server you used in the connect request.

getsockname asks about this end of a connection. GetSockName returns a list with information about this socket: {"getsockname", "ok", "localhost", "httpd"}. If this is a client socket, you can discover the port the operating system assigned. If this is a server socket, you already know this information from a bind you already made.

### Database access (Windows and Linux only)

JMP supports ODBC access to SQL databases through JSL with the Open Database command.

```
dt=Open Database(
   "Connect Dialog" | "DSN=...",                      // data source
   "sqlStatement" | "dataTableName" | "SQLFILE=...", // SQL statement
   "outputTableName"                                    // new table name
);
```

The first argument is a quoted connection string to specify the data source. It should be either "Connect Dialog" to display the ODBC connection dialog box, or "DSN=" and then the data source name and any other information needed to connect to the data source, such as:

```
"DSN=dBASE Files;DBQ=C:/Program Files/SAS/JMP7/Support Files English/
Sample Import Data;"
```

To see an example of what this string should look like for a particular database, open the log window, select Open from the File menu, pick the database source from the Files of Type combo box, select a table, and click Open. If a Log window is open, it will show the connection string.
The second argument is a double-quoted string that can be one of three different things:

1. An SQL statement to execute. For example, the second argument might be a SELECT statement in a quoted string like the following:
   "SELECT AGE, SEX, WEIGHT FROM BIGCLASS"
2. The name of a data table. In this case, the effect is an SQL "SELECT * FROM" statement for the data table indicated. For example, Open Database would in effect execute the statement "SELECT *
   FROM BIGCLASS" if you specify this for the second argument:
   "BIGCLASS"
3. "SQLFILE=" and a path to a text file containing an SQL statement to be executed. For example, with the following argument, JMP would attempt to open the file mySQLFile.txt from the C:\ directory and then execute the SQL statement in the file.
   "SQLFILE=C:\mySQLFile.txt"

The third argument, outputTableName, is optional and specifies the name of the output table to be created, if any. Note that Open Database does not always return a data table. The return value might be null. Whether it returns a data table depends on the kind of SQL statement executed. For example, a SELECT statement would return a data table, but a DROP TABLE statement would not.

To save a table back to a database through JSL, send the data table reference a Save Database() message:

```julia
  dt << Save Database("connectInfo", "TableName");
```

The first argument works the same way as it does in Open Database. Note that some databases will not allow you to save a table over one that already exists. If you want to replace a table in a database, use a DROP TABLE SQL statement in an Open Database command:

```julia
  Open Database ("connectinfo", "DROP TABLE TableName");
```

The following script opens a database with an SQL query, saves it back to the database under a new name, and then deletes the new table.

```julia
  dt = Open Database("Connect Dialog", "SELECT age, sex, weight FROM
  !"Bigclass!"", "My Big Class");
  dt << Save Database("Connect Dialog", "MY_BIG_CLASS");
  Open Database("Connect Dialog", "DROP TABLE BIGCLASS.MY_BIG_CLASS");
```

**Note:** When you import data from an ODBC database, a table variable is added that may contain user-id and password information. To prevent this from happening, set the following JSL-only preference:

```julia
  pref(ODBC Hide Connection String(1));
```

### Scheduling actions

A Schedule function lets you set up a script to be executed some number of seconds later.

```julia
  schedule(15, print("hello"));
```
A Scheduler window shows the time until the next event and has buttons for restarting (Go) or stopping (Stop) the schedule. Its pop-up menu has a command Show Schedule, which echoes the current schedule to the log. For instance, if you checked the schedule several times during the “hello” example, you would see something like this:

```
Scheduled at 11.55000000000018 :Print("hello")
Scheduled at 4.716666666666697 :Print("hello")
Scheduled at 3.083333333333485 :Print("hello")
```

The script might also be a name referring to a stored expression. For example, try submitting this script, which calls itself:

```
quickieScript = expr(show("Hi there"); schedule(15, quickieScript); );
quickieScript;
```

This script should show the string “Hi there” in a log window after 15 seconds, and reschedule itself for another 15 seconds, continuing until the Stop button is clicked.

More typically, in a production setting you might want to set up a schedule like this:

```
FifteenMinuteCheck = expr(show("Checking data");
  open("my file", options...);
  distribution(column(column1), capability(spec limits));
  schedule(15*60, FifteenMinuteCheck));
FifteenMinuteCheck;
```

Schedule initiates an event queue, but once it has the event queued, JMP proceeds with the next statement in the script. So, for example, the following has results that might surprise you:

```
schedule(3, print("one");
  print("two");
  "two"
  "one"
```

If you want the script to wait until the scheduled events are finished before proceeding, one solution would be to use Wait() with a suitable pause. Another is to embed the subsequent actions into the schedule queue. Schedule accepts a series of arguments to queue many events in sequence. Each event is a separate call to the schedule, and each event time is an absolute time relative to “now” (or the instant that Go is clicked)—so the following sequence finishes in five seconds, not in twelve:

```
schedule(3, print("hello");
schedule(4, print("", world");)
schedule(5, print("--bye");)
```

To cancel all events in a schedule queue, use Clear Schedule.

```
scheduler[1]<<Clear Schedule();
```

**Note:** It is not possible to create multiple threads using Schedule.
Working with SAS

JMP has several ways of interacting with the SAS System.

Making a SAS DATA step

Sending <<Make SAS Data Step to a data table returns the text for a SAS Data step that can recreate the data table in SAS. For example,

*Current Data Table() << Make SAS Data Step*

prints a data step to the log that can be used in the SAS Program Editor.

Sending <<Make SAS Data Step Window produces this code in a window with a .SAS suffix, so that it can be easily sent to SAS.

SAS Variable Names

SAS Open For Var Names() opens a SAS data set only to obtain the names of its variables, returning those names as a list of strings.

The rules for SAS variable names are more strict than those of JMP. The SAS Name function converts JMP variable names to SAS variable names, changing special characters and blanks to underscores, and various other transformations to produce a valid SAS name.

```
result = SAS Name(name);
result = SAS Name({list of names});
```

If the argument is a list of names, the result is a blank-separated character string of names. For example,

```
SAS Name({"x 1", "x 2"})
```

produces

```
  x_1 x_2
```

Connecting to a SAS Metadata Server

You can connect to a SAS server and work directly with SAS data sets. Making connections and interacting with SAS data sets is scriptable through JSL. For more information, see the JMP User Guide.
Making Connections

First, use the MetaConnect command to connect to a SAS metadata server:

```
connected = Meta Connect("MyMetadataServer", port)
```

If you supply only the machine name (for example, myserver.mycompany.com) and the port, you will be prompted to provide the authentication domain, your user name, and your password. You can also specify all that in JSL:

```
connected = Meta Connect("MyMetadataServer", port, “authdomain”, “username”, “password”)
```

When you are finished using the SAS metadata server, use Meta Disconnect() to disconnect the connection. No arguments are necessary; the command closes the current metadata server connection.

You can see the repositories that are available on a metadata server and set the one you want to use:

```
Meta Get Repositories();
{“Foundation”}
Meta Set Repository("Foundation”);
```

Note that if there is only one repository available, it will be selected automatically and you do not need to explicitly set it.

Once your repository is set, you can view the servers that are available:

```
mylist = Meta Get Servers();
{“SASMain”, ”Schroedl”, ”SASMain_ja”, ”SASMain_zh”, ”SASMain_ko”, ”SASMain_fr”, ”SASMain_de”, ”SASMain_Uncode”}
```

Next, set your SAS connection. You can also use this command to connect directly to a local or remote server instead of using a metadata server.

```
conn = SAS Connect("SASMain");
```

To see the libraries on the server, use the SAS Get Lib Ref command:

```
librefs = SAS Get Lib Refs();
```

If the library containing the data you want is not assigned, assign it:

```
librefs = SAS Assign Lib Refs("My Library", “c:\public\data”);
```

Opening SAS Data Sets

First, assign a SAS library reference:

```
SAS Assign Lib Refs("MyLib", “c:\public”);
```

The first argument is any name you want to use to refer to the library reference. The second is the path on the server where the data sets are located.

Next, get the list of data sets in the selected library:

```
datasets=SAS Get Data Sets("MyLib");
Now you can open a data set:

\[ \text{dt}=\text{SAS Import Data}("\text{BOOKS}, \text{"PURCHASES"}\); \]

or

\[ \text{dt}=\text{SAS Import Data}([\text{librefs[1]}, \text{datasets[12]}]); \]

or

\[ \text{dt}=\text{SAS Import Data}("\text{BOOKS.PURCHASES}"); \]

Once you have a reference to a library, you can get information about any SAS data sets in that library. For example, you can get a list of variables:

\[ \text{bookvars}=\text{SAS Get Var Names}("\text{BOOKS.PURCHASES}"); \]

\{"purchaseyear", "purchasemonth", "purchaseday", "bookid", "catid", "pubid", "price", "cost"\}

With that information, you can choose to import only part of the data set by specifying the variables to import.

\[ \text{dt}=\text{SAS Import Data}([\text{librefs[1]}, \text{datasets[12]}], \text{columns(bookvars[1], bookvars[2], bookvars[4]})]; \]

### Saving SAS Data Sets

To save a JMP data table or an imported SAS Data Set, use the SAS Export Data() command:

\[ \text{SAS Export Data}([\text{dt}, \text{librefs[1]}, \text{datasets[4]}], \text{ReplaceExisting}); \]

### Running a Stored Process

To get a reference to a stored process:

\[ \text{stp}=\text{Meta Get Stored Process}("\text{Samples/Stored Processes/Sample: Hello World}"); \]

There is no way to acquire a list of stored processes through JSL; you must know the path to the stored process you wish to run.

To run it, send the stored process a message:

\[ \text{stp}<<\text{run}(); \]

### Submitting SAS Code From JMP

You can also directly submit SAS code and get back SAS results. For example:

\[ \text{SAS Submit("proc print data=\text{sashelp.class}; run;")}; \]

Two optional arguments control whether you see the output and the SAS log in JMP:

\[ \text{SAS Submit("SAS Code" <,\text{No Output Window(\text{True|False})} <,\text{Get SAS Log(\text{True|False})});} \]

You can also see the SAS Log at any time using the command

\[ \text{SAS Get Log()}; \]

\text{SAS Get Log()} returns the contents, which can be placed in a JSL variable and used like any JSL string.
Sample Scripts

In your Sample Scripts folder is a folder named SAS Integration that contains examples. To run the stored process scripts successfully, the stored processes need to be placed on your SAS Metadata Server. The stored processes can be found in the sampleStoredProcesses.spk file, also in this folder.

To import sampleStoredProcesses.spk into your SAS Metadata Server:

Caution: We recommend that you import these stored processes into a SAS Metadata Server that is used for testing rather than into a production system.

1. Run SAS Management Console (SMC) and connect to your SAS Metadata Server using an account with administrative privileges.
2. Expand the BI Manager node in the left pane of the SMC and navigate to the folder in the tree under which you would like the imported sample stored processes to be created.
3. Right-click on that folder in either the left pane or the right pane of SMC and select Import. The Import Wizard appears.
4. Enter the full path to sampleStoredProcesses.spk or use the Browse button to navigate to it.
5. Select All Objects in the Import Options section of the wizard.
6. Click Next.

   The next panel reports that during the import process, you will need to specify values for Application servers and Source code repositories during the import process.
7. Click Next.

   The next panel allows you to select which of the application servers defined in your SAS Metadata Server you would like to use to execute the imported stored processes.
8. Select an application server from the drop-down list under Target.
9. Click Next.

   The next panel allows you to select the source code repository (directory) defined on your SAS Metadata Server where you would like the SAS code for the imported stored processes to be placed.
10. Select a source code repository from the drop-down list under Target Path.
11. Click Next.

   The next panel gives a summary of what will occur if you click Import.
12. Review the information on the panel, and if it looks correct, click Import.
13. During the Import process, you may be asked to provide login credentials for connecting to the metadata server to perform the import. Provide credentials with administrative privileges and click OK.

After the import completes, you will find a folder named BIP Tree under the folder that you imported the stored processes into. Under BIP Tree will be a folder named JMP Samples, and in the JMP Samples folder are two sample stored processes: Shoe Chart and Diameter.

Please note that the paths to the sample stored processes will need to be adjusted in the sample scripts storedProcessHTML.jsl and storedProcessJSL.jsl to match the folder into which you imported the sample stored processes in order to work correctly.
OLE Automation

Most of JMP can be driven through OLE automation. Please see the Automation Reference.pdf in JMP/8/Support Files English/Documentation for details on automating JMP. This document introduces how to automate JMP through Visual Basic and using Visual C++ with MFC. It also contains details for the methods and properties that JMP exposes to automation clients like Visual Basic and Visual C++.

You can find automation samples for Visual Basic, C++, and C# in JMP’s installation directory in JMP/8/Support Files English/Automation Samples.

Automating JMP through Visual Basic

Starting a JMP application

The first step in automating JMP is to start it up, but before that it's important to look at the resources available to help you with the JMP methods and properties. JMP provides a type library that allows automation controllers like Visual Basic (VB) to display a list of the methods and properties that JMP exposes, along with parameters that the methods require. This library is called JMP.TLB.

There are two steps to make JMP’s type library available to VB.
1. Select Project > References in VB. A list of applications that are known to VB will pop up. If JMP is not in that list, select Browse. A file dialog asks you to locate a .tlb (Type library) file. Look in the JMP directory and you will see an icon for the JMP type library. Select this library and click OK.
2. Now, bring up the object browser by selecting View > Object Browser in VB. Select JMP from the drop down list box.

Now you can see the JMP automation classes and constants. You can now select a class, and the methods available to that class will display in the right list box for the object browser. If you select a method, a short helper string will appear at the bottom of dialog. This string will list the parameters for the method. Constants are used when methods require a restricted set of parameters, typically denoting a specific action. For detailed information, see the Automation Reference document in your installation directory (Support Files English > Documentation).

Now that you have access to the type library information, write the necessary code to instantiate JMP. This is done with CreateObject. In global declarations for the VB project, create a variable of type JMP.Application. This is done as:

Dim MyJMP As JMP.Application

While you’re at it, dimension some other variables. Good examples are DataTable, Distrib, Oneway, and JMPDoc. These are specified with JMP.DataTable, JMP.Distribution, JMP.Oneway, and JMP.Document respectively.

To create a JMP session, make it visible, and load a data table, add the following code to your VB script.

Dim JMPDoc As JMP.Document
Set MyJMP = CreateObject("JMP.Application")
MyJMP.Visible = True
Set JMPDoc = MyJMP.OpenDocument("C:\\JMP\SAMPLE DATA\BIG CLASS.JMP")
The `Dim` statement indicates the type of variable. This declaration should go in the general declarations section of your VB project, though. If you don't do this, the JMP objects will be destroyed when the variable goes out of scope at the end of the procedure.

JMP comes up invisible by default, as required by automation guidelines. Therefore, one of your first moves should be to make it visible, as shown in the above code.

**Launching an analysis**

Now that you have a data table open, you can launch an analysis and manipulate it. Each analysis must first be created. Then, the required parameters for the analysis must be specified. Optional settings can also be specified. Then the analysis is launched. Additional option processing can then be done on the analysis object after the launch.

```vba
Dim Oneway As JMP.Oneway
Set Oneway = JMPDoc.CreateOneway
Oneway.LaunchAddY ("Height")
Oneway.LaunchAddX ("Age")
' Set an option before the launch
Oneway.Quantiles (True)
' Create the initial analysis output
Oneway.Launch
Oneway.MeansAnovaT (True)
Oneway.MeansStdDev (True)
Oneway.UnequalVariances (True)
Oneway.NormalQuantilePlot (True)
Oneway.SetAlpha (0.05)
Oneway.Save (oscCentered)
Oneway.Save (oscStandardized)
Oneway.CompareMeans occAllPairs, True
Oneway.CompareMeans occEachPair, True
```

The first step is to create the analysis object, which is done by calling the `CreateOneway` method of the document class. Next, X and Y columns are selected, and then `Launch` is called to create the actual One-way analysis. Each analysis platform has a distinct creation method, which you can view under the `Document` object in the object browser. In many cases, it is possible to specify options before the `Launch` of the object, so the analysis output will come up with options already set. In this example, most option processing is done after the launch of the analysis, which shows the options popup in the display. As you can see, most methods are a simple setting of options, like you might do from a menu. `SetAlpha` takes a parameter, since you don’t want to bring up a dialog for interaction during automation. `CompareMeans` takes two parameters, one for the type of comparison and one for the toggle to indicate on or off. The `Save` method takes a predefined constant (viewable in the object browser) that tells the Oneway analysis what to save.

Most analysis methods work this way, although some like Bivariate produce additional objects when methods are called. An example is:

```vba
Set Fit = Bivar.FitLine
Fit.ConfidenceFit (True)
Fit.ConfidenceIndividual (True)
```
Here, the FitLine method produces an object of type Fit. This object has methods and properties of its own, which can be manipulated. Remember, the new object created by FitLine can only be manipulated while its variable is in scope.

If a method produces an object that can also be automated, the object browser will indicate this. For FitLine, the object browser specifies that the return type is As Fit.

Since this isn't a predefined type like short or BSTR, you can probably guess that this is an object. If you look further down the object browser, you see Fit as an object type. This confirms that an object is produced, and also gives you the methods that Fit supports.

**Creating and populating a data table**

New data tables can be created with the (appropriately named) NewDataTable method of the Application object. A file name is assigned at creation time. This method returns a column object, which must be retained as long as you want to add rows. By default, 20 rows are created. The SetCellVal method can be used to populate individual cells, and AddRows can be used to add rows as needed. Here is an example:

```vba
Dim Col As Object
Set DT = JMP.NewDataTable("C:\test.jmp")
Set Col = DT.NewColumn("Col1", dtTypeNumeric, 0, 8)
DT.Visible = True

'You must add rows before populating the table with data
DT.AddRows 20,0

'Set Cell values to increments of 1.5
For i = 1 To 10
   Col.SetCellVal i, i * 1.5
Next i
DT.Visible = False
For i = 11 To 20
   Col.SetCellVal i, i * 1.5
Next i
DT.Visible = True

'This adds 5 rows to the end of the table
DT.AddRows 5, 0
'This adds 5 rows after row 2
DT.AddRows 5, 2

'Now save the data table using the previously specified file name
DT.Document.Save

'If you wanted to create a subset of the table, with only rows 1-3
'you could do the following
'Note: you could also create subsets using specific columns by adding the
'columns to a list using the AddToSubList member function of Datatable
Dim NewDT As JMP.Datatable
Dim DTDdoc As JMP.Document
DT.SelectRows 1,3
```
Set NewDT = DT.Subset

'Now save the new table
Set DTDoc = NewDT.Document
DTDoc.SaveAs("C:\MySubset.jmp")

Example programs

JMP's Support Files folder contains several example Visual Basic 5.0, Visual C#, and Visual C++ programs that automate features in JMP. You must have Visual Basic version 5 or 6 installed if you want to try them. It helps if the Microsoft Common Dialog DLL (commdlg.dll) is installed in your system path. The example programs can be loaded from VB by doing an Open Project on analysis.vbp, datatab.vbp and timport.vbp.

The ANALYSIS example program shows simple automation cases for almost all of the JMP platforms. The example code tests the features of a platform, but it does not pretend to do meaningful statistical analyses. Its purpose is for teaching automation coding. It is recommended that you make JMP's type library visible to the VB project. The first section of this document describes this process. This will allow you to see the methods and properties exposed by the automation platforms within JMP.

Likewise, the DATATAB example shows how to exercise the methods available for data table automation. No attempt is made to produce meaningful output.

The TIMPORT program shows the steps necessary to get a text file imported into JMP as a data table. Once this has been done, the data table can be manipulated just like the example in DATATAB, and analyses can be performed on the data just like in the ANALYSIS program.

The ODBCDEMO program shows a simple example of importing a dBase file into JMP using JMP's ODBC access.

The WORDEMO program shows the commands necessary to take a graphic section from a JMP report, copy it to the clipboard, and then insert it into a Microsoft Word document.

The sample code for all five example programs assumes the data files reside in the default SAMPLE DATA directory. If you move your sample data files, you will need to change the path information in the VB samples.

If there are differences between this document's examples of Visual Basic code and that in the sample programs, preference should be given to the sample program code.

Automating JMP From Excel 2000

This example automates JMP using a macro within an Excel 2000 worksheet. The macro code is written in Visual Basic. It starts JMP in a visible state when the Excel worksheet is initially opened. The Excel worksheet is then imported into JMP using the ODBC automation interface. Once the worksheet data is in JMP, changes to individual worksheet cells are sent to JMP and changed in the JMP data table.

The first time a row value in Excel changes, JMP generates a Control Chart. Subsequent changes to the excel worksheet result in changes to the Control Charts. This is because Control Chart output is dynamically linked to the JMP data table, which in this example is dynamically updated by Excel. Every fifth time the Excel worksheet changes, a method is called in JMP to generate a .PNG file for the Control Chart. This allows users without JMP to view the output through a web browser. Finally, when the Excel worksheet closes, JMP shuts down through automation.
Begin by opening Microsoft Excel. To create a Visual Basic script for an Excel workbook, select **Tools > Macro > Visual Basic Editor** from the Excel menu bar. The Visual Basic editor opens in a separate window. On the left of the Visual basic editor, there is a pane entitled **VBA Project** (shown here), which shows the sheets that may have Visual Basic code associated with them, as well as the workbook itself.

![VBA Project pane in Visual Basic Editor](image)

Code written for the workbook usually works for any of the sheets within the workbook.

There are three sections involved in the coding for this example. First, there are some variables that are global in scope that are declared in the module1.bas file. This allows these variables to be referenced in other code modules. A module can be inserted into the Visual Basic project by context-clicking on the VBA project icon and selecting **Insert > Module**. Type the following code into the module. The code declares instances of a JMP application, a JMP data table, and a flag to keep track of whether a document is open or not.

```vba
Public MyJMP as JMP.Application 'The JMP Application Object
Public DT As JMP.DataTable 'The JMP Data Table object
Public DocOpen as Boolean 'A flag indicating "JMP Table Open"
```

The next segment updates JMP when cells in the Excel worksheet change. It is called automatically because Excel generates the **Worksheet_change** event whenever a cell is changed, deleted, or added.

The Excel VBA Project Browser shows the sheets that are currently part of the workbook. The code below should be placed in the sheet that sends data to JMP. Double-click on the sheet icon in the VBA Project Window to bring up the code for that particular sheet.

```vba
Private Sub Worksheet_change(ByVal Target as Range)
    Dim Col as JMP.Column
    If(DocOpen) Then
        If(Target.Row = 1) Then
            Return
        End If
        If(DT.NumberRows < Target.Row - 1) Then
            DT.AddRows Target.Row - DT.NumberRows, 0
        End If
        If(Not IsArray(Target.Value) And Not IsEmpty(Target.Value)) Then
```
This code first checks to make sure JMP has a data table open. If the change is happening to the first row, then it is ignored because this is the column header in JMP. So, if a column name is changed in Excel, the corresponding change is not reflected in JMP. Code that would deal with heading changes could be inserted here, but is omitted in this example.

Next, if the row that has changed is beyond the number of rows that JMP is currently tracking in the data table, then the AddRows method is called to create more rows. Finally, if the operation is on a single value and doesn’t appear to signal a deletion, the JMP data table cell value is changed to the value that is passed into Worksheet_Change.

The main module is associated with the workbook. In the VBA Project Browser, the workbook code area is typically assigned the name ThisWorkbook, but this name can be easily changed. The following code goes into this area.

```
'Public(Global Variables) that all Workbook subroutines may access
Public Counter As Integer 'counter to update Control Chart every 5 changes
Public JMPDoc As JMP.Document 'instance of JMP Document
Public CChart As JMP.ControlChart 'instance of Control Chart
Public ChartOpen as Boolean 'Flag to set if chart is open

'Shut Down JMP before closing the workbook
Private Sub Workbook_BeforeClose(Cancel as Boolean)
    DocOpen = False
    MyJMP.Quit
End Sub

'As soon as the workbook is opened via File Open, load JMP for Automation
Private Sub Workbook_Open()
    Set MyJMP = CreateObject("JMP.Application") 'Create an instance of JMP
    MyJMP.Visible=True 'Make this instance of JMP visible
    Counter = 0 'initialize counter that counts changes
    DocOpen = False 'no document open yet
    ChartOpen = False 'no charts open yet, either

    Set JMPDoc = MyJMP.OpenDocument("C:\BOOK1.XLS") 'CHANGE THIS PATH TO POINT TO THE EXCEL WORKSHEET
    Set DT = JMPDoc.GetDataTable 'Create data table named DT
    DocOpen = True 'Set flag to say document is open
End Sub

'This is the most important part. After the first piece of data has been changed, generate a control chart. After every 5 changes to Excel worksheet cells, generate a new PNG of the Control Chart.
Private Sub Workbook_SheetChange(ByVal Sh As Object, ByVal Source As Range)
    Counter = Counter + 1
```
‘Save the control chart to a PNG every time 5 elements get updated
If (Counter Mod 5 = 0 Or Counter = 1) Then

‘If the Control Chart hasn’t been created yet, do so
If Not (ChartOpen) Then
    Set CChart = JMPDoc.CreateControlChart 'create chart
    CChart.LaunchAddProcess “Column 1” ‘Add column
    CChart.LaunchAddSampleUnitSize 5
    CChart.LaunchSetChartType jmpControlChartVar
    CChart.Launch 'launch the chart
    ChartOpen=True 'set flag to remember that a chart is open
EndIf

The Workbook_Open subroutine is called when the Excel table is initially loaded. It initializes some variables, starts JMP, and tells JMP to open (through ODBC) the same Excel file that is currently loaded into Excel 2000.

The Workbook_Change event is generated every time a user changes the data in any cell in any worksheet in the workbook. This sample assumes that there is only one active worksheet in the workbook. The first time that the user changes a cell value in the worksheet, the Workbook_Change subroutine creates a Control Chart in JMP using the current data table.

In this sample, the Workbook_Change subroutine also creates a PNG graphic file of the Control Chart output and updates it on the disk every fifth time a change is made to the workbook. This just gives some ideas on how Excel events and JMP automation can be used together to create output.

Finally, the Workbook_BeforeClose subroutine is invoked when the Excel workbook is closed, but before the window goes away. The code within this subroutine instructs JMP to close down as well.

Note that there are some limitations in this method. This example is good if the only activities that occur with the data are additions or changes. The Excel Worksheet_Change event is very limited in the reporting it provides. For example, it doesn’t give the type of action (deletion, change, drop) that caused the event. Worse still, the Excel documentation is incomplete. For instance, it says deletion doesn’t cause an event, but it actually does in practice. These problems make it hard to do cell-by-cell updating of a JMP data table in instances where deletion, drag/drop, or block replication needs support.

If these are problem cases, it is probably better to rely on a brute-force approach. One way is to reload the data into JMP every time a certain number of changes occur. An example is shown here.

Private Sub Workbook_SheetChange(ByVal Sh as Object, ByVal Source as Range)
    Counter = Counter + 1
    If (Counter Mod 10 = 0) Then
        ‘If there is a previous chart of Table opened, close it first
        If(DocOpen) Then
            JMPDoc.Close False, “”
            CChart.CloseWindow
        End If
        Set JMPDoc = MyJMP.OpenDocument(InstallDir + “C:\BOOK1.XLS”)
        Set DT = JMPDoc.GetDataTable
        DocOpen = True
    End If
End Sub
Now, create the control chart. This one is keyed to the data in “Column 1”. If 5 or more values are changed, JMP should generate a new Chart and save it as a PNG file to disk. The PNG file can be viewed with Internet Explorer.

Set CChart = JMPDoc.CreateControlChart
CChart.LaunchAddProcess “Column 1"
CChart.LaunchAddSampleUnitSize 5
CChart.LaunchSetChartType jmpControlChartVar
CChart.Launch
CChart.SaveGraphicsOutputAs “C:\ControlChart.png”, jmpPNG
EndIf
EndSub

This sample reloads the data every time there are 10 changes to the Excel Workbook. First, it removes JMP Control Charts and data tables that were previously created. Next, it loads the new data and creates a Control Chart.

This sample works best for small amounts of data. If huge Excel files are involved, this approach isn’t efficient because of the reloading of the table into JMP. These limitations stay in our mind during the JMP development process, and hopefully the Excel event support will be enhanced in the future, making these type of dynamic changes easier.

**Automating JMP through Visual C++**

Using C or C++ to create an automation client can be a long, tedious task. However, if you use the support provided by MFC in Microsoft Visual C++, the task is considerably easier. There are several steps that must be performed in order to get to a state where you can launch the automation server application (JMP in this case). The AutoClient application that is included in the Visual C++ Sample directory contains some code that provides ideas on how to get started. The Microsoft sample application CALCDRIV also shows a MFC based automation client. CALCDRIV is typically included with Visual C++, and on MSDN CD’s.

AutoClient shows how to start up JMP and drive a Bivariate analysis and the data table. The sample is much smaller than any of the Visual Basic samples, but the mechanics behind all the automation calls that you might want to use are the same as the examples with Bivariate and the data table. The following steps are based on the Visual C++ Version 5.0 UI.

**Steps for automating JMP**

1. Create your application, either manually or through App Wizard. Specify support for OLE automation. Even though you won’t be automating your own application, you need to include the OLE headers and initialization code. If you are retrofitting an existing application, you need to make sure that you include OLE support. This usually means including afxole.h in your application, and calling AfxOleInit() in your application InitInstance routine. Consult the MFC OLE documentation for details on this, though.

2. Bring up the Class Wizard and select the Automation tab. Select the Add Class drop down list and then the From a Type Library option. Navigate with the file dialog to the JMP install directory until you find JMPTLB. Select this type library.
3 You will be prompted with a dialog to confirm the classes that you want to use in your project. If you are unsure what objects (and interfaces) that you want, select them all by Shift-clicking. Select the names for the files where the class wizard will generate interface stubs and header information. Class Wizard is generating wrapper classes based on the MFC ColeDispatchDriver class. This will allow you easy access to the OLE `Invoke` automation function without having to know a lot of the technical details. Select OK. Class Wizard will generate the two files (.h and .cpp). You should include the .h file in whatever .cpp files will use the JMP automation objects, e.g. your View class implementation file.

4 The Class View of your Workspace will now show the Interface classes that you have imported. You can examine the methods and properties for each class through this class view.

5 To start JMP, define a variable of type `IJMPAutoApp` that will persist for the length of the automation session. Call `CreateDispatch` on this variable, passing in the JMP ProgID ("JMP.Application") as the lone parameter. At this point, when the code executes JMP will start.

6 Call `SetVisible(TRUE)` on the JMP object created in step 5. If you don't want to see JMP execute, don't do this step. However, for debugging it is necessary.

7 Now you can use the JMP application object to spawn further objects, which themselves can spawn more objects. The first thing you probably will want to do is load a Data table. To load an existing JMP data table, call the `OpenDocument` method on the JMP object created in step 5. If successful, this method will return a dispatch pointer that can be attached to an object of type `IJMPDoc` using the `AttachDispatch` method.

8 The `IJMPDoc` object provides the methods to launch the analysis and graphing platforms. Once you create an analysis and attach the dispatch pointer, you can specify the data table columns to use in the analysis and then you can launch it. Once the analysis is launched, you can manipulate it using the properties and methods specific to that particular type of analysis. Code that is taken from the sample application that describes steps 5–8 is shown below:

Example Program

```cpp
//Note, no error handling is done in this example
IJMPAutoApp m_DispDriver;
IJMPDoc m_Doc;
IAutoBivar m_Bivar;
IAutoFit m_FitLine;

//Create the initial dispatch driver that uses the IJMPAutoApp
//interface specification (taken from jmpauto.h)
m_DispDriver.CreateDispatch("JMP.Application");

if (m_DispDriver)
{
    //If JMP successfully started, make it visible
    m_DispDriver.SetVisible(TRUE);

    //Now open a data table as a document. The document interface
    //pointer that is returned is then attached to our Doc dispatch
    //driver class that uses the IJMPDoc interface specification.
    m_Doc.AttachDispatch(m_DispDriver.OpenDocument(
        "C:\\JMPDATA\\BIGCLASS.JMP");
```
} //First, call CreateBivariate on the Doc interface to create
//a dispatch object to a Bivariate analysis. If there is already
//a previous dispatch interface in m_Bivar, MFC will release it
//in AttachDispatch.
  m_Bivar.AttachDispatch(m_Doc.CreateBivariate());

  //Now add Height and Weight as the columns to analyze
  m_Bivar.LaunchAddX("Height");
  m_Bivar.LaunchAddY("Weight");

  //Launch the analysis
  m_Bivar.Launch();

  //Create a FitLine. Since the Fit can be automated, attach the dispatch
  //pointer that is returned from FitLine() to a DispatchDriver object
  m_FitLine.AttachDispatch(m_Bivar.FitLine());

  //Now do a few more fits. This example does not automate these fit
  //objects, although they do support automation.
  m_Bivar.FitPolynomial(3.0);
  m_Bivar.FitSpline(1000.0);

  //Now manipulate the first FitLine object
  m_FitLine.ConfidenceFit(TRUE);
  m_FitLine.ConfidenceIndividual(TRUE);
Chapter 12

Advanced Concepts

Complex scripting techniques and additional functions

This chapter includes advanced techniques, such as throwing and catching exceptions, defining your own functions, and using complex expressions.
## Advanced Programming Concepts

- Throwing and catching exceptions ........................................... 403
- Functions ........................................................................... 404
- Recursion ........................................................................... 405
- Includes ............................................................................ 405
- Loading and Saving Text Files ............................................. 405
- Dynamic Link Libraries (DLLs) (Windows Only) ...................... 406

## Encryption

- Encryption and Global Variables ......................................... 407

## XML Parsing Operations

- Example ............................................................................. 409

## Pattern Matching and Regular Expressions

- Patterns and Case .............................................................. 414
- Regular Expressions .......................................................... 414

## Lists and Expressions

- Stored expressions ............................................................ 415
- Macros .............................................................................. 423
- Manipulating lists .............................................................. 423
- Manipulating expressions ................................................... 425

## Projects ........................................................................... 428

## Hexadecimal and BLOB Functions .................................... 429

## Web ................................................................................. 431

## Additional Numeric Operators

- Matrices ............................................................................ 431
- Derivatives ....................................................................... 431
- Algebraic Manipulations .................................................... 433
- Maximize and Minimize ...................................................... 434
- Manipulating expressions .................................................... 425
Advanced Programming Concepts

Throwing and catching exceptions

A script can stop itself by executing the \texttt{Throw( )} function. If you want to escape from part of a script when it is in an error condition, you can enclose it in a \texttt{Try( )} expression.

\texttt{Try} takes two expression arguments. It starts by evaluating the first expression, and if or when the first expression throws an expression by evaluating \texttt{Throw}, it immediately stops evaluating that first expression, returns nothing, and evaluates the second expression.

\texttt{Throw} does not require an argument but has two kinds of optional arguments. If you include a character-valued expression as an argument, throwing stores that string in a global named \texttt{exception_msg}; this is illustrated in the first example below.

Examples

For example, you can use \texttt{Try} and \texttt{Throw} to escape from deep inside \texttt{For} loops.

\begin{verbatim}
   a = [1 2 3 , 4 5 , 7 8 9];
   b = a;
   nr = nrow(a);
   nc = ncol(a);
   //a[2,3]=2; //uncomment this line to see the "Missing b" outcome
   try(
      sum = 0;
      for(i=1,i<=nr,i++,
         for(j=1,j<=nc,j++,
            za = a[i,j]; if(isMissing(za),throw("Missing a"));
            zb = b[j,i]; if(isMissing(zb),throw("Missing b"));
            sum += za*zb;
         )
      ),
      show(i,j,exception_msg); throw();
   );
   i:2
e:3
exception_msg:"Missing a"
{0}
\end{verbatim}

You can also use \texttt{Try} and \texttt{Throw} to catch an exception that JMP itself throws:

\begin{verbatim}
   try(
      dt=open("My dataset.jmp"); // a file that can't be opened
      summarize( a =by(age),c=count,meanHt=mean(Height));
      show(a,c,meanHt),
      print("This script won't work without the data set"); throw();
   );
\end{verbatim}

You don't have to use \texttt{Try} to make use of \texttt{Throw}. In this example, \texttt{Throw} isn't caught by \texttt{Try} but still stops a script that can't proceed:

\begin{verbatim}
   dt=new table(); // to get an empty data table
\end{verbatim}
if (nrow(CurrentDataTable())==0, throw("!Empty Data Table");

Functions

JSL also has a function called Function to extend the macro concept with a local context arguments. Suppose that you want to create a function that takes the square root but tolerates negative arguments, returning zero rather than errors. You first specify the local arguments in a list with braces {} and then state the expression directly. You don't need to enclose the expression in Expr because Function stores it as an expression implicitly.

myRoot = function({x}, if(x>0, sqrt(x), 0));
a = myRoot(4);  // result in a is 2
b = myRoot(-1); // result in b is 0

Functions are stored in globals, the same as values. This means that you cannot have both a root function and a root value. It also means that you can redefine a function anytime except when you are inside the function itself.

When a function is called, its arguments are evaluated and given to the local variables specified in the list forming the first argument. Then the body of the function, the second argument, is evaluated.

The values of the arguments are for the temporary use of the function; when the function is exited, the values are discarded. The only value returned is the return value. If you want to return several values, then return a list instead of a single value.

Note: In defined functions, the stored function is not accessible directly, even by the Name Expr command. If you need to access the function expression in your script, you have to create the function within an expr() clause. For example,

makeFunction = expr(myRoot=function({x}, if (x>0, sqrt(x), 0)));
d = substitute(
  NameExpr(MakeFunction),
  expr(x), expr(y)
);
show(d);
makeFunction;

Local Symbols

You can declare variables as local to a function so that they do not affect the global symbol space. This is particularly useful for recursive functions, which need to keep separate the values of the local variables at each level of function call evaluation.

As shown above, a function definition looks as follows.

functionName=Function({arg1, ...}, body);

You can also have the function definition default all the unscoped names to be local.

functionName=Function({arg1, ...}, {Default Local}, body);

The use of Default Local localizes all the names that:

- Are not scoped as globals, e.g. ::name
- Are not scoped as data table column names, e.g. :name
- Occur without parentheses after them, e.g. are not of the form name(...)
For example, the following function sums three numbers.

```
add3 = Function({a, b, c}, {temp}, temp=a+b; temp+c);
X=add3(1, 5, 9);
```

The following function does the same thing, automatically finding locals.

```
add3 = Function({a, b, c}, {Default Local}, temp=a+b; temp+c);
X=add3(1, 5, 9);
```

In both cases, the variable `temp` is not a global, or, if it is already a global, remains untouched by evaluating the functions.

**Note:** If you use an expression initially as local, then use it as a global, JSL changes the context. However, an expression used globally stays resolved globally regardless of its future use.

### Recursion

The `Recurse` function makes a recursive call of the defining function. For example, you can make a function to calculate factorials, which are defined as the product of the number, the number minus 1, the number minus 2, and so on, down to 1.

```
myfactorial=function({a}, if (a==1, 1, a*recurse(a-1)));
myfactorial(5);
```

You can define recursive calculations without using `Recurse`. For instance you could replace `Recurse` by `myfactorial`, and the script would still work. However, `Recurse` offers several advantages: it avoids name conflicts when a local variable has the same name as the function, and `Recurse` allows you to recurse even if the function itself hasn’t been named (assigned to a global variable, such as `myfactorial` above).

### Includes

The `Include` function opens a script file, parses the script in it, and executes it.

```
include("pathname");
```

For example,

```
include("$SAMPLE_SCRIPTS/myStartupScript.jsl");
```

There is an option to obtain the parsed expression from the file, rather than evaluating the file.

```
include("pathname", <<Parse Only);
```

### Loading and Saving Text Files

The `Load Text File` and `Save Text File` commands allow manipulation of text files from JSL.

Note that the paths in the following code are strings.

```
text=Load Text File ("path");
Save Text File ("path", text);
```

You can load a text file from a website:
LoadTextFile("URL", <blob>);

The URL is a quoted string that contains the URL for the text file. The text file is returned as a string. If you add the optional named argument blob, a blob is returned instead.

Dynamic Link Libraries (DLLs) (Windows Only)

JSL can load a DLL and call a function exported by that DLL. There is one JSL command and two messages that implement this functionality:

   LoadDLL(pathToDLL)

LoadDLL loads the DLL indicated by pathToDLL.

   CallDLL(exportedFcn, signatureOfFcn, argumentToFcn)

The CallDLL message gets the proc address of exportedFcn from the DLL and then casts the fproc with a signature matching signatureOfFcn according to the Table 12.1 "signatureOfFcn values," p. 406 shown here:

<table>
<thead>
<tr>
<th>signatureOfFcn</th>
<th>parameter to exportedFcn is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;v&quot;</td>
<td>void</td>
</tr>
<tr>
<td>&quot;c&quot;</td>
<td>string</td>
</tr>
<tr>
<td>&quot;n&quot;</td>
<td>int</td>
</tr>
</tbody>
</table>

Then, it calls the fproc, passing in argumentToFcn as an argument to fproc.

   UnLoadDLL(void)

Finally, the UnLoadDLL message unloads the DLL.

The following code snippet shows how to use these functions.

```
// Load the DLL, given the specified path
dll = LoadDLL("D:\Release\MyDLL.DLL");
// Call the function inside the DLL named MyExportedFcn.
// This function takes one numeric as input and in this case,
// the value of the input parameter is to be 654.
dll << CallDLL("MyExportedFcn", "n", 654);
```

Effective, what these two lines accomplish is to call MyExportedFcn, passing 654 as input to the function. Conceptually, it is as if the JSL script executed MyExportedFcn(654);

```
// Unload the DLL, now that we are finished
dll << UnLoadDLL();
```

Note that the DLL functions are intended to call functions within a user's own DLL. They are not intended for calling operating system DLLs, since they only support passing one parameter (of the type shown in Table 12.1 "signatureOfFcn values," p. 406). So, for example, the Beep function in Kernel32.DLL would not be available because it takes two parameters.

However, an example of calling MessageBeep in User32.DLL is

```
dll = LoadDLL("C:\Winnt\System32\User32.DLL");
dll << CallDLL("MessageBeep", "n", 0);
```
Encryption

If you want to protect a JMP Scripting Language (JSL) file, you can encrypt it so only someone who knows the password can view it; you can also require a password to run it. This is useful in situations when you want to implement controlled sharing of a script.

To encrypt a script:
1. Open the script you want to encrypt.
2. Select Edit > Encrypt Script. The window shown in Figure 12.1 appears.

Figure 12.1 The Script Encryption Dialog

3. Assign password(s) to encrypt the files:
   - To encrypt a script so that a user can run it without a password, but needs a password to view it, supply JMP with only a decrypt password.
   - To encrypt a script so that a user must enter one password to run it and another password to view it, supply JMP with both a run and a decrypt password.

Note: The password must consist of single-byte characters; using a text Input Method Editor (IME) will not work.

The encrypted script appears in a new window, as shown in Figure 12.2.
Figure 12.2 Example of Encrypted Script

4 Save the encrypted script.

To view an encrypted JSL script:
1 Open the script in JMP.
2 Select Edit > Decrypt Script. The window in Figure 12.3 appears.

Figure 12.3 Decrypting a Script

3 Enter the decrypt password to see the script in a new window.

To run an encrypted JSL script:
1 Open the script in JMP.
2 Select Edit > Run Script. The window in Figure 12.4 appears.

Figure 12.4 Running an Encrypted Script

3 Enter the run password to run the script.

Note that entering the run password runs the script, but does not show the script: you must supply the decrypt password to actually view the script.

Encryption and Global Variables

Encryption alone does not hide global variables and their values. A Show Globals() command displays them normally. If you want to hide global variables in an encrypted script, you can give them special names.

Any global variable whose name begins with two underscore characters (__ is hidden, and Show Globals() will display neither its name nor its value. For example:

```plaintext
myvar = 2;
__myvar = 5;
```
Show Globals();
  //Globals

  myvar = 2;
  // 2 Globals (1 Hidden)

This strategy works whether your script is encrypted or not.

**XML Parsing Operations**

JSL has several commands available to parse XML.

Parse XML(string, On Element(“tagname”, Start Tag(expr), End Tag(expr)))
  parses an XML expression using the On Element expressions for specified XML tags.

value = xmlAttr(“attr name”)
  extracts the string value of an xml argument in the context of evaluating a Parse XML command.

value = xmlText()
  extracts the string text of the body of an XML tag in the context of evaluating a Parse XML command.

**Example**

The first part is an XML snippet of an Excel spreadsheet that has been saved as XML. This is one row of the Big Class data. This snippet can be saved as a valid XML document, which the JSL code will correctly parse into a one-row data table.
The second part is the JSL code that reads the XML file and creates a JMP data table with the information in it.

```xml
<Workbook xmlns="urn:schemas-microsoft-com:office:spreadsheet"
          xmlns:o="urn:schemas-microsoft-com:office:office"
          xmlns:x="urn:schemas-microsoft-com:office:excel"
          xmlns:ss="urn:schemas-microsoft-com:office:spreadsheet"
          xmlns:html="http://www.w3.org/TR/REC-html40">
  <Worksheet ss:Name="BigClass">
    <Table ss:ExpandedColumnCount="5" ss:ExpandedRowCount="41"
           x:FullColumns="1"
           x:FullRows="1">
      <Row>
        <Cell><Data ss:Type="String">name</Data></Cell>
        <Cell><Data ss:Type="String">age</Data></Cell>
        <Cell><Data ss:Type="String">sex</Data></Cell>
        <Cell><Data ss:Type="String">height</Data></Cell>
        <Cell><Data ss:Type="String">weight</Data></Cell>
      </Row>
      <Row>
        <Cell><Data ss:Type="String">KATIE</Data></Cell>
        <Cell><Data ss:Type="Number">12</Data></Cell>
        <Cell><Data ss:Type="String">F</Data></Cell>
        <Cell><Data ss:Type="Number">59</Data></Cell>
        <Cell><Data ss:Type="Number">95</Data></Cell>
      </Row>
    </Table>
  </Worksheet>
</Workbook>
```
file path = "bigclassexcel.xml";
file contents = load text file(file path);

Parse XML(file contents,

OnElement("urn:schemas-microsoft-com:office:spreadsheet^Worksheet",
    StartTag(
        sheetname=
        xmlAttr("urn:schemas-microsoft-com:office:spreadsheet^Name",
            "Untitled");
        dt = new table(sheetname);
        row = 1;
        col = 1;
    ),
),

OnElement("urn:schemas-microsoft-com:office:spreadsheet^Row",
    StartTag(
        if (row > 1,// assume first row is column names
            dt << Add Rows(1));
    ),
    EndTag(
        row++;    
        col = 1;
    )
),

OnElement("urn:schemas-microsoft-com:office:spreadsheet^Cell",
    EndTag(
        col++;
    )
),

OnElement("urn:schemas-microsoft-com:office:spreadsheet^Data",
    EndTag(
        data = xmlText(collapse);
        if (row == 1,
            New Column(data, Character(10)), // assume first row
            Column(col)[row-1] = data); // and other rows have data
    )
);
Pattern Matching and Regular Expressions

Pattern matching in JSL is a flexible method for searching and manipulating strings.

You define and use pattern variables just like any JMP variable:

```javascript
i = 3; // a numeric variable
a = "Ralph"; // a character variable
t = textbox("Madge"); // a display box variable
p = ("this" | "that") + patSpan(" ") + ("car" | "bus"); // a pattern variable
```

When the above statement executes, `p` is assigned a pattern value. The pattern value can be used either to construct another pattern or to perform a pattern match. The `patSpan` function returns a pattern that matches a span of characters specified in the argument; `patSpan("0123456789")` matches runs of digits.

```javascript
p2 = "Take " + p + "."; // using p to build another pattern
if( patMatch( "Take this bus.", p2 ), print("matches"), print("no match")
); // performing a match
```

Sometime all you need to know is that the pattern matched the source text, as above. Other times you’ll want to know what matched; for example, was it a bus or a car?

```javascript
p = ("this" | "that") + patSpan(" ") + ("car" | "bus") >? vehicleType;
// conditional assignment ONLY if pattern matches
if( patMatch( "Take this bus.", p ), show(vehicleType), print("no match")
); // don't use vehicleType in the ELSE because it is not set
```

You could pre-load `vehicleType` with a default value if you don’t want to check the outcome of the match with an `if`. The `?>` conditional assignment operator has two arguments, the first being a pattern and the second a JSL variable. `?>` constructs a pattern that matches the pattern (first argument) and stores the result of the match in the JSL variable (second argument) after the pattern succeeds. Similarly, `>>` doesn’t wait for the pattern to succeed. As soon (and as often) as the `>>` pattern matches, the assignment is performed.

```javascript
findDelimString = patLen(3)>>beginDelim + patArb()>?middlePart + expr(beginDelim);
testString = "SomeoneSawTheQuickBrownFoxJumpOverTheLazyDog'sBack";
rc = PatMatch( testString, findDelimString, "<<<" || middlePart || ">>>" );
show( rc, beginDelim, middlePart, testString );
```

The above example shows a third argument in the `patMatch` function: the replacement string. In this case, the replacement is formed from a concatenation (`||` operator) of three strings; one of the three strings, `middlePart`, was extracted from the `testString` by `?>`, which was appropriate because the replacement can’t occur unless the pattern match succeeds (`rc == 1`).

Look at the pattern assigned to `findDelimString`. It is a concatenation of 3 patterns. The first is a `>>` operator that matches 3 characters and assigns them to `beginDelim`. The second is a `?>` operator that matches an arbitrary number of characters and, when the entire match succeeds, assigns them to `middlePart`. The last is an unevaluated expression, consisting of whatever string is in `beginDelim` at the time the pattern is executing, _not_ at the time the pattern is built. Just like `expr()`, the evaluation of its argument is postponed. That makes the pattern hunt for two identical three letter delimiters of the middle part.
Other pattern functions may be faster and represent the problem you are trying to solve better than writing a lot of alternatives; for example, "a"|"b"|"c" is the same as patAny("abc"). The equivalent example for patNotAny("abc") is much harder. Similar to patSpan (above), patBreak("0123456789") will match up to, but not including, the first number.

Here's a pattern that matches numbers with decimals and exponents and signs. It also matches some degenerate cases with no digits; look at the pattern assigned to digits.

```plaintext
digits = patSpan("0123456789") | "";

number = ( patAny("+-") | "" ) >? signPart +
   ( digits ) >? wholePart +
   ( "." + digits | "" ) >? fractionPart +
   ( patAny("eEdD") + ( patAny("+-") | "" ) + digits | "" ) >?
   exponentPart;

if( patMatch( "-123.456e-78", number ), show( signPart, wholePart, fractionPart, exponentPart ) );
```

Sometimes data is in fixed fields. The patTab, patRTab, patLen, patPos, and patRPos functions make it easy to split out the fields in a fixed field string. PatTab and patRTab work from the left and right end of the string and take a number as their argument. They succeed by matching forward to the specified tab position. For example:

```plaintext
p = patPos(10) + patTab(15);
```

**PatPos(10)** matches the null string if it is in position 10; so at match time, the matcher works its way forward to position 10, then **patTab(15)** matches text from the current position (10) forward to position 15. This pattern is equivalent to **patPos(10)+patLen(5)**. Another example:

```plaintext
p = patPos(0) + patRTab(0);
```

This example matches the entire string, from 0 characters from the start to 0 characters from the end. the **patRem()** function takes no argument and is shorthand for **patRTab(0)**; it means the remainder of the string. Pattern matching can also be anchored to the beginning of the string like this:

```plaintext
patMatch( "now is the time", patLen(15) + patRPos(0), NULL, ANCHOR );
```

The above pattern uses NULL rather than a replacement value, and ANCHOR as an option. Both are uppercase, as shown. NULL means no replacement is done. ANCHOR means the match is anchored to the beginning of the string. The default value is UNANCHORED.

Patterns can be built up like this, but this is **not** recursive:

```plaintext
p = "a" | "b"; // matches one character
p = p + p; // two characters
p = p + p; // four characters
patMatch( "babbb", patPos(0) + p + patRPos(0) );
```

A recursive pattern refers to its current definition using **expr()**:

```plaintext
p = "<" + expr(p) + "*" + expr(p) + ">" | "x";
patMatch( "<x*x<x>x>*x", patPos(0) + p + patRPos(0) );
```

Remember, **expr()** is the procrastination operator; when the pattern is assigned to the variable p, **expr()** delays evaluating its argument (p) until later. In the next statement, **patMatch** performs the pattern match operation, and each time it encounters **expr()**, it looks for the current value of the
argument (which, in this example, does not change during the match). So, if $p$ is defined in terms of itself, how can this possibly work?

$p$ consists of two alternatives. The right hand choice is easy: a single letter $x$. The left side is harder: \(<p*p>\). Each $p$ could be a single letter $x$, since that is one of the choices $p$ could match, or it could be \(<p*p>\). The last few example have used $\text{patPos}(0) + \ldots + \text{patRPos}(0)$ to make sure the pattern matches the entire source text. Sometimes this is what you want, and sometimes you'd rather the pattern match a subtext. If you are experimenting with these examples by changing the source text, you’ll most likely want to match the entire string to easily tell what was matched. The result from $\text{patMatch}$ will be 0 or 1.

This example uses “Left” recursion:

```cpp
x = expr(x) + "a" | "b"; // + binds tighter than |
```

If the pattern is used in \textsc{fullscan} mode, it will eventually use up all memory as it expands; but, by default, the $\text{patMatch}$ function is not using \textsc{fullscan} and makes some assumptions that allow the recursion to stop and the match to succeed. The pattern matches either a “b”, or anything the pattern matches followed by an “a”.

```cpp
rc = \text{patMatch}( "baaaaa", x );
```

**Patterns and Case**

Unlike regular expressions, pattern matching is case insensitive. To force case sensitivity, you can add the named argument \textsc{matchcase} to either $\text{patMatch}$ or $\text{Regex\ Match}$. For example:

```cpp
string = "abcABC";

result = $\text{RegEx\ Match}(\ \text{string, Pat\ Regex( \"[aBc]+\")});$
Show(\ \text{string, result});
string:"abcABC"
result:{"abcABC"}

result = $\text{RegEx\ Match}(\ \text{string, Pat\ Regex( \"[cba]+\")}, \text{NULL, MATCHCASE});$
Show(\ \text{string, result});
string:"abcABC"
result:{"abc"}
```

**Regular Expressions**

You can also use standard regular expressions in JMP using the $\text{Regex()}$ function:

```cpp
$\text{Regex(source, regular expression, <format>, <ignorecase>);}$
```

$\text{Source}$ is the string you want to apply the regular expression to. Both arguments must be quoted strings or references to quoted strings.

An example:

```cpp
$\text{Regex(}
    "Are you there Alice?, asked Jerry.\\n    (here|there) (\w\+).+(said|asked) (\w\+).\\n\");$
```
Since no format was specified, all the text that matched the regular expression is returned. Use the optional format to create a new string using your source and the regular expression:

```regex
Regex(
    " Are you there Alice?, asked Jerry."
    " (here|there) (\w+).+(said|asked) (\w+)\.",
    " I am \1, \4, replied \2." );

    " I am there, Jerry, replied Alice."
```

Format defaults to \0, which is the entire match. \n specifies the nth match made by the regular expression. If the regular expression does not match anything in the source, Regex returns numeric missing.

Regular expressions are case sensitive. To perform a case-insensitive match, use the optional switch IGNORECASE.

**Tip:** For details on forming regular expressions, use a regular expressions reference.

## Lists and Expressions

### Stored expressions

An expression is something that can be evaluated. The first section of the chapter, “Context: meaning is local,” p. 40 in the “JSL Building Blocks” chapter, discussed how JMP evaluates expressions. Now you must consider when JMP evaluates expressions.

JMP tends to evaluate things as soon as it possibly can, and it returns a result. If an expression is on the right side of an assignment, the result is what is assigned. Usually that's what you want and expect, but sometimes you need to be able to delay evaluation.

### Quoting and unquoting expressions

The operators to control when expressions are evaluated are `Expr` and `Eval`, which you should think of as the procrastination and eager operators. `Expr` just copies its argument as an expression, rather than evaluating it. `Eval` does the opposite: it evaluates its argument, then takes that result and evaluates it again.

`Expr` and `Eval` can be thought of as quoting and unquoting operators, telling JMP when you mean the expression itself, and when you mean the result of evaluating the expression.

The following examples all assume these two assignments:

```julia
x=1; y=20;
```

If you assign the expression `x+y` to `a`, quoting it as an expression with `Expr`, then whenever `a` is evaluated, it evaluates the expression using the current values of `x` and `y` and returns the result. (Exceptions are the utilities `Show`, `Write`, and `Print`, which don't evaluate expressions for pure name arguments.)
\[ a = \text{expr}(x+y); \]
\[ a; \]
\[ 21 \]

If you want the expression that is stored in the name, rather than the result of evaluating the expression, use the `NameExpr` function, which is detailed in the section “Retrieve a stored expression, not its result,” p. 417:

\[ \text{show(nameExpr(a));} \]
\[ \text{NameExpr(a):x + y} \]

If you assign an extra level of expression-quoting, then when \( a \) is evaluated, it unpacks one layer and the result is the expression \( x+y \).

\[ a = \text{expr(expr}(x+y)); \]
\[ \text{show(a);} \]
\[ a:\text{Expr}(x + y) \]

If you want the value of the expression, then use `Eval` to unpack all layers:

\[ \text{show(eval(a));} \]
\[ \text{Eval(a):}21 \]

You can do this to any level, for example:

\[ a=\text{expr(expr(expr(expr}(x+y))); \]
\[ b=a; \]
\[ \text{Expr(Expr}(x + y)) \]
\[ c=\text{eval(a);} \]
\[ \text{expr}(x+y) \]
\[ d=\text{eval}((\text{eval(a);}) \]
\[ \text{x+y} \]
\[ e=\text{eval(eval(eval(a));} \]
\[ 21 \]

### Store scripts in global variables

The main use of `Expr` is to be able to store scripts (such as macros) in global variables.

\[ \text{dist = expr(Distribution(Column(\text{height}}));} \]

Now when you want to do the script, just mention the symbol:

\[ \text{dist}; \]

You could even put it in a loop to do it many times:

\[ \text{for(i=0, i<10, i=i+1, dist);} \]

You can use `Eval` to evaluate an expression explicitly:

\[ \text{eval(dist);} \]

Note, however, that in column formulas, `eval` only works if it is outermost in the formula. So, for example,

\[ \text{Formula(\text{log(eval(columnname}(i)))} \]

would generate an error. Instead, use

\[ \text{Formula(Eval(Substitute(\text{expr(log(xxx)),expr(xxx), columnname}(i)))))} \]

As another example,
12. Advanced Concepts

Chapter 12

Advanced Concepts

Lists and Expressions

Formula(eval(columnname(i))+10)
generates an error, since eval is actually under the Add function. Instead, use

Formula(Eval(Substitute(expr(xxx+10), expr(xxx), columnname(i))))

Retrieve a stored expression, not its result

What if you wanted the symbolic value of a global (such as the expression
Distribution(Column(height)) stored in dist above), rather than the evaluation of it (the actual launched platform)? The Name Expr function does this. Name Expr retrieves its argument as an expression without evaluating it, but if the argument is a name, it looks up the name's expression and uses that, unevaluated.

Expr returns its argument exactly, whereas Name Expr looks up the expression stored in its argument. Name Expr "unpacks" just one layer to get the expression, but doesn't keep unpacking to get the result.

You would need to use this if, for example, you had an expression stored in a name and you wanted to edit the expression:

\[
\text{popVar} = \text{expr}(\text{Summation}(i=1, \text{NRow}(), (y[i]-\text{Col Mean}(y))^2/\text{NRow}()));
\]

\[
\text{Summation}(i=1, \text{NRow}(), (y[i]-\text{Col Mean}(y))^2/\text{NRow}())
\]

unbiasedPopVar=Substitute(nameexpr(popVar), expr(NRow()),
expr((NRow()-1)));

\[
\text{Summation}(i=1, \text{NRow}()-1, (y[i]-\text{Col Mean}(y))^2/(NRow()-1))
\]

Compare x, Expr(x), NameExpr(x), and Eval(x) after submitting this script:

\[
a=1; b=2; c=3;
\]

\[
x = \text{Expr}(a+b+c);
\]

Table 12.2 Compare Eval, Name Expr, and Expr

<table>
<thead>
<tr>
<th>Command and result</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x; 6</td>
<td>Evaluates x to the expression a+b+c, and then evaluates the expression, returning the result, 6 (unpacks all layers).</td>
</tr>
<tr>
<td>Eval(x); 6</td>
<td>Equivalent to simply calling x. Evaluates x to the expression a+b+c, and then evaluates the expression, returning the result, 6 (unpacks all layers).</td>
</tr>
<tr>
<td>NameExpr(x); a+b+c</td>
<td>Returns the expression that was stored in x, which is a+b+c (unpacks the outside layer).</td>
</tr>
<tr>
<td>Expr(x); x</td>
<td>Returns the expression x (packs one layer).</td>
</tr>
</tbody>
</table>

JSL also supports functions to access and traverse expressions, all of them either a name or a literal expression as an argument. In the following, expressionArg is either a single name or a compound expression to be taken literally.

NArg(expressionArg) finds the number of arguments in expressionArg.

The expressionArg can be a name holding an expression, an expression evaluated to an expression, or a literal expression quoted by Expr(), i.e.

NArg (name) obtains the expression held in name (it is not evaluated) and returns the number of arguments
NArg \((expression)\) evaluates \(expression\) and returns the number of arguments.

NArg \((\text{Expr}(expression))\) returns the number of arguments to literal expression.

For example, if \(a\text{Expr} = \{a+b,c,d,e+f+g\}\):

- \(\text{NArg}(a\text{Expr})\) results in 4.
- \(\text{NArg}(\text{Arg}(a\text{Expr},4))\) results in 3.
- \(\text{NArg}(\text{Expr}\{(1,2,3,4)\})\) results in 4.

Head \((expressionArg)\) returns the head of the expression without any arguments. If the expression is an infix, prefix, or postfix special character operator, then it is returned as the functional equivalent.

The \(expressionArg\) can be a name holding an expression, an expression evaluated to an expression, or a literal expression quoted by \(\text{Expr}()\).

For example, if \(a\text{Expr} = \text{expr}(a+b)\):

- \(r = \text{Head}(a\text{Expr})\) results in \(\text{Add}()\).
- \(r = \text{Head}(\text{Expr}(\text{sqrt}(r)))\) results in \(\text{Sqrt}()\).
- \(r = \text{Head}(\{1,2,3\})\) results in \({}\).

Arg \((expressionArg, indexArg)\) extracts the specified argument of the symbolic expression, resulting in an expression.

For example,

- \(\text{Arg}(a\text{Expr},1)\) yields 12
- \(\text{Arg}(a\text{Expr},2)\) yields 13*\(\text{sqrt}(14-15)\)
- \(\text{Arg}(\text{Expr}(12+13*\text{sqrt}(14-15)),2)\) yields 13*\(\text{sqrt}(14-15)\)

To extract an argument of an argument inside an expression, you can nest Arg commands:

- \(\text{Arg}(\text{Arg}(a\text{Expr},2),1)\) yields the first argument within the second argument of aExpr, or 13.
- \(\text{Arg}(\text{Arg}(a\text{Expr},2),2)\) yields \(\text{Sqrt}(14 - 15)\)
- \(\text{Arg}(\text{Arg}(\text{Arg}(a\text{Expr},2),2),1)\) yields 14 - 15
- \(\text{Arg}(\text{Arg}(\text{Arg}(a\text{Expr},2),2),3)\) yields \(\text{Empty}()\)

Here's how that last one unwraps itself:

1. The inner Arg statement is evaluated.
   - \(\text{Arg}(a\text{Expr},2)\)
   - \(13 * \text{Sqrt}(14 - 15)\)
2. Then the next one is evaluated.
   - \(\text{Arg}(\text{Arg}(a\text{Expr},2),2)\)
12. Advanced Concepts

Chapter 12
Lists and Expressions

// which is equivalent to Arg((13 * Sqrt(14 - 15)), 2)
Sqrt(14 - 15)

Finally, the outer Arg is evaluated.

Arg(Arg(Arg(aExpr, 2), 2), 3)
// which is equivalent to Arg((Sqrt(14 - 15)), 3)
Empty()

There is only one element to the Sqrt expression, so a request for the third argument yields Empty(). To access the two arguments inside the Sqrt expression, try this:

Arg(Arg(Arg(Arg(aExpr, 2), 2), 1), 2);

HeadName(expressionArg) returns the name of the head of the expression as a string. If the expression is an infix, prefix, postfix, or other special character operator, then it is returned as the functional equivalent.

The expressionArg can be a name holding an expression, an expression evaluated to an expression, or a literal expression quoted by Expr().

For example, if aExpr = expr(a+b);

• r = HeadName(aExpr) results in "Add".
• r = HeadName(Expr(sqrt(r))) results in "Sqrt".
• r = HeadName({1,2,3}) results in "List".

In previous versions of JMP, other versions of Arg, Narg, Head, and HeadName were implemented, called ArgExpr, NArgExpr, HeadExpr, and HeadNameExpr, respectively. These did the same thing, but did not evaluate their argument. These forms are now deprecated and will not be documented in future versions.

Making lots of substitutions

Eval Insert is for the situation where you want to make a lot of substitutions, by evaluating expressions inside a character string. [In Perl, this is called interpolation.]

With Eval Insert, you specify characters that delimit the start and end of an expression, and everything in between is evaluated and expanded.

There are two functions, one to return the result, the other to do it in-place.

resultString = EvalInsert(string with embedded expressions, startDelimiter, endDelimiter)

EvalInsertInto(string l-value with embedded expressions, startDelimiter, endDelimiter)

The delimiter is optional. The default start delimiter is "^". The default end delimiter is the start delimiter.

xstring = "def";
r = EvalInsert("abc^xstring^ghi"); // results in "abcdefghi"

// in-place evaluation
r = "abc^xstring^ghi";
EvalInsertInto(r); // r now has "abcdefghi";
// with specified delimiter
r = EvalInsert("abc%xstring%ghi","%");  // results in "abcdefghi";

// with different start and end delimiters
r = EvalInsert("abc[xstring]ghi","[","]");  // results in "abcdefghi";

Evaluate expressions inside lists
Eval List evaluates expressions inside a list and returns a list with the results:

x = { 1+2, 3+4 };  
y = evalList(x);   // result in y is {3,7}

eval List is useful for loading the list of user choices returned by Dialog or Column Dialog.

Evaluate expressions inside expressions
Eval Expr evaluates expressions inside other expressions but returns an expression that includes the results. By comparison, Eval evaluates any expressions inside the expression, and then evaluates the expression. Here's an example where you would need to evaluate the inner expression first:

// assumes a data table with column named X3
x = expr( distribution(column( expr("X"||char(i)) )) );
i = 3;
y = Eval Expr(x);  // returns Distribution(Column("X3"))

However, Eval Expr only unpacks the inside layer then stops, returning the result as an expression. To evaluate further, you need to either call the result in a subsequent step, or else put Eval() around the EvalExpr().

// two-step method
y = Eval Expr(x);
y;

// one-step method
eval(eval expr(x));

See Table 12.4 “Compare all the operators for controlling evaluation,” p. 422, to learn what would happen if you tried to use Eval directly on x without first doing Eval Expr.

Parsing strings into expressions, and vice versa
Parsing is the syntactic scanning of character strings into language expressions. Suppose that you have read in a valid JSL expression into a character string, and now want to evaluate it. The Parse function will return the expression. To evaluate it, use the Eval function.

x = parse("a=1");  // x now has the expression a=1
eval(parse("a=1"));  // a now has the value 1

To go in the reverse, use the Char function, which converts an expression into a character string. Usually the argument to a Char function is an Expr function (or a NameExpr of a global variable), since Char evaluates its argument before deparsing it.

y = char(expr(a=1));  // results in y having the character value "a=1"
z = char(42);
The **Char** function allows arguments for field width and decimal parameter if the argument is a number. The default is 18 for width and 99 for decimal (Best format).

```plaintext
fortytwo = char(42, 5, 2); // results in the character value "42.00"
```

The reverse of **Char** is not quite as simple. To convert a character string into an expression, you use **Parse**, but to convert a character string into a number value, you use **Num**.

```plaintext
parse(y);
num(z);
```

### Table 12.3 Functions to store or evaluate expressions

<table>
<thead>
<tr>
<th>Function</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char</td>
<td>Char(Expr(expression)) Char(name)</td>
<td>Converts an expression into a character string. The expression must be quoted with Expr; otherwise its evaluation is converted to a string. Converts a number into its character representation. Width and decimal are optional arguments to specify formatting; the default is 18 for width and 99 for decimal.</td>
</tr>
<tr>
<td></td>
<td>string = char(number, width, decimal)</td>
<td></td>
</tr>
<tr>
<td>Eval</td>
<td>Eval(x)</td>
<td>Evaluates x, then evaluates the result of x (unquoting).</td>
</tr>
<tr>
<td>Eval Expr</td>
<td>Eval Expr(x)</td>
<td>Returns an expression with all the expressions inside x evaluated.</td>
</tr>
<tr>
<td>Eval List</td>
<td>Eval List(list)</td>
<td>Returns a list of the evaluated expressions inside list.</td>
</tr>
<tr>
<td>Expr</td>
<td>Expr(x)</td>
<td>Returns the argument unevaluated (expression-quoting).</td>
</tr>
<tr>
<td>NameExpr</td>
<td>NameExpr(x)</td>
<td>Returns the unevaluated expression of x rather than the evaluation of x. NameExpr is like Expr except that if x is a name, NameExpr returns the unevaluated expression stored in the name rather than the unevaluated name x.</td>
</tr>
<tr>
<td>Num</td>
<td>Num(&quot;string&quot;)</td>
<td>Converts a character string into a number.</td>
</tr>
<tr>
<td>Parse</td>
<td>Parse(&quot;string&quot;)</td>
<td>Converts a character string into a JSL expression.</td>
</tr>
</tbody>
</table>

### Summary

This table compares various ways you could use the evaluation-control operators with x. Assume that x and i have been assigned as before:

```plaintext
x = expr( distribution(column( expr("X"||char(i)) )) );
i = 3;
```
### Table 12.4 Compare all the operators for controlling evaluation

<table>
<thead>
<tr>
<th>Commands and results</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x; // or</code></td>
<td>Eval(x) and simply calling x are equivalent.</td>
</tr>
<tr>
<td>Eval(x);</td>
<td>Evaluates outer expression and inner expression at the same time, resulting in expression Distribution{&quot;X&quot;</td>
</tr>
<tr>
<td></td>
<td>and simply calling x are equivalent.</td>
</tr>
<tr>
<td></td>
<td>Evaluates outer expression and inner expression at the same time, resulting in expression Distribution{&quot;X&quot;</td>
</tr>
<tr>
<td>Expr(x);</td>
<td>Returns the expression x (packs an additional layer).</td>
</tr>
<tr>
<td></td>
<td>Returns the expression stored in x exactly as is: Distribution(Column(Expr(&quot;X&quot;</td>
</tr>
<tr>
<td>Name Expr(x);</td>
<td>Evaluates the inner expression but leaves the outer expression unevaluated, so that y is Distribution(Column(&quot;X3&quot;)).</td>
</tr>
<tr>
<td></td>
<td>Evaluates Distribution(Column(&quot;X3&quot;)) to launch the platform.</td>
</tr>
<tr>
<td>y = Eval Expr(x);</td>
<td>Evaluates the inner expression but leaves the outer expression unevaluated, so that y is Distribution(Column(&quot;X3&quot;)).</td>
</tr>
<tr>
<td></td>
<td>Evaluates Distribution(Column(&quot;X3&quot;)) to launch the platform.</td>
</tr>
<tr>
<td>z = Char(nameexpr(x));</td>
<td>Quotes the entire expression as a text string, adding &quot; escape characters as needed. Note that Char(x) would first attempt to evaluate x, producing an error and ultimately returning a quoted missing value: &quot;.&quot;</td>
</tr>
<tr>
<td>Parse(z);</td>
<td>Unquotes the text string and returns an expression.</td>
</tr>
<tr>
<td>a = Parse(Char(NameExpr(x)));</td>
<td>Evaluation control taken to its logical extreme.</td>
</tr>
<tr>
<td>Eval(EvalExpr(a));</td>
<td>Note that you must break this into at least two steps as shown. Combining it into one giant step produces different results because the Eval Expr layer causes the Parse layer to be copied literally, not executed.</td>
</tr>
<tr>
<td></td>
<td>Eval(EvalExpr(</td>
</tr>
<tr>
<td></td>
<td>Parse(</td>
</tr>
<tr>
<td></td>
<td>Char(</td>
</tr>
<tr>
<td></td>
<td>NameExpr(x)))));</td>
</tr>
<tr>
<td></td>
<td>Distribution(Column(Expr(&quot;X&quot;</td>
</tr>
</tbody>
</table>
Macros

Stored expressions can serve as a macro feature. You can store a generalized action as an expression in a global, and then call that global wherever you need that action to be performed. This example has four macros as the arguments to If:

```r
lastStdzdThickness = expr(
  (thickness[nrow()] - col mean(thickness)) / col std dev(thickness));
continue = expr(...<script to read in more data>...);
log = expr(print("In control at ", char(long date(today()))));
break = expr(...<script to shut down process>...); limitvalue = 1;

if(lastStdzdThickness < limitvalue, log; continue, break);
```

Storing the expression (the script itself, not its evaluation at the moment) with `Expr` delays its evaluation until the global is actually called, so that any variables, data points, or expressions included in that expression are evaluated on the fly when the expression is evaluated. The later section "Stored expressions," p. 415, examines the rules for storing expressions and later evaluating them in greater detail.

Manipulating lists

The following operators manipulate lists. They can also be used to manipulate expressions, as shown in the next section, “Manipulating expressions,” p. 425. A summary of commands with explanations is in Table 12.5 “Functions for manipulating lists or expressions,” p. 427.

Most of the function have two variants, one that produces a new value, and one that works in-place directly on its arguments. Here are some example pairs:

```r
A = Remove(A,3); // delete the third item in the list A, storing result in A
Remove From(A,3); // delete the third item in the list A, in place

onetwo = Insert({1},2); // onetwo is {1,2}
InsertInto(A,{1,2},4); // puts 1,2 before the current 4th item
```

**Note:** If position is omitted in the `Insert Into` command, items are placed at the end of the list.

```r
a = Shift({1,2,3,4},1); // stores the list {2,3,4,1} in a
Shift Into(a,-1); // a is now {1,2,3,4}

b = Reverse(a); // b is now {4,3,2,1}
Reverse Into(a); // a is now {4,3,2,1}, too

s = Sort List({1,4,2,5,-7.2,pi(),-11,cat, apple, cake});
    // s is now sorted list
Sort List Into(c); // c is now {-2,2,5,pi(),pie}
```

In-place operators

In-place operators are those that operate on lists or expressions directly. They have `From` or `Into` in their names, e.g. `Remove From` and `Insert Into`. They do not return a result; you have to show the list to

```r
```
see the result. The first argument for an in-place operator must be an L-value. An L-value is an entity such as a global variable whose value can be set.

```cpp
myList={a, b, c, d};
Insert Into(myList,2,3);
show(myList);
  myList:{a, b, 2, c, d}
```

These examples show how to use `Insert Into` and `Remove From` with nested lists:

```cpp
a = {{1, 2, 3}, {"A", "B", "C"}};
Show(a);
  a:{{1, 2, 3}, {"A", "B", "C"}}
Insert Into(a[1], 99, 1);
Show(a);
  a:{{99, 1, 2, 3}, {"A", "B", "C"}}
Remove From(a[1], 1);
Show(a);
  a:{{1, 2, 3}, {"A", "B", "C"}}
```

### Not in-place operators

For the not-in-place operators, you must either state the list directly or else quote a name that evaluates to a list. Such operators do not have `From` or `Into` in their names. They return manipulated lists or expressions without changing the original list or expression given in the first argument.

```cpp
myNewList=Insert({a, b, c, d}, 2, 3);
{a, b, 2, c, d}
oldList={a, b, c, d};
newList=Insert(oldList, 2, 3);
{a, b, 2, c, d}
```

### Substituting

`Substitute` and `Substitute Into` merit further discussion. Both find all matches to a pattern in a list (or expression) and replace them with another expression. Each pattern must be a name. The arguments are evaluated before they are applied, so most of the time you must quote them with an `Expr` function.

```cpp
Substitute([a,b,c], expr(a), 23); // produces [23,b,c]
Substitute(expr(sin(x)), expr(x), expr(y)); // produces sin(y)
```

To delay evaluating an argument, use `NameExpr` instead of `Expr`:

```cpp
a={quick,brown,fox,jumped,over,lazy,dogs};
b=Substitute(a, expr(dogs), expr(cat));
canine=expr(dogs);equine=expr(horse);
c=Substitute(a, nameexpr(canine), nameexpr(equine)); show(a,b,c);
  a:{quick,brown,fox,jumped,over,lazy,dogs}
  b:{quick,brown,fox,jumped,over,lazy,cat}
  c:{quick,brown,fox,jumped,over,lazy,horse}
```

`Substitute Into` does the same work, in place:
Substitute Into(a, expr(dogs), expr(horse));

You can list multiple pattern and replacement arguments to do more than one replacement in a single step:

d=Substitute(a, 
    nameexpr(quick), nameexpr(fast), 
    nameexpr(brown), nameexpr(black), 
    nameexpr(fox), nameexpr(wolf) 
); 
{fast, black, wolf, jumped, over, lazy, dogs}

Note that substitutions are done repeatedly over multiple instances of the expression pattern. For example:

Substitute(expr(a+a), expr(a), expr(aaa));

results in:

aaa + aaa

### Manipulating expressions

The operators for manipulating lists can also operate on most expressions. Be sure to quote the expression with Expr. For example:

Remove(Expr(A+B+C+D),2); // results in the expression A+C+D
b=Substitute(expr(log(2)^2/2), 2, 3); // results in the expression Log(3)^3/3

As with lists, remember that the first argument for in-place operators must be an L-value. An L-value is an entity such as a global variable whose value can be set. In-place operators are those that operate on lists or expressions directly. They have From or Into in their names, e.g. Remove From and Insert Into. They do not return a result; you have to show the expression to see the result.

\[
\text{polynomial}=\text{expr}(a*x^2 + b*x + c) ; \\
\text{insertinto}(\text{polynomial}, \text{expr}(d*x^3),1) ; \\
\text{show}(\text{polynomial}) ; \\
\text{polynomial}: d * x ^ 3 + a * x ^ 2 + b * x + c
\]

For the not-in-place operators, you must either state the expression directly or else quote a name that evaluates to an expression using NameExpr. Such operators do not have From or Into in their names. They return manipulated lists or expressions without changing the original list or expression given in the first argument.

\[
\text{cubic}=\text{insert}(\text{expr}(a*x^2 + b*x + c), \text{expr}(d*x^3),1) ; \\
d * x ^ 3 + a * x ^ 2 + b * x + c
\]

\[
\text{quadratic}=\text{expr}(a*x^2 + b*x + c) ; \\
\text{cubic}=\text{insert}(\text{nameexpr} (\text{quadratic}), \text{expr}(d*x^3),1) ; \\
d * x ^ 3 + a * x ^ 2 + b * x + c
\]

### Substituting

Substituting is extremely powerful; please review the earlier discussion “Substituting,” p. 424. Here are a few notes regarding substituting for expressions.
Substitute(pattern, name, replacement) substitutes for names in expressions.

NameExpr looks through the name but copies instead of evaluates:

```plaintext
a = expr(distribution(column(x), normal quantile plot)); show(NameExpr(a));
NameExpr(a):Distribution(Column(x), normal quantile plot)
```

Substitute evaluates all its arguments, so they must be quoted correctly:

```plaintext
b = substitute(NameExpr(a), expr(x), expr(weight)); show(NameExpr(b));
NameExpr(b):Distribution(Column(weight), normal quantile plot)
```

SubstituteInto needs an L-value, so the first argument isn't quoted:

```plaintext
SubstituteInto(a, expr(x), expr(weight)); show(NameExpr(a));
NameExpr(a):Distribution(Column(weight), normal quantile plot)
```

Substitute is useful for changing parts of expressions, such as in the following example that tests the Is functions:

```plaintext
data = {1, {1,2,3}, [1 2 3], "abc", x, x(y)};
ops = {is number, is list, is matrix, is string, is name, is expr};
m=J(n items(data),n items(ops),0);
test = expr(m[r,c] = _op(data[r]));
for (r=1,r<=n items(data),r++,
  for (c=1,c<=n items(ops),c++,
    eval(substitute(nameexpr(test), expr(_op),ops[c]))));
show(m);
```

```plaintext
m:
[1 0 0 0 0 0,
 0 1 0 0 0 1,
 0 0 1 0 0 0,
 0 0 0 1 0 0,
 0 0 0 0 1 1,
 0 0 0 0 0 1]
```

You can use SubstituteInto to have JMP solve quadratic equations. The following example solves $4x^2 - 9 = 0$:

```plaintext
/* FIND THE ROOTS FOR THE EQUATION: */
/*
 * a x^2 + b x + c = 0
 */
// The quadratic formula is x = (-b ± sqrt(b^2 - 4ac))/2a.
// Use a list to store both the + and - results of the ± operation
x={expr((-b + sqrt(b^2 - 4*a*c))/(2*a)),
   expr((-b - sqrt(b^2 - 4*a*c))/(2*a))};
// Next, plug in the coefficients:
substitute into(x,expr(a),4, expr(b),0, expr(c),-9);
show(x);             //see the result of substitution
show(eval(expr(x))); //see the solution
```

```plaintext
x:{Expr((-0+Sqrt(0^2-4*4*-9))/(2*4)),Expr((-0-Sqrt(0^2-4*4*-9))/(2*4))}
EvalExpr(x):{1.5,-1.5}
```

The operators for manipulating lists and expressions are discussed in the previous section, “Manipulating lists,” p. 423, and summarized in Table 12.5.
## Table 12.5 Functions for manipulating lists or expressions

<table>
<thead>
<tr>
<th>Function</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove</td>
<td>( x = \text{Remove}(\text{list}</td>
<td>\text{expr}) ) ( x = \text{Remove}(\text{list}</td>
</tr>
<tr>
<td>Remove From</td>
<td>( x = \text{Remove From}(\text{list}</td>
<td>\text{expr}, \text{position}) ) ( x = \text{Remove From}(\text{list}</td>
</tr>
<tr>
<td>Insert</td>
<td>( x = \text{Insert}(\text{list}</td>
<td>\text{expr}, \text{item}, \text{position}) ) ( x = \text{Insert}(\text{list}</td>
</tr>
<tr>
<td>Insert Into</td>
<td>( x = \text{Insert Into}(\text{list}</td>
<td>\text{expr}, \text{item}, \text{position}) ) ( x = \text{Insert Into}(\text{list}</td>
</tr>
<tr>
<td>Shift</td>
<td>( x = \text{Shift}(\text{list}</td>
<td>\text{expr}) ) ( x = \text{Shift}(\text{list}</td>
</tr>
<tr>
<td>Shift Into</td>
<td>( x = \text{Shift Into}(\text{list}</td>
<td>\text{expr}) ) ( x = \text{Shift Into}(\text{list}</td>
</tr>
<tr>
<td>Reverse</td>
<td>( x = \text{Reverse}(\text{list}</td>
<td>\text{expr}) ) ( x = \text{Reverse Into}(\text{list}</td>
</tr>
<tr>
<td>Sort List</td>
<td>( x = \text{Sort List}(\text{list}</td>
<td>\text{expr}) )</td>
</tr>
<tr>
<td>Sort List Into</td>
<td>( x = \text{Sort List Into}(\text{list}</td>
<td>\text{expr}) )</td>
</tr>
<tr>
<td>Sort Ascending</td>
<td>( x = \text{Sort Ascending}(\text{list}</td>
<td>\text{matrix}) )</td>
</tr>
<tr>
<td>Sort Descending</td>
<td>( x = \text{Sort Descending}(\text{list}</td>
<td>\text{matrix}) )</td>
</tr>
</tbody>
</table>
Projects

JMP projects are fully scriptable. The following script creates a new project, adds groups and files to it, and:

```plaintext
expj = New Project( "My Project" );
exg1 = expj << Add Group( "Data" );
exg2 = expj << Add Group( "Reports" );
exdt = Open( "$SAMPLE_DATA/Big Class.jmp" );
exg1 << Add Window( exdt );
exrp = Bivariate( X( height ), Y( weight ) );
exg2 << Add Window( exrp );
Close( exdt, NoSave );
exj << getname();
```

To open a project that already has been saved, use the Open Project function. For example,

```plaintext
prj = Open Project("filepath");
```
Jump can also handle binary (large) objects, commonly called BLOBs. The functions below convert between hexadecimal values, numbers, characters, and BLOBs. Some of the functions are covered in more detail following Table 12.6.

These functions are listed in the “Hex and BLOB Functions,” p. 467 in the Syntax Reference for Summary of operators, functions, and messages.

### Table 12.6 Hexadecimal and BLOB Functions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex('text')</td>
<td>Returns the hexadecimal codes for the characters in text, number, or blob.</td>
</tr>
<tr>
<td>Hex(num)</td>
<td></td>
</tr>
<tr>
<td>Hex(blob)</td>
<td></td>
</tr>
<tr>
<td>Hex To Blob('hexstring')</td>
<td>Returns a BLOB representation of the hexadecimal code supplied as a quoted string.</td>
</tr>
<tr>
<td>Hex To Char('hexstring', encoding)</td>
<td>Returns a character string that corresponds to the hexadecimal code supplied as a quoted string (see “Character Functions,” p. 67 in the “JSL Operators” chapter). The default encoding supported for the hex code is UTF-8. You can also specify one of these encodings: utf-8, utf-16le, utf-16be, us-ascii, iso-8859-1, and ascii-hex.</td>
</tr>
<tr>
<td>Hex to Number</td>
<td>Returns the number that corresponds to the hexadecimal code supplied as a quoted string.</td>
</tr>
<tr>
<td>Char To Blob(string)</td>
<td>Converts a string of characters into a binary (blob). The default encoding supported for the hex code is UTF-8. You can also specify one of these encodings: utf-8, utf-16le, utf-16be, us-ascii, iso-8859-1, and ascii-hex.</td>
</tr>
<tr>
<td>Char To Blob(string, encoding)</td>
<td></td>
</tr>
<tr>
<td>Blob To Char(blob)</td>
<td>Converts binary data to a Unicode string. The default encoding supported for the hex code is UTF-8. You can also specify one of these encodings: utf-8, utf-16le, utf-16be, us-ascii, iso-8859-1, and ascii-hex.</td>
</tr>
<tr>
<td>Blob To Char(blob, encoding)</td>
<td></td>
</tr>
<tr>
<td>Blob Peek(blob, offset, length)</td>
<td>Returns a new BLOB that is a subset of the given BLOB that is length bytes long and begins at the offset. Note that the offset is 0-based.</td>
</tr>
</tbody>
</table>

**Hex** *(string)* returns the hexadecimal codes for each character in the argument. For example,

Hex("Abc")

returns

"416263"

since 41, 62, and 63 are the hexadecimal codes (in Ascii) for “A”, “b”, and “c”.

**Hex to Char** *(string)* converts hexadecimal to characters. The resulting character string may not be valid display characters. All the characters must be in pairs, in the ranges 0-9, A-Z, and a-z. Blanks and commas are allowed, and skipped. As an example,

Hex To Char ("4142")
returns
"AB"
since 41 and 42 are the hexadecimal equivalents of “A” and “B”.

Hex and Hex To Char are inverses of each other, so

Hex To Char ( Hex("Abc") )
returns
"Abc"

Hex To Blob(string) takes a strings of hexadecimal codes and converts it to a binary object.

a = Hex To Blob("6A6B6C"); Show(a);
a:Char To Blob("jk1")

Blob Peek(blob,offset,length) extracts bytes as defined by the parameters from a blob.

b = Blob Peek(a,1,2); Show(b);
b:Char To Blob("kl")
b = Blob Peek(a,0,2); Show(b);
b:Char To Blob("jk")
b = Blob Peek(a,2); Show(b);
b:Char To Blob("l")

Hex(blob) converts a blob into hexadecimal.

c = Hex(a); Show(c);
c:"6A6B6C"
d = Hex To Char(c); Show(d);
d:"jk1"

Concat(blob1,blob2) or blob1 || blob2 concatenates two blobs.

e = Hex To Blob("6D6E6F"); Show(e);
f = a||e; show(f);
e:Char To Blob("mno")
f:Char To Blob("jkmlmno")

length(blob) returns the number of bytes in a blob.

g = length(f); show(g);
g:6

Note: When blobs are listed in the log, they are shown with the constructor function
Char To Blob("...").

Any hex code outside the ascii range (space to }, or hex 20 - 7D) is encoded as the three-character sequence [-][hexdigit][hexdigit]. For example,

h = Hex To Blob("19207D7E"); Show(h);
i = Hex(h); Show(i);
h:Char To Blob("~19 ~7D7E")
i:"19207D7E"

Char To Blob(string) creates a blob from a string, converting -hex codes.

Blob To Char(blob) creates a string with -hex codes to indicate non-visible and non-ascii codes.
Web

Web takes a string as its argument, where the string is a URL. This URL is opened with the default web browser of your system. The URL can be stored as a string variable or quoted directly. For example,

```javascript
url = "www.jmp.com";
Web(url);
```
does the same thing as

```javascript
Web("www.jmp.com");
```

Note: On the Windows version of JMP, you have the option of opening the URL in a JMP window instead of your default web browser. To open the url in a JMP window, add JMP Window to the web command:

```javascript
Web("www.jmp.com", JMP Window);
```

Additional Numeric Operators

JSL also offers several categories of operations that don't make much sense in the context of the formula editor: matrix operations and numeric derivatives of functions. Algebraic derivatives are available, as detailed below.

Matrices

The basic arithmetic operators can also be used with matrix arguments for matrixwise addition, subtraction, etc. Matrices also have a few special operators for elementwise multiplication and division, concatenation, and indexing. See the chapter “Matrices,” p. 349, for details.

Derivatives

JSL has three internal operators (not all available in the calculator) for taking derivatives.

Derivative takes the first derivative of an expression with respect to names you specify in the second argument. A single name may be entered as this second argument; or multiple values can be specified in a list, i.e. surrounded by braces.

Derivative is also available as an editing command inside the formula editor (calculator), located on the drop-down list in the top center of the formula editor (above the keypad). To use it, highlight a single variable in the expression (to designate which variable the derivative should be taken with respect to), then select the Derivative command from the menu. The whole formula is replaced by its derivative with respect to the highlighted name.

In scripts, the easiest way to use the function is with a single name. In this example, we first show the mathematical notation and then the JSL equivalent.
For \( f(x) = x^3 \), the first derivative is \( f'(x) \) or \( \frac{d}{dx}x^3 = 3x^2 \).

```javascript
result = derivative(x^3, x); show(result);
result: 3 * x ^ 2
```

If you want an efficient expression to take the derivative with respect to several variables, then the variables are specified in a list, and the result is a list containing a threaded version of the original expression, followed by expressions for the derivatives. The expression is threaded by inserting assignments to temporary variables of expressions that will be needed in several places for the derivatives.

Here is an example involving an expression involving three variables. Listing all three variables returns the first derivatives with respect to each. The result is a list with the original expression and then the derivatives in the order requested, but note here that JMP creates a temporary variable T\#1 for storing the subexpression x\^2, and then uses that subexpression subsequently to save calculations.

```javascript
result2 = derivative(3*y*x^2+z^3, {x,y,z}); show(result2);
result2: 3 * y * (T#1 = x ^ 2) + z ^ 3, 6 * x * y, 3 * T#1, 3 * z ^ 2
```

To take second derivatives, specify the variable as a third argument. Both the second and third arguments must be lists. JMP returns a list with the original expression, the first derivative(s), and then the second derivative(s) in the order requested.

```javascript
second = derivative(3*y*x^2, {x}, {x}, {x}); show(second);
second: 3 * y * x ^ 2, 6 * y
second = derivative(3*y*x^2, {y}, {y}); show(second);
second: 3 * y * (T#1 = x ^ 2), 3 * T#1, 0
second = derivative(3*y*x^2, {y}, {x}); show(second);
second: 3 * y * (T#2 = x ^ 2), 3 * T#2, 6 * x
```

**NumDeriv** takes the first numeric derivative of an operator or function with respect to the value of the first argument by calculating the function at that value and at that value plus a small delta (\( \Delta \)) and dividing the difference by the delta. **NumDeriv2** computes the second numeric derivative in a similar fashion. These are used internally for nonlinear modeling but aren’t frequently useful in JSL. Note that these functions do not differentiate using a variable, but only with respect to arguments to a function.

In order to differentiate with respect to \( x \), you have to make \( x \) one of the immediate arguments, not a symbol buried deep into the expression.

Suppose to differentiate \( y = 3x^2 \) at the value of \( x = 3 \). The incorrect way would be to submit

```javascript
x=3;
num=NumDeriv(3*x^2);
```

The correct way is to make \( x \) an argument in the function.

```javascript
x=3;
f=function({x}, 3*x^2);
num=NumDeriv(f(x), 1);
```

Consider both the mathematical notation and the JSL equivalent for another example:

For \( f(x) = x^2 \), it calculates \( \frac{d}{dx}x^2 = \frac{(x+\Delta)^2-x^2}{\Delta} \). At \( x_0 = 3 \), \( \frac{d}{dx}x^2 = 6.00001 \).

```javascript
x=3; y=numDeriv(x^2);  // or equivalently: y = numDeriv(3^2);
```
And here are a few more examples:

\[
\begin{align*}
x &= \text{numderiv}(\sqrt{7}); \\
y &= \text{numDeriv}([\text{Normal Distribution}(1)]); \\
z &= \text{Num deriv2}([\text{Normal distribution}(1)]);
\end{align*}
\]

### Table 12.7 Derivative functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derivative</td>
<td>Derivative(expr, {name, ...})</td>
<td>Returns the derivative of the ( expr ) with respect to ( name ). Note that the second argument can be specified in a list with braces { } or simply as a variable if there is only one. Give two lists of names to take second derivatives.</td>
</tr>
<tr>
<td>NumDeriv</td>
<td>NumDeriv(expr)</td>
<td>Returns the first numeric derivative of the ( expr ) with respect to the first argument in the expression.</td>
</tr>
<tr>
<td>NumDeriv2</td>
<td>NumDeriv2(expr)</td>
<td>Returns the second numeric derivative of the ( expr ) with respect to the first argument in the expression.</td>
</tr>
</tbody>
</table>

### Algebraic Manipulations

JSL provides a way of algebraically unwinding an expression—essentially, solving for a variable. It is accomplished through the \textit{Invert Expr()} function.

\[
\text{Invert Expr}([expression, name, y])
\]

where \( expression \) is the expression to be inverted, or the name of a global containing the expression, \( name \) is the name inside expression to unwind the expression around, and \( y \) is what the expression was originally equal to.

For example:

\[
\text{Invert Expr}(\sqrt{\log(x)}, x, y)
\]

is wound around the name \( x \) (which should only appear in the expression once), and results in

\[
\exp(y^2)
\]

It is performed exactly as you would when doing the algebra by hand

\[
\begin{align*}
y &= \sqrt{\log(x)} \\
y^2 &= \log(x) \\
\exp(y^2) &= x
\end{align*}
\]

\textit{Invert Expr} supports most basic operations that are invertible, and makes assumptions as necessary, such as assuming you are interested only in the positive roots, and that the trigonometric functions are in invertible areas so that the inverse function are legal.

\( F \), Beta, Chi-square, \( t \), Gamma, and Weibull distributions are supported for the first arguments in their Distribution and Quantile functions. If it encounters an expression it cannot convert, \textit{Invert Expr()} returns \textit{Empty}().

JSL provides a \textit{Simplify Expr} command that takes a messy, complex formula and tries to simplify it using various algebraic rules. To use it, submit
result = Simplify Expr(expr(expression));

or

result = Simplify Expr(nameExpr(global));

For example,

Simplify Expr (expr(2*3*a+b*(a+3-c)-a*b));

results in

6*a + 3*b + -1*b*c

Simplify Expr also unwinds nested If expressions. For example:

r = simplifyExpr(
    expr(If(cond1,result1,if(cond2,result2,if(cond3,result3,resultElse)))));

results in

If(cond1, result1, cond2, result2, cond3, result3, resultElse);

Maximize and Minimize

The Maximize and Minimize functions find the factor values that optimize an expression. The expression is assumed to be a continuous function of the factor values.

The form of the call is

result = Maximize(objectiveExpression, {list of factor names}, <<option(value))
result = Minimize(objectiveExpression, {list of factor names}, <<option(value))

objectiveExpression is the expression whose value is to be optimized, and can either be the expression itself, or the name of a global containing a stored expression.

{list of factor names} is an expression yielding a list of names involved in objectiveExpression.

The name can be followed by limits that bound the permitted values, for example name(lowerBound,upperBound).

If you want to limit the values on one side, make the other side a missing value, e.g.

{beta} //unconstrained
{beta (0,1)} //constrained between 0 and 1
{beta (.,1)} // upper limit of 1
{beta (0,.)} or {beta (0)} // lower limit of 0

Factor values can be either numbers or matrices.

Options available, shown with their default value, include:

<< tolerance(.00000001) //convergence criterion
<< maxIter( 250) // maximum number of iterations
<< limits() //
<<Method(NR|BFGS) // Specify either the Newton-Raphson method or the Quasi-Newton method with BFGS update.

Initial values are assumed to be already supplied the factor values prior to calling the function.
These functions are not expected to find global optima for functions that have multiple local optima; it is only useful for taking an initial value and moving to an optimum, local or global.

The return value is currently the value of the objective function, if the optimization was successful, or Empty() if not.

**Least-Squares Example**

The following example uses Minimize to find the least-squares estimates of this exponential model, with data taken from the US Population.jmp sample data table, found in the Nonlinear Examples sample data folder.

\[
\begin{align*}
b0 &= 3.9; \\
b1 &= .022; \\
0 &= 3.9; \\
b1 &= .022; \\
\text{sseExpr} &= \text{Expr} ( \text{Sum} ( \text{yy} - (b0 \times \text{Exp}(b1 \times (xx - 1790))))^2); \\
\text{sse} &= \text{Minimize} (\text{sseExpr}, \{b0, b1\}, <<\text{tolerance}(.00001))); \\
\text{Show}(b0, b1, sse); 
\end{align*}
\]
Appendix A

JSL Syntax Reference

Summary of operators, functions, and messages

This appendix gives abbreviated descriptions for all of JMP’s functions, operators, and general object messages. For platform messages, see the Scripting Platforms chapter. The tables are arranged in chapter order, and items within each table are arranged in alphabetical order, rather than topical order as shown in the chapters.

For an alphabetical index to all JSL functions, see the Index to the Syntax Reference.
Contents

Functions, Variables, and Other Basic Tools ................................................. 439
  Constants ........................................................................... 442
  Arithmetic and Assignment Operators ........................................... 442
  Numeric Functions ................................................................. 448
  Transcendental Functions ......................................................... 449
  Trigonometric Functions ......................................................... 452
  Random Functions ................................................................ 455
  Character/String Functions ...................................................... 458
  Comparison/Logical Functions .................................................. 469
  Date/Time Functions ............................................................... 475
  General Utility Functions ......................................................... 480
  Loop Controls and Conditional Statements ................................... 494
  Lists .................................................................................... 496
  Matrices ................................................................................ 497
  Associative Arrays .................................................................. 510
Data Tables, Rows, and Columns ......................................................... 512
Platforms .................................................................................... 528
Windows, Reports, and Dialogs ........................................................ 531
  Display Boxes ..................................................................... 531
  Modal Dialogs ...................................................................... 548
  Graphs ................................................................................... 549
  3d Scenes ............................................................................. 559
Statistical and Probability Functions .................................................... 559
  Probability Functions ............................................................ 560
  Discrete Probability Functions ................................................ 567
  Specialty Probability Functions ............................................... 570
  Log Density Functions ............................................................ 570
  Statistical Functions ............................................................... 572
  Column Statistics .................................................................. 576
  Additional Statistical Functions ................................................. 579
Advanced Concepts ........................................................................ 582
SAS Integration ............................................................................. 598
Tools for Production Settings ......................................................... 630
Functions, Variables, and Other Basic Tools

This section gives abbreviated descriptions for JMP’s basic functions, operators, and general object messages.

Comments

// comment

Function:
Comments to end of line.

Notes:
Everything after the // is ignored when running the script.

/* comment */

Function:
A comment that can appear in the middle of a line of script.

Notes:
Anything between the beginning tag /* and the end tag */ is ignored when running the script. This comment style can be used almost anywhere, even inside lists of arguments. If you place a comment inside a double-quoted string, the comment is treated merely as part of the string and not a comment. You cannot place comments in the middle of operators.

Examples:
+//*comment*/=
:/*comment*/name
are invalid and will produce errors. The first comment interrupts += and the second interrupts :name.
sums = {(a+b /*comment*/), /*comment*/ (c^2)}
is valid JSL; the comments are both ignored.

//!

Function:
If placed on the first line of a script, this comment line causes the script to be run when opened in JMP without opening into the script editor window.
/*debug step*/
/*debug run*/

Function:
If placed on the first line of a script, the script is opened into the debugger when it is run.

Notes:
All letters must be lower case. There must be one space between debug and step or run, and there
must be no other spaces present. Only one of these lines may be used, and it must be the first line of
the script; a first line that is blank followed by this comment negates the debug command.

---

Glue

Glue(expr1, expr2, ...)
;

Function:
Evaluates each argument in turn.

Returns:
The result of the last argument evaluated.

Arguments:
One or more valid JSL expressions.

---

Variables

Global and Local

Clear Globals(<name>, <name>, ...)

Function:
Clears the values of all global variables if no arguments are supplied. Clears the values of only the
named global variables if arguments are supplied.

Returns:
Null.

Arguments:
name: Optional: any global variable name(s).

Local({name=value, ...}, script)

Function:
Resolves names to local expressions.
Lock Globals(name1, name2, ...)
Function:
Locks one or more global variables to prevent it or them from being changed.

Show Globals()
Function:
Shows all defined global symbols and their values.

Unlock Globals(name1, name2, ...)
Function:
Releases a lock from a Lock Globals() command.

Watch(global, global, ...)
Function:
Displays globals’ names and values in a window. If “all” is provided as the argument, all globals are placed into the window.

Note:
New globals will not be added to the window list.

Scoping Operators

As Column(name)
As Column(dt, name)
:name
dt:name
Function:
This scoping operator forces name to be evaluated as a data table column in the current data table (or the table given by the optional data table reference argument, dt) rather than as a global variable.

Arguments:
name Variable name.
dt The data table reference

Note:
:name refers to a column name in the current data table. You can also specify which data table to refer to by use dt:name.
As Global(name)
::name

Function:
This scoping operator forces name to be evaluated as a global variable rather than as a data table column.

Arguments:
name: Variable name.

Constants

e()

Function:
Returns the constant $e$, which is 2.7182818284590451...

Pi( )

Function:
Returns the constant $\pi$, which is 3.1415926535897931...

Arithmetic and Assignment Operators

Add(a, b, ...)

Function:
Adds the values of the listed arguments. No arguments are changed.

Returns:
The sum.

Arguments:
For Add(), a comma-separated list of variables, numbers, or matrices.
For a+b, any number of variables, numbers, or matrices.

Notes:
Any number of arguments is permitted. If no argument is specified, Add() returns 0.

See Also:
The Matrix Algebra chapter in the JMP Scripting Guide.
Add To(a, b)

a+=b

Function:
Adds a and b and places the sum into a.

Returns:
The sum.

Arguments:
- a Must be a variable.
- b Can be a variable, number, or matrix.

Notes:
The first argument must be a variable, because its value must be able to accept a value change. A number as the first argument will produce an error.
For Add To(): Only two arguments are permitted. If one or no argument is specified, Add To() returns a missing value. Any arguments after the first two are ignored.
For a+=b: More than two arguments may be strung together. JMP evaluates pairs from right to left, and each sum is placed in the left-hand variable. All arguments except the last must be a variable.

Example:
a+=b+=c
JMP adds b and c and places the sum into b. Then JMP adds a and b and places the sum into a.

See Also:
The Matrix Algebra chapter in the JMP Scripting Guide.

Assign(a, b)

a=b

Function:
Places the value of b into a.

Returns:
The new value of a.

Arguments:
- a Must be a variable.
- b Can be a variable, number, or matrix.

Notes:
a must be a variable, because it must be able to accept a value change. A number as the first argument will produce an error. If b is some sort of expression, it's evaluated first and the result is placed into a.
Divide(a, b)
Divide(x)
a/b

Function:
Divides $a$ by $b$. If only one argument is given (divide(x)), divides 1 by $x$.

Returns:
The quotient of $a/b$; or the reciprocal of $x$ ($1/x$) if only one argument is provided.

Arguments:
a, b, x Can be a variable, number, or matrix.

Notes:
If both arguments are matrices, it does matrix division.

See Also:
The Matrix Algebra chapter in the *JMP Scripting Guide*.

Divide To(a, b)
a/=b

Function:
Divides $a$ by $b$ and places the result into $a$.

Returns:
The quotient.

Arguments:
a Must be a variable.
b Can be a variable, number, or matrix.

See Also:
The Matrix Algebra chapter in the *JMP Scripting Guide*.

Minus(a)
-a

Function:
Reverses the sign of $a$.

Returns:
- $a$ if $a$ is positive ($a=3; -a=-3; \text{Minus}(a)=-3$).
- $a$ if $a$ is negative ($a=-3; -a=3; \text{Minus}(a)=3$).
- 0 if $a$ is 0 ($a=0; -a=0; \text{Minus}(a)=0$).
- Missing if $a$ is missing.

Argument:
a Can be variable or a number. A variable must contain a number or a matrix.
Multiply(a, b, ...)  
a*b*...

Function:
   Multiplies all values. No arguments are changed.

Returns:
   The product.

Arguments:
   Any number of variables, numbers, or matrices.

Notes:
   Any number of arguments is permitted. If no argument is specified, Multiply() returns 1.

See Also:
   The Matrix Algebra chapter in the JMP Scripting Guide.

Multiply To(a, b)  
a*=b

Function:
   Multiplies a and b and places the product into a.

Returns:
   The product.

Arguments:
   a  Must be a variable.
   b  Can be a variable, number, or matrix.

Notes:
   The first argument must be a variable, because its value must be able to accept a value change. A number as the first argument will produce an error.
   For Multiply To(): Only two arguments are permitted. If one or no argument is specified, Multiply To() returns a missing value. Any arguments after the first two are ignored.
   For a*=b: More than two arguments may be strung together. JMP evaluates pairs from right to left, and each product is placed in the left-hand variable. All arguments except the last must be a variable.

Example:
   a*=b*=c
   JMP multiplies b and c and places the product into b. Then JMP multiplies a and b and places the product into a.

See Also:
   The Matrix Algebra chapter in the JMP Scripting Guide.
Post Decrement(a)

\[ a-- \]

Function:
Post-decrement: Subtracts 1 from \( a \) and places the difference into \( a \).

Returns:
a-1

Argument:
a Must be a variable.

Notes:
If \( a-- \) or Post Decrement(a) is inside another expression, the expression is evaluated first, and then the decrement operation is performed. This expression is mostly used for loop control.

See Also:
The Conditional Expressions and Loops section of the JMP Scripting Guide.

Post Increment(a)

\[ a++ \]

Function:
Post-increment: Adds 1 to \( a \) and places the sum into \( a \).

Returns:
a+1

Argument:
a Must be a variable.

Notes:
If \( a++ \) or Post Increment(a) is inside another expression, the expression is evaluated first, and then the increment operation is performed. Mostly used for loop control.

See Also:
The Conditional Expressions and Loops section of the JMP Scripting Guide.

Power(a, <b>)

\[ a^b \]

Function:
Raises \( a \) to the power of \( b \).

Returns:
The product of \( a \) multiplied by itself \( b \) times.

Arguments:
a Can be a variable, number, or matrix.
b Optional. Can be a variable or a number.

Notes:
For \( \text{Power}() \), the second argument \( (b) \) is optional, and the default value is 2. \( \text{Power}(a) \) returns \( a^2 \).
Subtract(a, b)

a-b-

Function:
Subtracts the values of the listed arguments, left to right. No arguments are changed.

Returns:
The difference.

Arguments:
Two or more variables, numbers, or matrices.

Notes:
Two or more arguments are permitted. Specifying fewer than two arguments produces an error.

See Also:
The Matrix Algebra chapter in the JMP Scripting Guide.

Subtract To(a, b)

a-=b

Function:
Subtracts $b$ from $a$ and places the difference into $a$.

Returns:
The difference.

Arguments:
- $a$ Must be a variable.
- $b$ Can be a variable, number, or matrix.

Notes:
The first argument must be a variable, because its value must be able to accept a value change. A number as the first argument will produce an error.

For SubtractTo(): Only two arguments are permitted. If fewer than two or more than two arguments are specified, SubtractTo() returns a missing value.

For $a-=b$: More than two arguments may be strung together. JMP evaluates pairs from right to left, and each difference is placed in the left-hand variable. All arguments except the last must be a variable.

Example:
$a-=b-=c$

JMP subtracts $c$ from $b$ and places the difference into $b$. Then JMP subtracts $b$ from $a$ and places the difference into $a$.

See Also:
The Matrix Algebra chapter in the JMP Scripting Guide.
Numeric Functions

Abs(n)
Function:
Calculates the absolute value of n.
Returns:
Returns a positive number of the same magnitude as the value of n.
Argument:
n Any number, numeric variable, or numeric expression.

Ceiling(n)
Function:
If n is not an integer, rounds n to the next highest integer.
Returns:
Returns the smallest integer greater than or equal to n.
Argument:
n any number

Floor(n)
Function:
If n is not an integer, rounds n to the next lowest integer.
Returns:
Returns the largest integer less than or equal to n.
Argument:
n any number

Modulo(number, divisor)
Mod(number, divisor)
Function:
Returns the remainder when number is divided by divisor.

Round(n, places)
Function:
Rounds n to number of decimal places given.
Transcendental Functions

Arrenhius(n)

Function:
Converts the temperature \( n \) to activation energy.

Returns:
\[ \frac{11605}{n+273.15} \]

Argument:
\( n \) Temperature in Celsius.

Notes:
This is frequently used as a transformation.

Arrenhius Inv(n)

Function:
The inverse of the Arrhenius function. Converts the activation energy \( n \) to temperature in Celsius.

Returns:
\[ \frac{11605}{n-273.15} \]

Argument:
\( n \) Activation energy.

Notes:
This is frequently used as a transformation.

Beta(a, b)

Function:
\[ \frac{\Gamma(a)\Gamma(b)}{\Gamma(a + b)} \]

Returns:
Returns the beta function.

Arguments:
a, b numbers

Digamma(n)

Function:
The derivative of the log of the gamma function (\( \text{LGamma} \)).

Returns:
The digamma function evaluated at \( n \).

Argument:
\( n \) A number
**Exp(a)**

**Function:**
 Raises e to the power \(a\).

**Returns:**
\(e^a\).

**Argument:**
\(a\) A number

**Equivalent Expression:**
\(e^n^a\)

---

**Factorial(n)**

**Function:**
 Multiplies all numbers 1 through \(n\), inclusive

**Returns:**
The product.

**Arguments:**
\(n\) Any integer

**Notes:**
One and only one argument must be specified.

---

**Gamma(x, <limit>)**

**Function:**
The gamma function of \(x\), or for each row if \(x\) is a column:

\[
\Gamma(i) = \int_{0}^{\infty} (x^{i-1}) e^{-x} dx.
\]

**Returns:**
The gamma.

**Note:**
\(\Gamma(x, \text{limit})\) is the same integral as \(\Gamma(x)\) but with the limit of integration \(x\) instead of infinity. \(\Gamma(x, \text{limit})\) is equivalent to \(\Gamma\text{ Distribution}(x, \text{limit}) \ast \Gamma(x)\). Note that the arguments are in a different order.

**Arguments:**
\(x\) a number or a column
\(\text{limit}\) optional limit. The default is infinity.

---

**IGamma(t, shape)**

**Function:**
Returns the incomplete gamma function for \(t\).
LGamma(t)
Function:
Returns the log gamma function for \( t \), which is the natural log of gamma.

Log(n)
Log(n, base)
Ln(n)
Function:
Returns the natural (base \( e \)) or base \( base \) logarithm of \( n \).

Log10(n)
Function:
Returns the common (base 10) logarithm of \( n \).

Log1P(n)
Function:
Same as Log(1 + \( x \)), except that it is more accurate when \( x \) is very small.

Logit(p)
Function:
Returns \( \log(p/(1-p)) \).

N Choose K(n, k)
Function:
The number of ways to choose \( k \) items out of \( n \), ignoring order.
Returns:
The number of \( n \) things taken \( k \) at a time ("\( n \) choose \( k \)).

Root(n, r)
Function:
Returns the \( r \)th root of \( n \), where \( r \) defaults to 2 for square root.

Scheff\'e Cubic(x1, x2)
Function:
Returns \( x1 \times x2^*(x1-x2) \). This function supports notation for cubic mixture models.
**Sqrt(n)**

**Function:**
- Returns the square root of \( n \).

**Squash(expr)**

**Function:**
- An efficient computation of the function \( 1/ [1 + \exp(expr)] \).

**Squish(expr)**

**Function:**
- Equivalent to \( \text{Squash}(-\text{expr}) \), or \( 1/[1 + \exp(-\text{expr})] \).

**Trigamma()**

**Function:**
- Returns the trigamma function evaluated at \( n \). The trigamma function is the derivative of the digamma function.

**Trigonometric Functions**

**ArcCosH(x)**

**Function:**
- Inverse hyperbolic cosine.

**Returns:**
- The inverse hyperbolic cosine of \( x \).

**Argument:**
- \( x \) Any number, numeric variable, or numeric expression.

**ArcCosine(x)**

**ArCos(x)**

**Function:**
- Inverse cosine.

**Returns:**
- The inverse cosine of \( x \), an angle in radians.

**Argument:**
- \( x \) Any number, numeric variable, or numeric expression.
ArcSine(x)
ArcSin(x)
Function:
Inverse sine.
Returns:
The inverse sine of x, an angle in radians.
Argument:
\(x\) Any number, numeric variable, or numeric expression.

ArcSinH(x)
Function:
Inverse hyperbolic sine.
Returns:
The inverse hyperbolic sine of x.
Argument:
\(x\) Any number, numeric variable, or numeric expression.

ArcTangent(x)
ArcTan(x)
ATan(x)
Function:
Inverse tangent.
Returns:
The inverse tangent of x, an angle in radians.
Argument:
\(x\) Any number, numeric variable, or numeric expression.

ArcTanH(x)
Function:
Inverse hyperbolic tangent.
Returns:
The inverse hyperbolic tangent of x.
Argument:
\(x\) Any number, numeric variable, or numeric expression.
Cosine(x)
Cos(x)
Function:
Cosine.
Returns:
The cosine of x.
Argument:
  x Any number, numeric variable, or numeric expression. The angle in radians.

CosH(x)
Function:
Hyperbolic cosine.
Returns:
The hyperbolic cosine of x.
Argument:
  x Any number, numeric variable, or numeric expression.

Sine(expr)
Sin(expr)
Function:
Returns the sine.

SinH(expr)
Function:
Returns the hyperbolic sine.

Tangent(expr)
Tan(expr)
Function:
Returns the tangent of an argument given in radians.

TanH(expr)
Function:
Returns the hyperbolic tangent of its argument.
Random Functions

Col Shuffle()

Function:  
Shuffles the values randomly each time it's evaluated.

Returns:  
The last value placed into the last row.

Argument:  
none

Example:  
foreachrow(x=shuffle = colshuffle());
  show(x)
  x:6

In a table with 40 rows, the above script places the values 1-40 randomly into each row of the
column named shuffle. All numbers appear only once. Each time the script is run, the numbers are
placed in a different random order. If the value placed into the row 40 of column shuffle is 6, that
number is assigned to x, as in the above example.

Random Beta(alpha, beta)

Function:  
Generates a pseudo-random number distributed Beta (alpha, beta)

Random Beta Binomial(n, p, delta)

Function:  
Returns a random number from a beta binomial distribution for n trials with probability p and
correlation delta.

Random Binomial(n, p)

Function:  
Returns random numbers from a binomial distribution with n trials and probability p of the event of
interest occurring.

Random Cauchy(alpha, beta)

Function:  
Returns a Cauchy distribution with given alpha and beta.

Random Exp()

Function:  
Returns a random number distributed exponentially from 0 to infinity. Equivalent to the negative
log of Random Uniform.
Random Gamma(\(\lambda, <\text{scale}\))

Function: Gives a gamma distribution for given \(\lambda\) and optional \(\text{scale}\).

Random Gamma Poisson(\(\lambda, \sigma\))

Function: Returns a random number from a gamma Poisson distribution with parameters \(\lambda\) and \(\sigma\).

Random Geometric()

Function: Returns random numbers from the geometric distribution with probability \(p\) that a specific event occurs at any one trial.

Random GLog(\(\mu, \sigma, \lambda\))

Function: Returns a random number from a generalized logarithmic distribution.

Random Integer(\(<k>, n\))

Function: Returns a random integer from 1 to \(n\) or from \(k\) to \(n\).

Random Johnson Sb(\(\gamma, \delta, \theta, \sigma\))

Function: Returns a random number from the Johnson Sb distribution.

Random Johnson Sl(\(\gamma, \delta, \theta, <\sigma=1>\))

Function: Returns a random number from the Johnson Sl distribution.

Random Johnson Su(\(\gamma, \delta, \theta, \sigma\))

Function: Returns a random number from the Johnson Su distribution.

Random Lognormal(\(\mu, \sigma\))

Function: Returns a Lognormal-distributed random number with location parameter \(\mu\) and scale parameter \(\sigma\).
Random Negative Binomial\( (r, p) \)
Function:
Generates a negative binomial distribution for \( r \) successes with probability of success \( p \).

Random Normal\( (\ ) \)
Function:
Generates random numbers that approximate a normal distribution with mean 0 and variance 1.

Random Poisson\( (\lambda) \)
Function:
Generates a Poisson variate for given \( \lambda \).

Random Reset\( (\text{seed}) \)
Function:
Restarts the random number sequences with \( \text{seed} \).

Random Seed\( (\ ) \)
Function:
Sets seed for the seeded random number generator functions Random Seeded Normal and Random Seeded Uniform.
Note:
This function, Random Seeded Normal, and Random Seeded Uniform have been deprecated.

Random Seeded Normal\( (\ ) \)
Function:
Generates random numbers that approximate a normal distribution with mean 0 and variance 1 using Random Seed as the seed value.
Note:
This function has been deprecated.

Random Seeded Uniform\( (\ ) \)
Function:
Generates random numbers uniformly between 0 and 1 using Random Seed as the seed value.
Note:
This function has been deprecated.

Random Triangular\( (\text{midpoint}) \)
Function:
Generates a triangular distribution of numbers between 0 and 1 with the midpoint you specify.
Random Uniform()  
Random Uniform(x)  
Random Uniform(high, low)  

Function:  
Generates random numbers uniformly between 0 and 1. Random Uniform(x) generates numbers between 0 and x. Random Uniform (high, low) generates numbers between low and high.

Random Weibull(beta, alpha)  
Function:  
Returns a random number from a Weibull distribution.

Resample Freq()  
Resample Freq(fraction)  
Resample Freq(n)  

Function:  
ResampleFreq() generates 100% resample. ResampleFreq(fraction) generates fraction*nrow frequency sample. ResampleFreq(n) generates an n frequency sample.

Character/String Functions

As Name(string)  
Function:  
Evaluates argument as a string and changes it into a name.  
Returns:  
A name.

Char(x, <width>, <decimal>)  
Function:  
Converts an expression or numeric value into a character string.  
Returns:  
A string.  
Arguments:  
  x an expression or a numeric value.  
  width optional number that sets the maximum number of characters in the string.
Functions, Variables, and Other Basic Tools

decimal optional number that sets the maximum number of places after the decimal that is included in the string.

**Note:**
width overrides decimal.

**Example:**

```
char(pi(), 10, 4)
"3.1416"

char(pi(), 3, 4)
"3.1"
```

---

**Concat(a, b)**

a||b

**Function:**
Appends the string \( b \) to the string \( a \). Neither argument is changed.

**Returns:**
A string composed of the string \( a \) directly followed by the string \( b \).

**Arguments:**
Two or more strings or string variables.

**Notes:**
More than two arguments may be strung together. Each additional string is appended to the end, in left to right order. This function also works with matrices as the arguments.

**Example:**

```
a="Hello"; b=" "; c="World"; a||b||c;
"Hello World"
```

---

**Concat Items({string1, string2, ...}, <delimiter>}**

**Function:**
Converts a list of string expressions into one string, with each item separated by a delimiter. The delimiter is a blank, if unspecified.

**Returns:**
The concatenated string.

**Arguments:**
string any string
delimeter an optional string that is placed in front of each component string. The delimiter can be more than one character long.

**Example:**

```
str1 = "one";
str2 = "two";
str3 = "three";

comb = concat Items ({str1, str2, str3});
" one two three"
```
JSL Syntax Reference
Functions, Variables, and Other Basic Tools

comb = concat Items ({{str1, str2, str3}, " : ");
    " : one : two : three"
del = ";
comb = concat Items ({{str1, str2, str3}, del};
    ",one,two,three"

Concat To(a, b)
a||=b

Function:
Appends the string b to the string a and places the new concatenated string into a.

Returns:
A string composed of the string a directly followed by the string b.

Arguments:
Two or more strings or string variables. The first must be a variable whose value can be changed.

Notes:
More than two arguments may be strung together. Each additional string is appended to the end, in
left to right order. This function also works with matrices as the arguments.

Example:
a="Hello"; b=" "; c="World"; ConcatTo(a, b, c); Show(a);
a:"Hello World"

Contains(whole, part, <start>)

Function:
Determines if part is contained within whole.

Returns:
If part is found: For lists and strings, the numeric position where the first occurrence of part is
located. For associative arrays, 1.
If part is not found, 0 is returned in all cases.

Arguments:
whole A string, list, or associative array.
part For a string, a string which may be part of the string whole. For a list, an item which may be
an item in the list whole. For an associative array, a key that may be one of the keys in the map
whole.
start An optional numeric argument that specifies a starting point. within whole. If start is
negative, contains searches whole for part backwards, beginning with the position specified by
the length of whole – start. Note that start is meaningless for associative arrays and is ignored.
**Ends With(string, substring)**

**Function:**
Determines if substring appears at the end of string.

**Returns:**
1 if string ends with substring, otherwise 0.

**Arguments:**
- string A quoted string or a reference to one. May also be a list.
- substring A quoted string or a reference to one. May also be a list.

**Equivalent Expression:**
Right(string, Length(substring)) == substring

**Insert(source, item, <position>)**

**Function:**
Inserts a new item into the source at the given position. If position is not given, item is added to the end.

**Arguments:**
- source A string, list, or expression.
- item Any value to be placed within source.
- position Optional numeric value that specifies the position in source to place the item into.

**Note:**
Insert may also be used with associative arrays. See “Insert(source, key, value),” p. 510 for details.

**Insert Into(source, item, <position>)**

**Function:**
Inserts a new item into the source at the given position in place. The source must be an L-value.

**Arguments:**
- source A string, list, or expression.
- item Any value to be placed within source.
- position Optional numeric value that specifies the position in source to place the item into.

**Note:**
Insert Into may also be used with associative arrays. See “Insert(source, key, value),” p. 510 for details.

**Item(n, string, <delimiters>)**

**Function:**
Extracts the nth word from a quoted string according to the quoted string delimiters given.

**Note:**
Item() is the same as Word() except that Item() treats each delimiter character as a separate delimiter, and Word() treats several adjacent delimiters as a single delimiter.
Left(string, n, <filler>)
Left(list, n, <filler>)

Function:
Returns a truncated or padded version of the original string or list. The result contains the left n characters or list items, padded with any filler on the right if the length of string is less than n.

Length(string)

Function:
Calculates the number of characters (length) of the quoted string.

Lowercase(string)

Function:
Converts any upper case character found in quoted string to the equivalent lowercase character.

Munger(string, offset, find/length)
Munger(string, offset, find, replace)

Function:
Computes new character strings from the quoted string by inserting or deleting characters.

Num(string)

Function:
Converts a character string into a number.

Parse(string)

Function:
Converts a character string into a JSL expression.

Regex(source, text, <format>, <IGNORECASE>)

Function:
Searches for the text within the source string.

Returns:
The matched text as a string or numeric missing if there was no match.

Arguments:
source  A quoted string.
text  A quoted regular expression.
format  Optional. The default is \0, which is the entire matched string, \n returns the nth match.
IGNORECASE  Optional. The search is case sensitive, unless you specify IGNORECASE.
Regex Match()

See “Regex Match(source, pattern),” p. 594.

Remove(source, position, <n>)

Function: Deletes the n item(s), starting from the indicated position. If n is omitted, the item at position is deleted. If position and n are omitted, the item at the end is removed.

Returns: A copy of the source with the items deleted.

Arguments: source A string, list, or expression.
position An integer (or list of integers) that points to a specific item (or items) in the list or expression.
  n Optional. An integer that specifies how many items to remove.

Note: Remove may also be used with associative arrays. See “Remove(source, key),” p. 511 for details.

Remove From(source, position, <n>)

Function: Deletes the n item(s) in place, starting from the indicated position. If n is omitted, the item at position is deleted. If position and n are omitted, the item at the end is removed. The source must be an L-value.

Returns: The original source with the items deleted.

Arguments: source A string, list, or expression.
position An integer (or list of integers) that points to a specific item (or items) in the list or expression.
  n Optional. An integer that specifies how many items to remove.

Note: Remove From may also be used with associative arrays. See “Remove(source, key),” p. 511 for details.

Repeat(source, a)

Repeat(matrix, a, b)

Function: Returns a copy of source concatenated with itself a times. Or returns a matrix composed of a row repeats and b column repeats. The source can be text, a matrix, or a list.
Reverse(source)

Function:
Reverse order of elements or terms in the source.

Argument:
source A string, list, or expression.

Reverse Into(source)

Function:
Reverse order of elements or terms in list or expr in place.

Argument:
source A string, list, or expression.

Right(string, n, <filler>)
Right(list, n, <filler>)

Function:
Returns a truncated or padded version of the original string or list. The result contains the right n characters or list items, padded with any filler on the left if the length of string is less than n.

Set Clipboard(string)

Function:
Evaluates the string argument looking for a character result, then places the string on the clipboard.

Shift(source, <n>)

Function:
Shifts an item or n items from the front to the back of the source.

Arguments:
source A string, list, or expression.

n Optional. An integer that specifies the number of items to shift. Positive values shift items from the beginning of the source to the end. Negative values shift items from the end of the source to the beginning. The default value is 1.

Shift Into(source, <n>)

Function:
Shifts items in place.

Arguments:
source A string, list, or expression.

n Optional. An integer that specifies the number of items to shift. Positive values shift items from the beginning of the source to the end. Negative values shift items from the end of the source to the beginning. The default value is 1.
Starts With(string, substring)

Function:
Determine if substring appears at the start of string.

Returns:
1 if string starts with substring, otherwise 0.

Arguments:
- string A quoted string or a reference to one. May also be a list.
- substring A quoted string or a reference to one. May also be a list.

Equivalent Expression:
Left(string, Length(substring)) = substring

Substitute(string, substring, replacementString, ...)
Substitute(list, listItem, replacementItem, ...)
Substitute(Expr(sourceExpr), Expr(findExpr), Expr(replacementExpr), ...)

Function:
This is a search and replace function. It searches for a specific portion (second argument) of the source (first argument), and replaces it (third argument).

If a string, finds all matches to substring in the source string, and replaces them with the replacementString.

If a list, finds all matches to listItem in the source list, and replaces them with the replacementItem.

If an expression, finds all matches to the findExpr in the sourceExpr, and replaces them with the replacementExpr. Note that all expressions must be enclosed within an Expr() function.

Arguments:
- string, list, sourceExpr A string, list, or expression in which to perform the substitution.
- substring, listItem, findExpr A string, list item, or expression to be found in the source string, list, or expression.
- replacementString, replacementItem, replacementExpr A string, list item, or expression to replace the found string, list item, or expression.
### Substitute Into(string, substring, replacementString, ...)  
### Substitute Into(list, listItem, replacementItem, ...)  
### Substitute Into(Expr(sourceExpr), Expr(findExpr), Expr(replacementExpr), ...)  

**Function:**  
This is a search and replace function, identical to `Substitute()` except in place. It searches for a specific portion (second argument) of the source (first argument), and replaces it (third argument). The first argument must be an L-value.  
If a string, finds all matches to `substring` in the source `string`, and replaces them with the `replacementString`.  
If a list, finds all matches to `listItem` in the source `list`, and replaces them with the `replacementItem`.  
If an expression, finds all matches to the `findExpr` in the `sourceExpr`, and replaces them with the `replacementExpr`. Note that all expressions must be enclosed within an `Expr()` function.  

**Arguments:**  
- `string`, `list`, `sourceExpr` A string, list, or expression in which to perform the substitution.  
- `substring`, `listItem`, `findExpr` A string, list item, or expression to be found in the source string, list, or expression.  
- `replacementString`, `replacementItem`, `replacementExpr` A string, list item, or expression to replace the found string, list item, or expression.

---  

### Substr(string, start, length)  

**Function:**  
Extracts the characters that are the portion of the `string` beginning at `start` and ending at `length`.  

---  

### Trim("text")  

**Function:**  
Produces a new character string from its argument, removing any trailing blanks.  

---  

### Uppercase(string)  

**Function:**  
Converts any lower case character found in the quoted `string` to the equivalent uppercase character.

---  

### Word(n, "text", <"delimiters">)  

**Function:**  
Extracts the \textit{n}th word from a character string according to the delimiters given.

---  

### Words("text", <"delimiters">)  

**Function:**  
Extracts the words from \textit{text} according to the delimiters given.
Hex and BLOB Functions

**Blob MD5(blob)**

*Function:* Converts the blob argument into a 16-byte blob.

*Note:* The 16-byte blob is the MD5 checksum, or the hash, of the source blob.

**Blob Peek(blob, offset, length)**

*Function:* Creates a new blob from a subrange of bytes of the blob argument.

*Returns:* A blob object.

*Arguments:*
- **blob** a binary large object.
- **offset** An integer that specifies how many bytes into the blob to begin construction. The first byte is at offset 0, the second byte at offset 1.
- **length** An integer that specifies how many bytes to copy into the new blob, starting at the offset.

**Blob To Char(blob, <encoding>)**

*Function:* Reinterpret binary data as a Unicode string.

*Returns:* A string.

*Arguments:*
- **blob** a binary large object.
- **encoding** Optional quoted string that specifies an encoding. Supported encodings are: utf-8, utf-16le, utf-16be, us-ascii, iso-8859-1, and ascii-hex.

*Notes:* The optional argument ascii is intended to make conversions of blobs containing normal ASCII data simpler when the data might contain CR, LF, or TAB characters (for example) and those characters don’t need any special attention.

**Char To Blob(string, <encoding>)**

*Function:* Converts a string of characters into a binary (blob).

*Returns:* A binary object.

*Arguments:*
- **string** a quoted string or a reference to a string.
encoding Optional quoted string that specifies an encoding. Supported encodings are: utf-8, utf-16le, utf-16be, us-ascii, iso-8859-1, and ascii-hex.

**Char To Hex(value, <"integer" | encoding>)**

**Hex(value, <"integer" | encoding>)**

Function:
Converts the given value and encoding into its hexadecimal representation.

Returns:
A hexadecimal string.

Arguments:
- value Any number, quoted string, or blob.
- integer Switch that causes the value to be interpreted as an integer.
- encoding Optional quoted string that specifies an encoding. Supported encodings are: utf-8, utf-16le, utf-16be, us-ascii, iso-8859, and ascii-hex.

**Hex To Blob(string)**

Function:
Converts the quoted hexadecimal string (including whitespace) to a blob (binary large object).

Example:
Hex To Blob("4A4D50");
Char To Blob("JMP")

**Hex To Char(string)**

Function:
Converts the quoted hexadecimal string to its character equivalent.

Example:
Hex To Char ("30") results in "0".

**Hex To Number(string)**

Function:
Converts the quoted hexadecimal string to its integer equivalent.

Example:
Hex To Number("80");
128

Note:
16-digit hex numbers are converted as IEEE 754 64-bit floating point numbers.

**Hex(value, <"integer" | encoding="enc")**

See "Char To Hex(value, <"integer" | encoding>)," p. 468.
Comparison/Logical Functions

And(a, b)

Function:
Logical And.

Returns:
1 (true) if both a and b are true.
0 (false) if either a or b is false or if both a and b are false.
Missing if either a or b is a missing values or if both a and b are missing values.

Arguments:
Two or more variables or expressions.

Notes:
More than two arguments may be strung together. \texttt{a\&b} returns 1 (true) only if all arguments evaluate to true.

See Also:
The Conditional Expressions and Loops section of the \textit{JMP Scripting Guide}.

AndMZ(a, b)

Function:
Logical And with JMP 3 behavior, which treats missing values as 0.

Returns:
1 (true) if both a and b are true.
0 (false) if either a or b is false or if both a and b are false.
0 (false) if either a or b is a missing values or if both a and b are missing values.

Arguments:
Two or more variables or expressions.

Notes:
More than two arguments may be strung together. \texttt{a:\&b} returns 1 (true) only if all arguments evaluate to true. When opening a JMP 3 data table, this function is automatically used for any \texttt{And} function.

See Also:
The Conditional Expressions and Loops section of the \textit{JMP Scripting Guide}. 
Equal(a, b, ...)
a==b==...

Function:
Compares all the listed values and tests if they are all equal to each other.

Returns:
1 (true) if all arguments evaluate to the same value.
0 (false) otherwise.

Arguments:
Two or more variables, references, matrices, or numbers.

Notes:
If more than two arguments are specified, a 1 is returned only if all arguments are exactly the same.
This is typically used in conditional statements and to control loops.

See Also:
The Conditional Expressions and Loops section of the JMP Scripting Guide.

Greater(a, b, ...)
a>b>...

Function:
Compares all the list values and tests if, in each pair, the left value is greater than the right.

Returns:
1 (true) if \( a \) evaluates strictly greater than \( b \) (and \( b \) evaluates strictly greater than \( c \), etc.).
0 (false) otherwise.

Arguments:
Two or more variables, references, matrices, or numbers.

Notes:
If more than two arguments are specified, a 1 is returned only if each argument is greater than the one that follows it. This is typically used in conditional statements and to control loops.
\textbf{Greater}, \textbf{Less}, \textbf{GreaterOrEqual}, and \textbf{LessOrEqual} can also be strung together. If you do not group with parentheses, JMP will evaluate each pair left to right. You can also use parentheses to explicitly tell JMP how to evaluate the expression.

See Also:
The Conditional Expressions and Loops section of the JMP Scripting Guide.
Greater or Equal(a, b, ...)
\[ a \geq b \geq \ldots \]

**Function:**
Compares all the list values and tests if, in each pair, the left value is greater than or equal to the right.

**Returns:**
1 (true) if \( a \) evaluates strictly greater than or equal to \( b \) (and \( b \) evaluates strictly greater than or equal to \( c \), etc.).
0 (false) otherwise.

**Arguments:**
Two or more variables, references, matrices, or numbers.

**Notes:**
If more than two arguments are specified, a 1 is returned only if each argument is greater than or equal to the one that follows it. This is typically used in conditional statements and to control loops. Greater, Less, GreaterOrEqual, and LessOrEqual can also be strung together. If you do not group with parentheses, JMP will evaluate each pair left to right. You can also use parentheses to explicitly tell JMP how to evaluate the expression.

**See Also:**
The Conditional Expressions and Loops section of the JMP Scripting Guide.

Is Missing(expr)

**Function:**
Returns 1 if the expression yields a missing value and 0 otherwise.

Less(a, b, ...)
\[ a < b < \ldots \]

**Function:**
Compares all the list values and tests if, in each pair, the left value is less than the right.

**Returns:**
1 (true) if \( a \) evaluates strictly less than \( b \) (and \( b \) evaluates strictly less than \( c \), etc.).
0 (false) otherwise.

**Arguments:**
Two or more variables, references, matrices, or numbers.

**Notes:**
If more than two arguments are specified, a 1 is returned only if each argument is less than the one that follows it. This is typically used in conditional statements and to control loops. Greater, Less, GreaterOrEqual, and LessOrEqual can also be strung together. If you do not group with parentheses, JMP will evaluate each pair left to right. You can also use parentheses to explicitly tell JMP how to evaluate the expression.

**See Also:**
The Conditional Expressions and Loops section of the JMP Scripting Guide.
Less or Equal(a, b, ...)

Function:
Compares all the list values and tests if, in each pair, the left value is less than or equal to the right.

Returns:
1 (true) if \(a\) evaluates strictly less than or equal to \(b\) (and \(b\) evaluates strictly less than or equal to \(c\), etc.).
0 (false) otherwise.

Arguments:
Two or more variables, references, matrices, or numbers.

Notes:
If more than two arguments are specified, a 1 is returned only if each argument is less than or equal to the one that follows it. This is typically used in conditional statements and to control loops.

Greater, Less, GreaterOrEqual, and LessOrEqual can also be strung together. If you do not group with parentheses, JMP will evaluate each pair left to right. You can also use parentheses to explicitly tell JMP how to evaluate the expression.

See Also:
The Conditional Expressions and Loops section of the JMP Scripting Guide.

Less LessEqual(a, b, c)

Function:
Range check, exclusive below and inclusive above.

Returns:
1 (true) if \(b\) is greater than \(a\) and less than or equal to \(c\).
0 (false) otherwise.

Arguments:
a, b, c variables, references, matrices, or numbers.

Notes:
If more than two arguments are specified, a 1 is returned only if each argument is less than the one that follows it. This is typically used in conditional statements and to control loops.

See Also:
The Conditional Expressions and Loops section of the JMP Scripting Guide.
\texttt{LessEqual \texttt{Less(a, b, c)}}
\begin{verbatim}
a<=b<c
\end{verbatim}

\textbf{Function:}
Range check, inclusive below and exclusive above.

\textbf{Returns:}
\begin{itemize}
  \item 1 (true) if \( b \) is greater than or equal to \( a \) and less than \( c \).
  \item 0 (false) otherwise.
\end{itemize}

\textbf{Arguments:}
\( a, \ b, \ c \) variables, references, matrices, or numbers.

\textbf{Notes:}
If more than two arguments are specified, a 1 is returned only if each argument is less than the one that follows it. This is typically used in conditional statements and to control loops.

\textbf{See Also:}
The Conditional Expressions and Loops section of the \textit{JMP Scripting Guide}.

\begin{verbatim}
Not(a)
!a
\end{verbatim}

\textbf{Function:}
Logical Not.

\textbf{Returns:}
\begin{itemize}
  \item 0 (false) if \( a > 0 \).
  \item 1 (true) if \( a \leq 0 \).
  \item Missing value if \( a \) is missing.
\end{itemize}

\textbf{Argument:}
\( a \) Any variable or number. The variable must have a numeric or matrix value.

\textbf{Notes:}
Mostly used for conditional statements and loop control.

\textbf{See Also:}
The Conditional Expressions and Loops section of the \textit{JMP Scripting Guide}. 
Not Equal(a, b)

\( a \neq b \)

**Function:**
Compares a and b and tests if they are equal.

**Returns:**
- 0 (false) if \( a \) and \( b \) evaluate to the same value.
- 1 (true) otherwise.

**Argument:**
- \( a, b \) Any variable or number.

**Notes:**
Mostly used for conditional statements and loop control.

**See Also:**
The Conditional Expressions and Loops section of the "JMP Scripting Guide".

Or(a, b)

\( a \mid b \)

**Function:**
Logical Or.

**Returns:**
- 1 (true) if either of or both \( a \) and \( b \) are true.
- 0 (false) otherwise.
- Missing if either are missing.

**Arguments:**
- \( a, b \) Any variable or number.

**Notes:**
Mostly used for conditional statements and loop control.

**See Also:**
The Conditional Expressions and Loops section of the "JMP Scripting Guide".
**OrMZ(a, b)**

Function:
Logical Or with version 3.x behavior, which treats missing values as 0.

Returns:
1 (true) if either of or both \( a \) and \( b \) are true.
0 (false) otherwise.
Missing if either are missing.

Arguments:
\( a, b \) Any variable or number.

Notes:
Mostly used for conditional statements and loop control. When opening a JMP 3 data table, this function is automatically used for any Or function.

See Also:
The Conditional Expressions and Loops section of the *JMP Scripting Guide*.

---

**Zero Or Missing(expr)**

Function:
Returns 1 if \( expr \) yields a missing value or zero, 0 otherwise.

---

**Date/Time Functions**

**Abbrev Date(date)**

Function:
Converts the provided \( date \) to a string.

Returns:
A string representation of the date.

Argument:
\( date \) Can be the number of seconds since the base date (midnight, January 1, 1904), or any date-time operator.

Example:
Abbrev Date(29Feb2004)

\[ Feb \ 29, \ 2004 \]

See Also:
Section on Date-Time formats in the *JMP Scripting Guide*. 
As Date(x)

Function:
Formats the number or expression x so that it shows as a date when displayed in a text window.

Returns:
A date which is calculated from the number or expression provided.

Argument:
x Number or expression.

See Also:
Section on Date-Time formats in the JMP Scripting Guide.

Date DMY(day, month, year)

Function:
Constructs a date value from the arguments.

Returns:
The specified date, expressed as the number of seconds since midnight, 1 January 1904.

Arguments:

day number, day of month, 1-31. Note that there's no error-checking, so you can enter February 31.

month number of month, 1-12.

year number of year.

Date MDY(month, day, year)

Function:
Constructs a date value from the arguments.

Returns:
The specified date, expressed as the number of seconds since midnight, 1 January 1904.

Arguments:

month number of month, 1-12.

day number, day of month, 1-31. Note that there's no error-checking, so you can enter February 31.

year number of year.
Day(datetime)

Function:
Determine the day of the month supplied by the `datetime` argument.

Returns:
Returns an integer representation for the day of the month of the date supplied.

Arguments:
- `datetime` Number of seconds since midnight, 1 January 1904. This can also be an expression.

Example:
```plaintext
d1=datedmy(12, 2, 2003);
3127852800
day(3127852800);
12
day(d1);
12
```

Day Of Week(datetime)

Function:
Determine the day of the week supplied by the `datetime` argument.

Returns:
Returns an integer representation for the day of the week of the date supplied.

Arguments:
- `datetime` Number of seconds since midnight, 1 January 1904. This can also be an expression.

Day Of Year(datetime)

Function:
Determine the day of the year supplied by the `datetime` argument.

Returns:
Returns an integer representation for the day of the year of the date supplied.

Arguments:
- `datetime` Number of seconds since midnight, 1 January 1904. This can also be an expression.

Format(x, "format", <"currency code">, <decimal>)

Format Date(x, "format", <"currency code">, <decimal>)

Function:
Converts the value of `date` into the `format` you specify in the second argument.

Returns:
Returns the converted date in the format specified.

Arguments:
- `x` Can be a column, a number, or a `datetime`. 
format  Any valid format, as a quoted string: Best, Fixed Decimal, Percent, PValue, Scientific, Currency, or any of the Date/Time formats.
currency code  An optional ISO 4217 code for a specific currency; for example, GBP for Great Britain, Pound. This argument is valid only if format is specified as Currency.
decimal  An optional integer that specifies the number of decimal places to be shown.

Hour(datetime, <12|24>)
Function:
  Determines the hour supplied by the datetime argument.
Returns:
  Returns an integer representation for the hour part of the date-time value supplied.
Arguments:
  datetime  Number of seconds since midnight, 1 January 1904. This can also be an expression.
  12|24  Changes the mode to 12 hours (with am and pm). The default is 24-hour mode.

In Minutes(n); In Hours(n); In Days(n); In Weeks(n); In Years(n)
Function:
  These operators return the number of seconds per n minutes, hours, days, weeks, or years.

In Format(string, format)
Parse Date(string, format)
Function:
  Parses a quoted string of a given quoted format and returns a date/time value expressed as number of
  seconds since 12am, 1 January 1904.
Example:
  Informat("07152000", "MMDDYYYY")
    15Jul2000

Long Date(date)
Function:
  Returns a locale-specific string representation for the date supplied, formatted like "Sunday,
  February 29, 2004".

MDYHMS(date)
Function:
  Returns a string representation for the date supplied, formatted like "2/29/04 00:02:20".
**Minute** (datetime)

Function:
Determines the minute supplied by the `datetime` argument, 0-59.

Returns:
Returns an integer representation for the minute part of the date-time value supplied.

**Month** (date)

Function:
Returns an integer representation for the month of the `date` supplied.

**Second** (datetime)

Function:
Determines the second supplied by the `datetime` argument.

Returns:
Returns an integer representation for the second part of the date-time value supplied.

Argument:
- `datetime` Number of seconds since midnight, 1 January 1904. This can also be an expression.

**Short Date** (date)

Function:
Returns a string representation for the `date` supplied, in the format mm/dd/yy, e.g. "2/29/04" for the next Leap Day.

**Time Of Day** (date)

Function:
Returns an integer representation for the time of day of the `date/time` supplied

**Today** ()

Function:
Returns the current date and time expressed as the number of seconds since midnight, 1 January 1904.

**Week Of Year** (date)

Function:
Returns an integer representation of week of year for `date`.

**Year** (date)

Function:
Returns an integer representation for the year of `date`.
General Utility Functions

**Current Data Table(<dt>)**

**Function:**
Without an argument, gets the current (topmost) data table. With an argument, sets the current data table.

**Returns:**
Reference to the current data table.

**Argument:**

dt  Optional name of or reference to a data table.

**Current Journal()**

**Function:**
Gets the display box at the top of the current (topmost) journal.

**Returns:**
Returns a reference to the display box at the top of the current journal.

**Current Report()**

**Function:**
Gets the display box at the top of the current (topmost) report window.

**Returns:**
Returns a reference to the display box at the top of the current report window.

**Current Window()**

**Function:**
Returns a reference to the current window.

**Decode64 Double("string")**

**Function:**
Creates a floating point number from a base 64 encoded string.

**Returns:**
A floating point number.

**Arguments:**

string  a base 64 encoded string.
Empty()

Function:
   Does nothing. Used in the formula editor for making empty boxes.

Returns:
   Missing.

Arguments:
   None.

Encode64 Double(n)

Function:
   Creates a base 64 encoded string from a floating point number.

Returns:
   A base 64 encoded string.

Arguments:
   n  a floating point number.

Functions for Identification

Is Alt Key()

Function:
   Returns 1 if the Alt key is being pressed, or 0 otherwise.

Note:
   On a Macintosh, Is Alt Key() tests for the Option key.

Is Associative Array(name)

Function:
   Returns 1 if the evaluated argument is an associative array, or 0 otherwise.

Is Command Key()

Function:
   Returns 1 if the Command key is being pressed, or 0 otherwise.

Is Context Key()

Function:
   Returns 1 if the Context key is being pressed, or 0 otherwise.
Is Control Key()
Function:
Returns 1 if the Control key is being pressed, or 0 otherwise.

Note:
On a Macintosh, Is Control Key() tests for the Command (⌘) key.

Is Empty(global)
Is Empty(dt)
Is Empty(col)
Function:
Returns 1 if the global variable, data table, or data column does not have a value (is uninitialized), or
0 otherwise.

Is Expr(x)
Function:
Returns 1 if the evaluated argument is an expression, or 0 otherwise.

Is List(x)
Function:
Returns 1 if the evaluated argument is a list, or 0 otherwise.

Is Matrix(x)
Function:
Returns 1 if the evaluated argument is a matrix, or 0 otherwise.

Is Name(x)
Function:
Returns 1 if the evaluated argument is a name-only expression, or 0 otherwise.

Is Number(x)
Function:
Returns 1 if the evaluated argument is a number or missing numeric value, or 0 otherwise.

Is Option Key()
Function:
Returns 1 if the Option key is being pressed, or 0 otherwise.
Is Scriptable(x)

Function:
Returns 1 if the evaluated argument is a scriptable object, or 0 otherwise.

Is Shift Key()

Function:
Returns 1 if the Shift key is being pressed, or 0 otherwise.

Is String(x)

Function:
Returns 1 if the evaluated argument is a string, or 0 otherwise.

Type(x)

Function:
Returns a string that names the type of object x is. The list of possible types is: Unknown, List, DisplayBox, Picture, Column, TableVar, Table, Empty, Pattern, Date, Integer, Number, String, Name, Matrix, RowState, Expression, Associative Array, Blob.

Functions for directories and filepaths

Convert File Path(path, <absolute|relative>, <posix|windows>, <base(path)>)

Function:
Converts a file path according to the arguments.

Returns:
The converted path.

Arguments:

path A pathname that can be either Windows or POSIX.

absolute|relative Optional, specifies whether the returned pathname is in absolute or relative terms. The default value is absolute.

posix|windows Optional, specifies whether the returned pathname is in Windows or POSIX style. The default is POSIX.

base(path) Optional, specifies the base pathname, useful if relative is specified. The default is the default directory.
Host Is(argument)

Function:
Determines whether the host environment is the specified OS.

Returns:
True (1) if the current host environment matches the argument, false (0) otherwise.

Argument:
Argument Windows, Mac, or Linux tests for the specified operating system. Bits32 or Bits64 tests for the specified 32-bit or 64-bit machine.

Note:
Only one argument may be tested at a time. Invalid arguments return false (0).

Files In Directory(path)

Function:
Returns a list of filenames in the path given.

Returns:
List of filenames.

Argument:
path A valid pathname.

Get Addr Info(address, <port>)

Function:
Converts a name to its numeric address.

Returns:
A list of strings. The first element is the command (Get Addr Info). The second is the results (for example, “ok” if the command was successful). The third is a list of strings of information. Included in that information is the address that corresponds to the name that was supplied.

Arguments:
address A string that specifies the name (for example, www.sas.com).
port The port of the address.

Get Current Directory()

Function:
Retrieves the current directory, which is used in the Open File window.

Returns:
The absolute pathname as a string.

Arguments:
none
Get Default Directory()

Function:
Retrieves the JMP default directory, which is used as a base for subsequent relative paths.

Returns:
The absolute pathname as a string.

Arguments:
none

Get File Search Path()

Function:
Retrieves the current list of directories to search for opening files.

Returns:
A list of pathnames as strings.

Arguments:
none

Get Name Info(address, <port>)

Function:
Converts a numeric address to its name.

Returns:
A list of strings. The first element is the command (GetNameInfo). The second is the results (for example, “ok” if the command was successful). The third is a list of strings of information. Included in that information is the port name that corresponds to the address that was supplied.

Arguments:
address A string that specifies the numeric address (for example, 149.173.5.120).
port The port of the address.

Get Path Variable()

Function:
Retrieves the value of name, a path variable.

Returns:
The absolute pathname as a string.

Argument:
name A quoted string that contains a path variable. (Examples: SAMPLE_DATA, SAMPLE_SCRIPTS)
Get Project(<name>)

Function:
Retrieves either the first project listed in the Project window or the project named.

Returns:
The project.

Argument:
name  An optional project name as a quoted string.

JMP Version()

Function:
Returns the version number of JMP that you are running.

Returns:
release.revision<.fix>

Arguments:
none

Pick Directory(<"prompt string">)

Function:
Prompts the user for a directory, returning the directory path as a string. If a quoted string is provided, that string is included in the window.

Returns:
The path for the directory that the user selects.

Arguments:
prompt  An optional quoted string. If provided, that string is used for the window title.

Pick File(<caption, initial directory, {filter list}>)

Function:
Prompts the user to select a file in the Open File window.

Returns:
The path of the file that the user selects.

Arguments:
caption  An optional quoted string. If provided, that string is used for the window title, providing a prompt for the user.
initial directory  An optional quoted string that is a valid filepath to a folder. If provided, it specifies where the Open window begins. If not provided, or if it’s an empty string, the JMP Default Directory is used.
filter list  An optional list of quoted strings that define the filetypes to show in the Open window. See the following example for syntax.

Note:
You must use either no arguments, resulting in a standard Open File window, or all three arguments.

Example:
Pick File("Select JMP File", "$DOCUMENTS", {"JMP Files|jmp;jsl;jrn", "All Files|*"})

Set Current Directory( )
Function:
Sets the current directory for Open File operations.

Set Default Directory( )
Function:
Sets the default directory, which is used for resolving relative paths.

Set File Search Path({path or list of paths})
Function:
Sets the current list of directories to search for opening files. "." means the current directory.

Set Path Variable(name)
Function:
Sets the path stored in the variable.

Color Functions

Color To HLS(color)
Function:
Converts the color argument to a list of HLS values.

Returns:
A list of the hue, lightness, and saturation components of color. The values range between 0 and 1.

Argument:
color  a number from the JMP color index.

Example:
The output from ColorToHLS() can either be assigned to a single list variable or to a list of three scalar variables:

hls = colortohls(8);
{h, l, s} = colortohls(8);
show(hls, h, 1, s);

hls:{0.75, 0.529411764705883, 1}
h:0.75
l:0.529411764705883
Color To RGB(color)

Function:
Converts the color argument to a list of RGB values.

Returns:
A list of the red, green, and blue components of color. The values range between 0 and 1.

Argument:
color a number from the JMP color index.

Example:
The output from ColorToRGB() can either be assigned to a single list variable or to a list of three scalar variables:
rgb = colortorgb(8);
{r, g, b} = colortorgb(8);
show(rgb, r, g, b)
  rgb:{0.67843137254902, 0.247058823529412, 0.972549019607843}
  r:0.67843137254902
g:0.247058823529412
b:0.972549019607843

Heat Color(n, <theme>)

Function:
Returns the JMP color that corresponds to n in the color theme.

Returns:
An integer that is a JMP color.

Arguments:
n A number between 0 and 1.
theme Any color theme that is supported by Cell Plot. The default value is "Blue to Gray to Red".

HLS Color(h, l, s)
HLS Color({h, l, s})

Function:
Converts hue, lightness, and saturation values into a JMP color number.

Returns:
An integer that is a JMP color number.

Arguments:
Hue, lightness, and saturation, or a list containing the three HLS values. All values should be between 0 and 1.
RGB Color(r, g, b)
RGB Color({r, g, b})

Function:
Converts red, green, and blue values into a JMP color number.

Returns:
An integer that is a JMP color number.

Arguments:
Red, green, and blue, or a list containing the three RGB values. All values should be between 0 and 1.

Functions to Communicate with Users

Beep()

Function:
Produces an alert sound.

Returns:
Null.

Caption({h, v}, text, <Delayed(n)>, <Spoken(bool))

Function:
Displays a caption window at the location described by \( h, v \) that displays \( text \). The caption may optionally be delayed before being displayed by \( n \), or may optionally be spoken.

Returns:
Null.

Arguments:

\{h, v\} a list with two values. \( h \) is the horizontal displacement from the top left corner of the monitor in pixels. \( v \) is the vertical displacement from the top left corner in pixels.

\text A quoted string or a reference to a string that is to be displayed in the caption.

Delayed(n) \( n \) is optional delay before displaying the caption in seconds.

Spoken(bool) Causes \( text \) to be spoken as well as displayed.

Mail("host.id", "subject", "message", "attachment")

Function:
(Windows only) Sends email (using MAPI) to the \( host.id \) with the specified \( subject \) and \( message \) texts. Sends an attachment specified by the optional \( attachment \) parameter.

Print(expr, expr, ...)

Function:
Prints the values of the specified \( expressions \) to the log.
Save Log(pathname)

Function:
Writes the contents of the log to the specified file location.

Show(expr, expr, ...)

Function:
Prints the name/s and value/s of each expression to the log.

Show Commands()

Function:
Lists scriptable objects and operators. Arguments are All, DisplayBoxes, Scriptables, Scriptable Objects, StatTerms, Translations.

Show Properties(object)

Function:
Shows the messages that the given object can interpret, along with some basic syntax information.

Site ID()

Function:
Returns the user's annual license Site ID.

Speak(text, <wait(bool)>)

Function:
Calls system's speech facilities to read aloud the text. If Wait is turned on, script execution pauses until speaking is done.

Status Msg("message")

Function:
Writes the message string to the status bar.

Web(string, <JMP Window>)

Function:
Opens the URL stored in string in the default system web browser. Under Microsoft Windows, you can optionally add JMP Window as the second argument to have the HTML open in a JMP browser.
Window(<string|int>)

Returns:
Either a list of references to all open windows, or a reference to an explicitly named window.

Arguments:
  string A quoted string containing the name of a specific open window.
  int the number of a specific open window.

Write("text")

Function:
Prints text to the log without surrounding quotation marks.

Functions to Control JMP

Clear Log()

Function:
Empties the log.

Exit()

Quit()

Function:
Exits JMP.

Returns:
N/A

Arguments:
None.

Main Menu(string, <string>)

Function:
Execute the command found on JMP's menu named by the quoted string. The optional second
quoted string specifies the name of the window to send the command to.

Tick Seconds()

Function:
Measures the time taken for a script to run, measured down to the 60th of a second.

Wait(n)

Function:
Pauses n seconds before continuing the script.
Menu and Preference Functions

Revert Menu()

Function:
Resets your JMP menus to factory defaults.

Set Menu(path)

Function:
Sets your JMP menus to the JMPMENU file specified by the quoted string path.

Preferences(pref1(value1), ...)
Preference(pref1(value1), ...)
Pref(pref1(value1), ...)

Function:
Sets preferences for JMP.

Arguments:
Analysis Destination(window) Specifies where to route new analyses.
Background Color(color) Sets the background color for windows.
Calculator Boxing(boolean) Turns on boxing to show hierarchy of expressions.
DataTable Title on Output(boolean) Titles reports with name of data table.
Date Title on Output(boolean) Titles reports with current date.
Excel Has Labels(boolean) When on, forces JMP to interpret the first row of data as column labels.
Excel Selection(boolean) When on, the user is prompted for which non-blank Excel worksheets should be imported from an Excel workbook.
Foreground Color(color) Sets the foreground color for windows.
Graph Background Color(color) Sets the color for the background area inside the graph frame.
Graph Marker Size(size) Default size for drawing markers.
Heading Font(font, size, style) Font choice for table column headings in reports.
Initial JMP Starter Window(boolean) Specifies whether the JMP Starter window is shown at launch.
Initial Splash Window(boolean) Allows you to show or suppress the initial splash screen.
Monospaced Font(font, size, style) Font choice for monospaced text.
Outline Connecting Lines(boolean) Draws lines between titles for same-level outline nodes.
Report Table Style(style) Specify how columns in report tables are bordered. Choices are Plain, Bordered, Embossed.
Functions, Variables, and Other Basic Tools

Save Table In Extended File Format(boolean) Enables saving data tables with more than 32,766 columns.
Show Explanations(boolean) Some analyses have optional text that explains the output.
Show Menu Tips(boolean) Turns menu tips on or off.
Show Status Bar(boolean) Turns display of the status bar on or off.
Small Font(font, size, style) Font choice for small text.
Text Font((font, size, style) Font choice for general text in reports.
Thin Postscript Lines(boolean) Macintosh only. Specifies that line widths drawn to a Postscript printer be narrower than otherwise.
Title Font(font, size, style) Font choice for titles. Arguments are name of font (e.g., Times), size in points, and style (bold, plain, underline, italic).

Platform Preferences(platform(option(value)), ...) Function:
Sets preferences for platforms.

Show Preferences(<all>)
Function:
Shows current preferences. If no argument is specified, preferences that have been changed are shown. If all is given as the argument, all preferences are shown.

Project Functions

Get Project(<name>)
Function:
Retrieves either the first project listed in the Project window or the project named.
Returns:
The project.
Argument:
name An optional project name as a quoted string.

New Project(name)
Function:
Creates a new project with the specified name.
Returns:
The project.
Argument:
name A quoted string that will be the name of the new project.
Open Project(pathname)

Function:
   Opens the project identified by the quoted string pathname.

Returns:
   The project.

Arguments:
   pathname A pathname that points to a project file.

Loop Controls and Conditional Statements

And()

See
   “And(a, b),” p. 469

Break()

Function:
   Stops execution of a loop completely and continues to the statement following the loop.

Note:
   Break works with For and While loops, and also with For Each Row.

Choose(expr, r1, r2, r3, ..., rElse)

Function:
   Evaluates expr. If the value of expr is 1, r1 is returned; if 2, the value of r2 is returned, etc. If no matches are found, the last argument (rElse) is returned.

Returns:
   The value whose index in the list of parameters matches expr, or the value of the last parameter.

Arguments:
   expr an expression or a value.
   r1, r2, etc. an expression or a value.

Continue()

Function:
   Ends the current iteration of a loop and begins the loop at the next iteration.

Note:
   Continue works with For and While loops, and also with For Each Row.
For(init, while, increment, body)

Function:
Repeats the statement(s) in the body as long as the while condition is true. Init and increment control iterations.

Returns:
Null.

Arguments:
init Initialization of loop control counter.
while Condition for loop to continue/end. As long as the conditional statement while is true, the loop is iterated one more time. As soon as while is false, the loop is exited.
increment Increments (or decrements) the loop counter after while is evaluated every time the loop is executed.
body Any number of valid JSL expressions, glued together if there are more than one.

Example:
mysum=0;myprod=1;
for(i=1, i<=10, i++, mysum+=i;myprod*=i);
show(mysum, myprod);

mysum:55
myprod:3628800

If(condition, result, condition, ..., <elseResult>)

Function:
Returns result when condition evaluates true. Otherwise returns next result when that condition evaluates as true. The optional elseResult is used if none of the preceding conditions are true. If no elseResult is given, and none of the conditions are true, then nothing happens.

IfMZ(condition, result, condition, ..., <elseResult>)
IfV3(condition, result, condition, ..., <elseResult>)

Function:
Logical If with version 3.x behavior, which treats missing values as 0; used automatically when opening v3 data tables.

Interpolate(x, x1, y1, x2, y2)
Interpolate(x, xmatrix, ymatrix)

Function:
Linearly interpolates the y-value corresponding to a given x-value between two points (x1, y1), and (x2, y2) or by matrices xmatrix and ymatrix. The points must be in ascending order.
Match(a, value1, result1, value2, result2, ...)
Function:
If `a` is equal to `value1`, then `result1` is returned; if `a` is equal to `value2`, `result2` is returned, etc.

MatchMZ(a, value1, result1, value2, result2, ...)
MatchV3(a, value1, result1, value2, result2, ...)
Function:
Match with version 3.x behavior, which treats missing values as 0; used automatically when opening v3 data tables.

Not()
See
“Not(a),” p. 473

Or()
See
“Or(a, b),” p. 474

Step(x, x1, y1, x2, y2, ...)
Function:
Finds corresponding `y` for a given `x` from a step-function fit. The points must be in ascending order.

While(expr, body)
Function:
Repeatedly tests the `expr` condition and executes the `body` until the `expr` is no longer true.

Lists

List(a, b, c, ...)
{a, b, c, ...}
Function:
Constructs a list from a set of items.
N Items(source)

Function:
Determines the number of elements in the source specified. Can be assigned to a variable.

Returns:
For a list, the number of items in the list. For an associative array, the number of keys.

Arguments:
source A list or an associative array.

Sort Ascending(source)

Function:
Returns a copy of a list or matrix source with the items in ascending order.

Sort Descending(source)

Function:
Returns a copy of a list or matrix source with the items in descending order.

Sort List(list|expr)

Function:
Sort the elements or terms of list or expr.

Sort List Into(list|expr)

Function:
Sort the elements or terms of list or expr in place.

Subscript(a, b, c)
list[i]
matrix[b, c]

Function:
Subscripts for lists extract the ith item from the list, or the bth row and the cth column from a matrix.

Matrices

All(A...)

Returns:
1 if all arguments are nonzero; 0 otherwise.
Any(A...)  
**Returns:**
1 if one or more elements of the matrix are nonzero; 0 otherwise.

---

**As List(matrix)**

**Function:**
Converts a matrix into a list. Multi-column matrices are converted to a list of row lists.

**Returns:**
A list.

**Argument:**
matrix Any matrix.

---

**As Table(A)**

**See:**
“As Table(A),” p. 512.

---

**Cholesky(A)**

**Function:**
Finds the lower Cholesky root (L) of a symmetric matrix. \( L \cdot L' = A \).

**Returns:**
L (the Cholesky root).

**Arguments:**
A: a symmetric matrix.

---

**Chol Update(L, X, Sym)**

**Function:**
If \( L \) is the cholesky root of an \( m \times n \) matrix \( A \), then after calling `cholUpdate` \( L \) will be replaced with the cholesky root of \( A + V \cdot C \cdot V' \) where \( C \) is an \( m \times m \) symmetric matrix and \( V \) is an \( n \times m \) matrix.

---

**Concat(A, B)**

A||B

**Function:**
Horizontal concatenation of two matrices.

---

**Concat To(A, B)**

A||=B

**Function:**
Horizontal concatenation of two matrices in place. This is an assignment operator.
Concat Items\{string1, string2, \ldots, <delimiter>\}

**Function:**
Converting a list of string expressions into one string, with each item separated by a delimiter. The delimiter is a blank, if unspecified.

**Cumulant Quadrature(vector)**

**Function:**
Returns the optimal quadrature abscesses and weights.

**Argument:**
vector A vector of cumulants (1 by m) between 0 and m-1 for a distribution.

**Design(vector)**

**Function:**
Creates design columns for a vector of values.

**Returns:**
A design matrix.

**Argument:**
vector A vector.

**Design Nom(vector)**

**DesignF(vector)**

**Function:**
A version of Design for making full-rank versions of design matrices for nominal effects.

**Returns:**
A full-rank design matrix.

**Argument:**
vector A vector.

**Design Ord(vector)**

**Function:**
A version of Design for making full-rank versions of design matrices for ordinal effects.

**Returns:**
A full-rank design matrix.

**Argument:**
vector A vector.
## Det(A)

**Function:**
Determinant of a square matrix.

**Returns:**
The determinant.

**Argument:**
det A square matrix.

## Diag(A, <B>)

**Function:**
Creates a diagonal matrix from a square matrix or a vector. If two matrices are provided, concatenates the matrices diagonally.

**Returns:**
The matrix.

**Argument:**
A a matrix or a vector.

## Direct Product(A, B)

**Function:**
Direct (Kronecker) product of square matrices or scalars: A[ij]*B.

**Returns:**
The product.

**Arguments:**
A, B Square matrices or scalars.

## Distance(x1, x2, <scales>, <powers>)

**Function:**
Produces a matrix of distances between rows of x1 and rows of x2.

**Returns:**
A matrix.

**Arguments:**
x1, x2 Two matrices.
scales Optional argument to customize the scaling of the matrix.
powers Optional argument to customize the powers of the matrix.
E Div(A, B)
A/B
Function:
Element-by-element division of two matrices.
Returns:
The resulting matrix.
Arguments:
A, B Two matrices.

Eigen(A)
Function:
Eigenvalue decomposition.
Returns:
A list \{M, E\} such that \( E \cdot \text{Diag}(M) \cdot E = A' \).
Argument:
A A symmetric matrix.

E Mult(A, B)
A*B
Function:
Element-by-element multiplication of two matrices.
Returns:
The resulting matrix.
Arguments:
A, B Two matrices.

G Inverse(A)
Function:
Generalized (Moore-Penrose) matrix inverse.

H Direct Product(A, B)
Function:
Horizontal direct product of two square matrices of the same dimension or scalars.
Identity(n)

Function:
Creates an \( n \)-by-\( n \) identity matrix with ones on the diagonal and zeroes elsewhere.

Returns:
The matrix.

Argument:
\( n \) An integer.

Index(i, j)
i::j

Function:
Creates a column matrix whose values range from \( i \) to \( j \).

Returns:
The matrix.

J(nrows, <ncols>, <value>)

Function:
Creates a matrix of identical values.

Returns:
The matrix.

Arguments:
\( \text{nrows} \) Number of rows in matrix. If \text{ncols} is not specified, \text{nrows} is also used as \text{ncols}.
\( \text{ncols} \) Number of columns in matrix.
\( \text{value} \) The value used to populate the matrix. If \text{value} is not specified, 1 is used.

Matrix({{x11, x12, ..., x1m}, {x21, x22, ..., 2m}, {...}, {xn1, xn2, ..., xnm}})

Function:
Constructs an \( n \)-by-\( m \) matrix from a list of \( n \) lists of \( m \) row values.

Returns:
The matrix.

Arguments:
A list of lists in which each list forms a row with the specified values.

Example:
\[
\text{mymatrix} = \text{matrix}([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]);
\]
\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{bmatrix}
\]
Equivalent Expression:

\[
[x_{11} \ x_{12} \ ... \ x_{1m}, \\
... \\
x_{n1} \ x_{n2} \ ... \ x_{nm}]
\]

**Moment Quadrature(vector)**

**Function:**

Returns the optimal quadrature abscissas and weights.

**Argument:**

- `vector` A vector of moments (1 by m) between 0 and m-1 for a distribution.

---

**Print Matrix(M, <named arguments>)**

**Function:**

Prints a well-formatted matrix to the log.

**Argument:**

- `M` A matrix.

**Named Arguments:**

Note that the following named arguments are all optional.

- `<<ignore locale(boolean)` Set to false (0) to use the decimal separator for your locale. Set to true (1) to always use a period (.) as a separator. The default value is false (0).

- `<<decimal digits(n)` An integer that specifies the number of digits after the decimal separator to print.

- `<<style("style name")` Use one of three available styles: Parseable is a reformatted JSL matrix expression. Latex is formatted for LaTex. If you specify Other, you must define the following three arguments.

- `<<separate("character")` Define the separator for concatenated entries.

- `<<line begin("character")` Define the beginning line character.

- `<<line end("character")` Define the ending line character.

---

**Random Index(n, k)**

**Function:**

Returns a `k` by 1 matrix of random integers between 1 and `n` with no duplicates.

---

**Random Shuffle(matrix)**

**Function:**

Returns the matrix with the elements shuffled into a random order.
Shape(A, nrow, <ncol>)

Function:
Reshapes the matrix A across rows to the specified dimensions. Each value from the matrix A is placed row-by-row into the re-shaped matrix.

Returns:
The reshaped matrix.

Arguments:
A a matrix
nrow the number of rows that the new matrix should have.
ncol optional. The number of columns the new matrix should have.

Notes:
If ncol is not specified, the number of columns is whatever is necessary to fit all of the original values of the matrix into the reshaped matrix.
If the new matrix is smaller than the original matrix, the extra values are discarded.
If the new matrix is larger than the original matrix, the values are repeated to fill the new matrix.

Examples:
a = matrix({{1, 2, 3}, {4, 5, 6}});
[{1 2 3,
  4 5 6,
  7 8 9}]
shape(a, 2);
[{1 2 3 4 5,
   6 7 8 9 1}]
shape(a, 2, 2);
[{1 2,
    3 4}]
shape(a, 4, 4);
[{1 2 3 4,
   5 6 7 8,
   9 1 2 3,
   4 5 6 7}]

Subscript(a, b, c)
matrix[b, c]

See:
“Subscript(a, b, c),” p. 497
**Transpose(A)**

Function:
- Transposes the rows and columns of the matrix \( A \).

Returns:
- The transposed matrix.

Arguments:
- \( A \) A matrix.

Equivalent Expression:
- \( A' \)

**V Concat(A, B, ...)**

Function:
- Vertical concatenation of two or more matrices.

Returns:
- The new matrix.

Arguments:
- Two or more matrices.

**V Concat To(A, B, ...)**

Function:
- Vertical concatenation in place. This is an assignment operator.

Returns:
- The new matrix.

Arguments:
- Two or more matrices.

**Vec Diag(A)**

Function:
- Creates a vector from the diagonals of a square matrix \( A \).

Returns:
- The new matrix.

Arguments:
- A square matrix.

Note:
- Using a matrix that is not square results in an error.
Vec Quadratic(symmetric matrix, rectangular matrix)

Function:
Constructs an $n$-by-$m$ matrix. Used in calculation of hat values.

Returns:
The new matrix.

Arguments:
Two matrices. The first must be symmetric.

Equivalent Expression:
Vec Diag(X*Sym*X')

Loc(A)

Function:
Creates a matrix of subscript positions where $A$ is nonzero and non-missing.

Returns:
The new matrix.

Argument:
A a matrix

Loc Min(A)

Function:
Returns the position of the minimum element in a matrix.

Returns:
An integer that is the specified position.

Argument:
A a matrix

Loc Max(A)

Function:
Returns the position of the minimum element in a matrix.

Returns:
An integer that is the specified position.

Argument:
A a matrix
Loc Sorted(A, B)

Function:
Creates a matrix of subscript positions where the values of A have values less than or equal to the values in B. A must be a matrix sorted in ascending order.

Returns:
The new matrix.

Argument:
A, B matrices

Matrix Mult(A, B)

C=A*B, ...

Function:
Matrix multiplication.

Arguments:
Two or more matrices, which must be conformable (all matrices after the first one listed must have the same number of rows as the number of columns in the first matrix).

Note:
Matrix Mult() allows only two arguments, while using the * operator allows you to multiply several matrices.

Maximum(A, <B, ...)>
Max(A, <B, ...>)

Function:
R maximum element inside the matrix or matrices.

See
“Maximum(var1, var2, ...).” p. 573

Minimum(A, <B, ...>)
Min(A, <B, ...>)

Function:
Minimum element inside the matrix or matrices.

See
“Minimum(var1, var2, ...).” p. 573

N Col(A)
N Cols(A)

Function:
Returns the number of columns in the matrix A.
N Row(A)
N Rows(A)

Function:
Returns the number of rows in the matrix A.

Rank Index(vector)
Rank(vector)

Function:
Returns a vector of indices that, used as a subscript to the original vector vector, sorts the vector by rank. Excludes missing values.

Ranking(vector)

Function:
Returns a vector of ranks of the values of vector, low to high as 1 to n, ties arbitrary.

Ranking Tie(vector)

Function:
Returns a vector of ranks of the values of vector, but ranks for ties are averaged.

Trace(A)

Function:
The trace, or the sum of the diagonal elements of a square matrix.

Inverse(A)
Inv(A)

Function:
Returns the matrix inverse. The matrix must be square non-singular.

Inv Update(A, X, 1|-1)

Function:
Efficiently update an X’X matrix. X is the row to add. In the third parameter, 1 adds a row, -1 deletes a row.

Ortho(A, <Centered(0)>, <Scaled(1)>)

Function:
Orthonormalizes the columns of matrix A using the Gram Schmidt method. Centered(0) makes the columns to sum to zero. Scaled(1) makes them unit length.
Ortho Poly(vector, order)
Function:
Returns orthogonal polynomials for a vector of indices representing spacings up to the order given.

QR(A)
Function:
Returns the QR decomposition of A. Typical usage is \{Q, R\} = QR(A).

Solve(A, b)
Function:
Solves a linear system, i.e. \(x = \text{inverse}(A) * b\).

SVD(A)
Function:
Singular value decomposition.

Sweep(A, <indices>)
Function:
Sweeps, or inverts a matrix a partition at a time.

V Max(matrix)
Function:
Returns a row vector containing the maximum of each column of \(matrix\).

V Mean(matrix)
Function:
Returns a row vector containing the mean of each column of \(matrix\).

V Min(matrix)
Function:
Returns a row vector containing the minimum of each column of \(matrix\).

V Sum(matrix)
Function:
Returns a row vector containing the sum of each column of \(matrix\).

V Std(matrix)
Function:
Returns a row vector containing the standard deviations of each column of \(matrix\).
Associative Arrays

Associative Array({key, value}, ...)  
Associative Array(keys, values)

Function:  
Creates an associative array (also known as a dictionary or hash map).

Returns:  
An associative array object.

Arguments:  
Either list of key/value pairs; or a list, matrix, or data table column that contains keys followed by a list, matrix, or data table column that contains the corresponding values.

Contains()

See  
Using Contains for lists or strings on p. 460.

Insert(source, key, value)  
Insert Into(source, key, value)

Function:  
Adds the key into the source associative array and assigns value to it. If the key exists in source already, its value is overwritten with the new value.

Arguments:  
source  An associative array.
key  A key that may or may not be present in source.
value  A value to assign to the key.

Note:  
Insert Into performs the Insert function in place. For Insert Into, source must be an L-value.

See also  
Using Insert (p. 461) and Insert Into (p. 461) for lists, strings, or expressions.

N Items()

See  
Using N Items for lists or strings on p. 497.
Remove(source, key)

Remove From(source, key)

**Function:**
Deletes the key and its value.

**Returns:**
A copy of the source with the items deleted.

**Arguments:**
- source An associative array.
- key A key that is contained within source.

**Note:**
Remove From performs the Remove function in place. For Remove From, source must be an L-value.

**See also**
Using Remove (p. 463) and Remove From (p. 463) for lists, strings, or expressions.

**Messages for Associative Arrays**
For the following messages, map stands for an associative array or a reference to one.

**map << Insert(key, value)**
Inserts the key into map and assigns value to it. If key already exists in map, its value is replaced by the new value given. This message is equivalent to the function Insert Into.

**map << Get Value(key)**
Returns the value for the key within map.

**map << Remove(key)**
Removes the key and value from map. This message is equivalent to the function Remove From.

**map << Get Keys**
Returns a list of all the keys within map.

**map << Get Contents**
Returns a list of all key-value pairs within map.

**map << Get Values(<keylist>)**
If no argument is provided, a list of all values within map is returned.
If a list of keys is provided, a list of the values corresponding to only those keys is returned.
map << First

Returns the first key within map, or Empty() if map has no keys. Note that keys are returned in lexicographical order.

map << Next(key)

Returns the key following the given key within the map, or Empty() if map has no keys. Note that keys are returned in lexicographical order.

map << Set Default Value(v)

Sets the implicit value of all absent keys. Any key added without a value is assigned this value by default.

map << Get Default Value()

Returns the implicit value of all absent keys, or Empty() if none has been set.

---

Data Tables, Rows, and Columns

This section gives abbreviated descriptions for functions related to Data Tables.

Data Tables

As Table(A)

Function:
  Creates a new data tables from matrix A.

Returns:
  The new data table.

Argument:
  A Any matrix.

Close(<dt>, <nosave|save("path")>)

Function:
  Closes a data table. If no arguments are specified, the current data table is closed.

Returns:
  N/A

Arguments:
  dt  an optional reference to a data table.
Data Tables, Rows, and Columns

nosave|save("path")  an optional switch to either save the data table to the specified path
before closing or to close the data table without saving it. If the table has been changed and
neither nosave nor save has been supplied, you will be prompted to save or disregard changes
before the data table is closed.

Close All(type, <NoSave|Save>)

Function:
Closes all open resources of type.

Argument:
type  A named argument that defines the type of resources you want to close. The allowable types
are: Data Tables, Reports, and Journals.
NoSave or Save  An optional argument that specifies to close all the windows of type either
saving all of them or not saving any of them. No prompts appear if you use one of these two
arguments. Otherwise, you will be prompted to save any unsaved windows that are to be closed.

Current Data Table(<dt>)

See:
"Current Data Table(<dt>)," p. 480.

Data Table(n)
Data Table("name")

Function:
Gets reference to the n'th open data table or the table with the given name in a global variable.

Returns:
Reference to the specified data table.

Argument:
n  Number of a data table.
name  Quoted string, name of a data table.

N Col(x)
N Cols(x)

Function:
Returns the number of columns in either a data table or a matrix.

Argument:
x  Can be a data table or a matrix.
New Column("name", attributes)

Function:
   Adds a new column named name after the last column in dt.

Note:
   Can also be used as a message: dt<<New Column("name", attributes).

New Table("name")

Function:
   Creates a new data table with the specified name.

N Table()

Function:
   Returns the number of open data tables.

Open("path", <options>)

Function:
   Opens the data table named by the path.

Options:
   Strip Quotes(bool), End of Line(choice), EOLOther("char"), End of Field(choice), EOFOther("char"), Labels(bool)

Suppress Formula Eval(bool)

Function:
   Turns off automatic calculation of formulas for all data tables.

Messages for Data Tables

dt<<Add Multiple Columns("prefix", n, position, attributes)

   Adds n columns to dt at the position indicated.

dt<<Begin Data Update

   Turns off display updating to allow for quick updating of a data table. Use End Data Update in conjunction with this command to turn display updating back on.
**dt<<Data Filter(<Mode(...>), <Add Filter (...)>)**

Constructs a data filter. If no arguments are specified, the Add Filter Columns window appears. Arguments for `Mode()` include `Select()`, `Show()`, and `Include()`. They are all boolean. `Select` defaults to true (1), and `Show` and `Include` default to false (0).

Arguments for `Add Filter()` include `Columns()`, and `Where()`. `Columns()` takes one or more column names separated by commas. You can add one or more `Where` clauses to define the filter. There are several additional arguments. For more information, see the Data Filter chapter in the *JMP User Guide* and the Data Tables chapter in the *JMP Scripting Guide*.

**dt<<Delete Columns(col, col2, ...)**

**dt<<Delete Column**

Deletes column(s) from the data table `dt`. Specify which column or columns to delete. Without an argument, deletes the selected columns, if any. **Delete Column** is a synonym.

**dt<<End Data Update**

Resumes display updating after a `Begin Data Update` message. These commands are used for quick updates of the data table when many changes have to be made. Speed is gained by turning off display updating.

**dt<<Get Name()**

Returns the name of the table.

**dt<<Get Column Names(arguments)**

Returns a list of column names in a data table. The `arguments` restrict the names retrieved as follows: `Numeric`, `Ordinal`, `Rowstate`, `Continuous`, `Ordinal`, and `Nominal` get only the specified types of columns. More than one can be specified. `String` returns a list of strings rather than a list of column references.

**Get Selected Columns()**

Returns a list of selected columns.

**dt<<Maximize Display**

Forces the data table to remeasure all of its columns and zoom to the best-sized window.

**dt<<New Column("name", attributes)**

Adds a new column named `name` after the last column in `dt`. Can also be used as a command: **New Column("name", attributes)***
dt<<Original Order
Restores saved order of columns in \(dt\).

dt<<Print Window
Prints the data table

dt<<Reorder By Data Type
Reorders columns in \(dt\), row state first, then character, then numeric.

dt<<Reorder By Modeling Type
Reorders columns in \(dt\) to continuous, then ordinal, then nominal.

dt<<Reorder By Name
Reorders columns in \(dt\) to alphanumeric order by name.

dt<<Reverse Order
Reverses columns in \(dt\) from current order.

dt<<Revert
Reverts to the most recently saved version of \(dt\).

dt<<Run Script("name")
Finds the table property \(name\) and runs it as a JSL script.

dt<<Save("path")
Saves the table under the \(path\) given.

dt<<Select Rows({list})
Selects the rows given in the list of row numbers.

dt<<Select Randomly(p)
Randomly selects the given percentage \(p\) of the rows in the data table.

dt<<Set Name("name")
Gives a \(name\) to the table.
```julia
\texttt{dt<<Suppress Formula Eval(bool)}

Turns off automatic calculation of formulas for data table \texttt{dt}.

\texttt{dt<<Delete Table Variable(name)}

Delete a table variable.

\texttt{dt<<Delete Table Property(name)}

Delete a table property.

\texttt{dt<<Set Row States(matrix)}

Assigns row states to each row of a data table. \texttt{matrix} contains the codes corresponding to the desired state and contains one entry per row.

\texttt{dt<<Clear Row States}

Cancels any row states in effect.

\texttt{dt<<Get As Matrix}

Returns values from the numeric columns of \texttt{dt} in a matrix.

\texttt{dt<<Get All Columns As Matrix}

Returns the values from all columns of \texttt{dt} in a matrix. Character columns are numbered according to the levels, starting at 1.

\texttt{dt<<Get Property("name")}

Returns the \texttt{script} in the property \texttt{name}.

\texttt{dt<<Get Script}

Returns as an expression a script to recreate the data table.

\texttt{dt<<Get Table Variable("name")}

Returns the \texttt{value} from the variable \texttt{name}.

\texttt{dt<<New Table Property("name", script)}

Creates a new table property \texttt{name} that stores \texttt{script}.```
dt<<New Table Variable("name", value)

Creates a new table variable with the indicated name and value.

dt<<Save Script to Script Window

Saves a script to reproduce the data table in a Script Journal window.

dt<<Set Matrix(matrix)

Inserts matrix into a data table, adding new columns and rows as necessary.

dt<<Set Property("name", script)

Stores the script in the property name.

dt<<Set Table Variable("name", value)

Stores the value in the variable name.

dt << Set Label Columns("name", ...)

dt << Set Scroll Lock Columns("name", ...)

Turns on the Label attribute for the specified columns. If no columns are listed, turns Label attribute off.

Locks scrolling for the specified columns. If no columns are listed, unlocks scrolling.

dt<<Clear Select

Turns off the current selection.

dt<<Color by Column(col)

Assigns colors according to the values of a data table column.

dt<<Colors(n)

Assigns color n to the selected rows.

dt<<Marker by Column(col)

Assigns markers according to the values of a data table column.
**dt<<Markers(n)**
Assigns marker *n* to the selected rows.

**dt<<Next Selected**
Scrolls data table down to show the next selected row that isn’t already in view.

**dt<<Previous Selected**
Scrolls data table up to show the previous selected row that isn’t already in view.

**dt<<Exclude**
**dt<<Unexclude**
Toggles selected rows in *dt* from excluded to unexcluded or vice versa.

**dt<<Hide**
**dt<<Unhide**
Toggles selected rows in *dt* from hidden to unhidden or vice versa.

**dt<<Label**
**dt<<Unlabel**
Toggles selected rows in *dt* from labeled to unlabeled or vice versa.

**dt<<Add Rows(count, <rownum>)**
**dt<<Add Rows(assignment list)**
Add the number of rows specified to the bottom of the data table or starting at the *rownum* specified.

**dt<<Move Rows(At Start)**
**dt<<Move Rows(At End)**
**dt<<Move Rows(After(n))**
Moves selected rows in *dt* so that they are At Start, At End, or After(rownumber).

**dt<<Delete Rows(<n>)**
**dt<<Delete Rows({n, o, p, ...})**
**dt<<Delete Rows({n::q})**
Deletes the currently selected rows or rows specified.
dt<<Go To Row(n)
Locates and selects row number n in dt.

dt<<Invert Row Selection
Selects any row currently unselected and deselects any row currently selected.

dt<<Select All Rows
Selects all rows in the data table.

dt<<Select Excluded
Selects only those rows in the data table that are currently excluded.

dt<<Select Hidden
Selects only those rows in the data table that are currently hidden.

dt<<Select Labeled
Selects only those rows in the data table that are currently labeled.

dt<<Select Where(condition)
Selects the rows in dt where the condition evaluates as true.

dt<<Concatenate(dt2, ..., Keep Formulas, Output Table Name("name"))
Creates a new table (name) from the rows of dt and dt2.

dt<<Join(With(table), Select(columns), Select With(columns), method, Drop Multiples(bool, bool), Include Non Matches(bool, bool), Output Table Name("name"))
Combines data tables dt and table side to side.

dt<<Sort(By(columns), order(Descending or Ascending), Output Table Name("name"))
Creates a new table (name) by rearranging the rows of dt according to the values of one or more columns.
**Data Tables, Rows, and Columns**

`dt<<Split(Split(columns), Group(gcol), Col ID(idcol), Remaining Columns(choice), Output Table Name("name"))`

Creates a new table *(name)* by breaking one or more *columns* from *dt* into several.

`dt<<Stack(Stack(columns), ID(columns), Stacked(newcol), Output Table Name("name"))`

Creates a new table *(name)* by combining the values from several columns in *dt* into one column *newcol*.

`dt<<Subset(Columns(columns), Rows(matrix), Linked, Table Name("name"), Copy Formula(1|0), Suppress Formula Evaluation(1|0), Sampling Rate(rate))`

Creates a new table *(name)* from the rows you specify in *dt*.

`dt<<Summary(Group(col), Subgroup(col), statistic(col))`

Creates a new table of summary *statistics* for the *col* you specify, optionally according to groups and subgroups.

`dt<<Transpose(Columns(columns), Rows(matrix), Output Table Name("name"))`

Creates a new table *(name)* from the rows and columns you specify.

### Rows

**Count(from, to, step, times)**

Function:

Used for column formulas. Creates row by row the values beginning with the *from* value and ending with the *to* value. The number of *steps* specifies the number of values in the list between and including the *from* and *to* values. Each value determined by the first three arguments of the count function occurs consecutively the number of times you specify. When the *to* value is reached, *count* starts over at the *from* value. If the *from* and *to* arguments are data table column names, *count* takes the values from the first row only. Values in subsequent rows are ignored.

Returns:

The last value.

Arguments:

- *from* Number, column reference, or expression. *Count* starts counting with this value.
- *to* Number, column reference, or expression. *Count* stops counting with this value.
- *step* Number or expression. Specifies the number of steps to use to count between *from* and *to*, inclusive.
times  Number or expression. Specifies the number of times each value is repeated before the next step.

Examples:
  foreachrow(:colname[row()] = count(0, 6, 3, 1))  
  //The rows in the column named colname are filled with the series 0, 3, 6, 0, ... until all rows are filled.

  foreachrow(:colname[row()] = count(0, 6, 3, 2))  
  //The rows in the column named colname are filled with the series 0, 0, 3, 3, 6, 6, 0, ... until all rows are filled.

Note:
  Count() is dependent on Row(), and is therefore mainly useful in column formulas.

---

Dif(col, n)

Function:
  Calculates the difference of the value of the column col in the current row and the value n rows previous to the current row.

Returns:
  The difference.

Arguments:
  col  A column name (for example, :age).
  n    A number.

---

For Each Row(script)

Function:
  Repeats the script on each row of the data table.

Returns:
  Null.

Argument:
  script  Any valid JSL expressions.

---

Lag(col, n)

Function:
  Returns for each row the value of the column n rows previous.

---

N Row(dt); NRow(matrix)
N Rows(dt); NRows(matrix)

Function:
  Returns the number of rows in the data table given by dt or in the matrix.
Row()

Function:

Returns or sets the current row number. No argument is expected.

Sequence(from, to, stepsize, repeatTimes)

Function:

Produces an arithmetic sequence of numbers across the rows in a data table, where the from, to, and stepsize are specified. repeatTimes is optional.

Subscript()

See:

“Subscript(a, b, c),” p. 497

Rowstates

As Row State(i)

Function:

Converts i into a rowstate value.

Returns:

A row state from the i given.

Argument:

i an integer

Color Of(rowstate)

Function:

Returns or sets the color index.

Returns:

The color index of rowstate.

Argument:

rowstate a rowstate argument

Example:

colorof(rowstate(5))=3
sets color of the third row to red.
Color State(i)

Function:
   Returns a row state with the color index of \( i \).

Returns:
   A rowstate.

Argument:
   \( i \) index for a JMP color

Combine States(rowstate, rowstate, ...)

Function:
   Generates a row state combination from two or more rowstate arguments.

Returns:
   A single numeric representation of the combined rowstates.

Arguments:
   rowstate Two or more rowstates.

Excluded(rowstate)

Function:
   Returns or sets an excluded index.

Returns:
   The excluded attribute, 0 or 1.

Argument:
   rowstate Two or more rowstates.

Excluded State(num)

Function:
   Returns a row state for exclusion from the \( num \) given.

Hidden(rowstate)

Function:
   Returns or sets the hidden index.

Hidden State(num)

Function:
   Returns a row state for hiding from the \( num \) given.

Hue State(num)

Function:
   Returns a hue state from the \( num \) given.
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Labeled(rowstate)</td>
<td>Returns or sets the labeled index.</td>
</tr>
<tr>
<td>Labeled State(num)</td>
<td>Returns a labeled state from the num given.</td>
</tr>
<tr>
<td>Marker Of(rowstate)</td>
<td>Returns or sets the marker index of a rowstate.</td>
</tr>
<tr>
<td>Marker State(num)</td>
<td>Returns a marker state from the num given.</td>
</tr>
<tr>
<td>Row State(&lt;n&gt;)</td>
<td>Returns the active row state condition of the current row as true or false.</td>
</tr>
<tr>
<td>Selected(rowstate)</td>
<td>Returns or sets the selected index.</td>
</tr>
<tr>
<td>Selected State(num)</td>
<td>Returns a selected state from the num given.</td>
</tr>
<tr>
<td>Shade State(num)</td>
<td>The Shade State function assigns 5 shade levels to a color or hue.</td>
</tr>
</tbody>
</table>
Columns

Column(<dt>, name)
Column(<dt>, n)

Function:
Gets a reference to the data table column.

Arguments:
  dt  Optional reference to a data table. If this is not supplied, the current data table is used.
  name  Quoted string that is the name of the column.
  n  Column number.

Column Name(n)

Function:
Determines the name of the column specified by number.

Returns:
The name of the \( n \)th column as an expression (not a string).

Argument:
  n  The number of a column.

N Col(dt)
N Cols(dt)

See:
“N Col(x),” p. 513.

New Column("name", attributes)

See:

Subscripting for Columns

\( \text{colName}[n] \)

Function:
Returns the value of \( \text{colName} \) in row \( n \).

Arguments:
  colname  Not an argument, but the name of a column in an open data table.
  n  An integer that is a row number.

Example:
  age[5];
Messages for Columns

col<<Data Type(type)
    Sets the type for col; choices are Numeric, Character, Rowstate.

col<<Delete Formula
    Delete the formula from a column.

col<<Delete Property(name)
col<<Delete Column Property(name)
    Delete the property name from a column.

col<<Eval Formula
    Forces the formula to evaluate (perhaps again).

col<<Format("format", w, d)
    Sets the numeric display format specified.

col<<Formula(expression)
    Sets the formula for the variable and evaluates it.

col<<Get Data Type
    Returns the data type of col.

col<<Get Formula
    Returns the formula.

col<<Get Lock
    Returns the current Lock setting.

col<<Get Modeling Type
    Returns the modeling type of the column ("Continuous" or "Ordinal" or "Nominal").

col<<Get Name()
    Returns the name of the column.
col<<Get Format
Returns the format of the column.

col<<Get Role
Returns the preselected role of col.

col<<Preselect Role(role)
Preselects the role for the column. Choices are Y, X, Weight, Freq, and None.

col<<Set Modeling Type(type)
Sets the modeling type for the variable. Choices are Continuous, Ordinal, or Nominal.

col<<Set Name("name")
Sets the name for the column.

col<<Set Property("name", expression)
Sets the property name to the expression given. Choices for name are Notes, Axis, Coding, Mixture, Spec Limits, Control Limits, Response Limits, Design Role, Sigma, Units, or a user-specified property.

col<<Values([matrix] or {list})
Sets values from the matrix (for numeric variables) or list (for character variables).

Platforms
This section gives abbreviated descriptions for functions related to statistical and graphing platforms.
All Platforms

Platform(..., Title("string")
Function:
Sets the title of the resulting report to "string", overriding the default title. For example:
Bivariate (X(height), Y(weight), Title("my title"))

Platform(Action(expression))
Function:
Evaluates the expression when the platform is launched. Useful for producing several platform launches in a row.

Messages for All Platforms

obj << message
Send(obj, message)
   Sends a message to a platform object.

obj << message << message...
obj << {message, message, ...}
   Sends a series of messages (in order, left to right) to the platform object.

obj << (child << message)
   Sends a message to a child object within a platform object.

obj<<Close Window
   Closes window identified by obj, typically a platform surface.

obj<<Data Table Window
   Makes the associated data table window active (front-most).

obj<<Journal Window
   Appends the contents of the window to the journal.

obj<<Move Window(x, y)
   Moves the window x pixels to the right and y pixels down.
obj<<Redo Analysis

Launches the platform again with the same options.

obj<<Report

Report(obj)

Returns a display box reference for the report in the platform window.

obj<<Save Script for All Objects

Saves script to reproduce all analyses found within the object’s window in the Script Journal window.

obj<<Save Script to Datatable

Saves script to reproduce analysis as a property in the associated data table.

obj<<Save Script to Report

Saves script to reproduce analysis as a text box at the top of the report.

obj<<Save Script to Script Window

Saves a script to reproduce analysis in the Script Journal.

obj<<Size Window(x, y)

Resizes the window to \( x \) pixels wide by \( y \) pixels high.

obj<<Zoom Window

Resizes the window to the maximum size of its contents.

obj<<Minimize Window

Minimizes the window. Equivalent to pushing the minimize button in the corner of the window.

obj<<Maximize Window

Maximizes the window. Equivalent to pushing the maximize button in the corner of the window.

obj<<Scroll Window(x, y)

obj<<Scroll Window({x, y})

Scrolls the window \( x \) pixels to the left and \( y \) pixels down from the current position.
obj<<Bring Window To Front

Brings the current window to the front.

obj<<Get Script

Returns script to reproduce the analysis as an expression.

obj<<Get Window Size

Gets the window size, in pixels. Returns an ordered pair.

obj<<Get Window Position

Gets the position of the window. Returns an ordered pair.

obj<<Show Window

Shows a hidden window.

obj<<Print Window

Sends the selected window to the printer.

Windows, Reports, and Dialogs

This section gives abbreviated descriptions for functions related to report windows and interactive dialogs.

Display Boxes

Border Box(<Left(pix)>, <Right(pix)>, <Top(Pix)>, <Bottom(Pix)>, <Sides(0)>, db)

Function:

Constructs a display box that holds contains another display box. Optional arguments (Left, Right, Top, Bottom) add space between the border box and what it contains. The other optional argument (Sides) draws borders around the border box on any single side or combination of sides; draws the
border in black or the highlight color; makes the background transparent or white; erases the
background of a display box that contains it; draws the border in an embossed style or flat.

Returns:
The display box.

Arguments:

Left  An integer that measures pixels.
Right  An integer that measures pixels.
Top  An integer that measures pixels.
Bottom  An integer that measures pixels.
Sides  An integer that maps to settings for the display box.
db  a display box object (for example, a text box or another border box)

Notes:
The formula for deriving the integer for Sides is: 1*top + 2*left + 4*bottom + 8*right +
16*highlightcolor + 32*whitebackground + 64*erase + 128*emboss. Thus, if you want to just draw
a black border on the top and bottom, 1+4 = 5. If you want that same box with a white background,
5+32 = 37. If you want the top and bottom border embossed, 37+128 = 165.

Button Box(title, script)

Function:
Constructs a button with the text title that executes script when clicked.

Returns:
The display box (button box).

Arguments:

title  a quoted string or a reference to a string.
script  a quoted string or a reference to a string where the string is a valid JSL script.

Check Box(list, <script>)

Function:
Constructs a display box to show one or more check boxes.

Returns:
The display box (CheckBox).

Arguments:

list  a list of quoted strings or a reference to a list of strings.
script  an optional JSL script.

Example:

checkbox({"one", "two", "three"}, <<set(1, 1), <<set(3, 1))
Produces three checkboxes labelled “one”, “two”, and “three”. The first and third checkbox are
checked.
Col List Box(<all>, <width(pix)>, <maxSelected(n)>, <nlines(n)>, <script>, <MaxItems(n)>, <MinItems(n)>, <character|numeric>)

Function:
Constructs a display box to show a list box that allows selection of data table columns.

Returns:
The display box (ColList Box).

Arguments:
- **all** an optional command that adds all columns of the current data table into the list.
- **width(pix)** an optional command that sets the width of the list box to pix. pix is a number that measures pixels.
- **maxSelected(n)** an optional command that sets whether only one item can be selected. For \( n>1 \), \( n \) is ignored.
- **nlines(n)** an optional command that sets the length of the list box to \( n \) number of lines. \( n \) is an integer.
- **script** an optional script.
- **MaxItems(n)** An optional number that only allows \( n \) columns to be added to the list.
- **MinItems(n)** An optional number that only requires at least \( n \) columns for the list. If \( n=2 \), the top two slots in the Col List Box an initial display of "required numeric" (or whatever you've set the data type to be).
- **character | numeric** If you don't specify All, adding either character or numeric results in an empty collistbox that lists "optional character" or "optional numeric".

Note:
The maxSelected argument only affects whether one or more than one item can be selected. It does not enforce a limit greater than 1.

Combo Box(list, <script>)

Function:
Constructs a display box to show a combo box with a popup menu.

Returns:
The display box (ComboBox).

Arguments:
- **list** a list of quoted strings or a reference to a list of strings.
- **script** an optional JSL script.

Excerpt Box(report, subscripts)

Function:
Returns a display box containing the excerpt designated by the report held at number report and the list of display subscripts subscripts. The subscripts reflect the current state of the report, after previous excerpts have been removed.
H Center Box(display box)

Function:
Returns a display box with the display box argument centered horizontally with respect to all other sibling display boxes.

H Sheet Box(<<Hold(report), display boxes)

Function:
Returns a display box that arranges the display boxes provided by the arguments in a horizontal layout. The <<Hold() message tells the sheet to own the report(s) that will be excerpted.

Hier Box("text", Hier Box(...), ...)

Function:
Constructs a node of a tree (similar to Diagram output) containing text. Hier Box can contain additional Hier Boxes, allowing you to create a tree. The text may be a Text Edit Box.

Icon Box(name)

Function:
Constructs a display box containing an icon, where the quoted string name can be the name of any JMP icon. For example, Icon Box("Nominal") constructs a display box that contains the Nominal icon.

Argument:
name Quoted string that is the name of a JMP icon.

If Box(boolean, display boxes)

Function:
Constructs a display box whose contents are conditionally displayed.

Arguments:
boolean 1 displays the display boxes inside the If Box, does not display them.
display boxes Any display box tree.

Note:
If you use a variable to contain the boolean argument, you can display and remove the display box arguments merely by changing the variable's value.

Journal Box("Journal Text")

Function:
Constructs a display box that displays the quoted string journal box. We recommend that you do not generate the journal text by hand.
Lineup Box(<NCol(n)>, <Spacing(pixels)>, display boxes, ...)
Function:
Constructs a display box to show an alignment of boxes in \( n \) columns.

List Box(OperItem("item", ...), <width(pixels)>, <maxSelected(n)>, <nLines(n)>, <script>)
Function:
Constructs a display box to show a list box of selection items.

Matrix Box(matrix)
Function:
Displays the matrix given in the usual array form.

New Window("title", display box tree)
Function:
Makes a new window with the indicated title (a required argument) and a display box tree.
Additional Arguments:
  1. <<Script, <script>> Creates a new script window. The optional quoted string script is placed inside the script window.
  2. <<Journal>> Creates an empty journal.
  3. <<Size(x, y)>> Creates a window set to the size described by the \( x \) and \( y \) coordinates.

Number Col Box("title", numbers)
Function:
Creates a column named title with numeric entries given in list or matrix form.

Outline Box("title", display box, ...)
Function:
Creates a new outline named title containing the listed display boxes.

Page Break Box()
Function:
Creates a display box that forces a page break.

Panel Box("title", display box)
Function:
Creates a display box labeled with the quoted string title that contains the listed display boxes.
Plot Col Box(title, numbers)
Function:
Returns a display box labeled with the quoted string title to graph the numbers. The numbers may be either a list or a matrix.

Popup Box({"command1", script1, "command2", script2, ...})
Function:
Creates a popup box. The single argument is an expression yielding a list of an even number of items alternating between the command string and the expression you want evaluated when the command is selected. If the command is an empty string, a separator line is inserted.

Radio Box({"item", ...}, <script>)
Function:
Constructs a display box to show a set of radio buttons. The optional script is run every time a radio button is selected.

Report(obj)
Function:
Returns the display tree of a platform obj. This can also be sent as a message to a platform: obj<<Report

Script Box(<script>, <width>, <height>)
Function:
Constructs an editable box that contains the quoted string script. The editable box is a script window and may both be edited and run as JSL.
Arguments:
- script An optional quoted string that will appear in the script box.
- width An optional integer that sets the width of the script box.
- height An optional integer that sets the height of the script box.

Scene Box(x size, y size)
Function:
Creates an x by y-sized scene box for 3-D graphics.

Send(obj, message)
obj << message
Function:
Sends a message to a platform object.
Sheet Part(title, display box)
Function:
    Returns a display box containing the display box argument with the quoted string title as its title.

Slider Box(min, max, global, script, <set width(n)>, <rescale slider(min, max)>)
Function:
    Creates an interactive slider control.
Returns:
    The display box (SliderBox).
Arguments:
    min, max  Numbers that set the minimum and maximum value the slider represents.
    global  the global variable whose value is set and changed by the slider box.
    script  Any valid JSL commands that is run as the slider box is moved.
    set width(n) specify the width of the slider box in pixels.
    rescale slider(l, u) resets the max and min values for the slider box.
Notes:
    You can send Set Width and Rescale Slider as commands to a slider object. For example:
    ex = .6;
    New Window("Example", mybox=Slider Box(0, 1, ex, Show(ex));
    mybox<<Set Width(200)<<Rescale Slider(0, 5);

String Col Box("title", {"string", ...})
Function:
    Creates column in the table containing the string items listed.

Tab Box("page title1", contents of page 1, "page title 2", contents of page 2, ...)
Function:
    A tabbed dialog pane. The arguments are an even number of items alternating between the name of a tab page and the contents of the tab page.

Table Box(display box, ...)
Function:
    Creates a report table with the display boxes listed as columns.
Text Box("text", script)
Text Box("text", << Set Script(script))

Function:
Constructs a box that contains the quoted string text. The optional script argument attaches a script to the text box, either by adding the script as a second argument, or by sending the Set Script message.

Text Edit Box("text")

Function:
Constructs an editable box that contains the quoted string text.

V Center Box(display box)

Function:
Returns a display box with the display box argument centered vertically with respect to all other sibling display boxes.

V List Box(display box, ...)

Function:
Creates a display box that contains other display boxes and displays them vertically.

V Sheet Box(<<Hold(report), display boxes)

Function:
Returns a display box that arranges the display boxes provided by the arguments in a vertical layout. The <<Hold() message tells the sheet to own the report(s) that will be excerpted.

Subscripting for Display Boxes

Db[outlineBox | columnBox("text")]
Db["text"]
Subscript(db, spec)

Function:
Returns the display box of type outlineBox or columnBox titled with text. Text can include the ? wildcard character.

Db[boxType(n)]

Function:
Returns the n-th display box of the given boxType.
\textbf{db}[n]}

\textbf{Function:}
Returns the \textit{n}th child of the box, e.g. the \textit{n}th column for a table box or the \textit{n}th sub-node of an outline node.

\textbf{Messages for All Display Boxes}

\texttt{obj << message}
\texttt{Send(obj, message)}

Sends a \textit{message} to a platform \textit{object}.

\texttt{Show Properties(db)}

Shows the messages a given display box can interpret.

\texttt{db<<Select}
\texttt{db<<Deselect}

Selects (highlights) or deselects the box.

\texttt{db<<Reshow}

Forces a screen-refresh.

\texttt{db<<Journal}

Appends the box to the journal.

\texttt{db<<Copy Picture}

Puts a picture of the box on the clipboard.

\texttt{db<<Page Break}

Inserts a page break before the box.

\texttt{db<<Set Report Title("title")}

Sets a new title.

\texttt{db<<Class Name}

Returns the name of the display class for the box.
db<<Child

Returns the child or nth child of the box.

db<<Child(n)

Returns the sibling or nth sibling of the box.

db<<Parent

Returns the parent of this display box.

db<<Append(db2)

Add db2 to the display tree after db.

db<<Sib Append(db2)

Appends a display as a sibling to this one. The argument must evaluate to a display box owner or reference.

db<<Prepend(db2)

Add db2 to the display tree before db.

db<<Delete

Delete the display box

db<<Get Text

Returns a string containing the text of the box.

db<<Get HTML

Returns a string containing HTML source for the box.

db<<Get RTF

Returns a string containing RTF source for the box.

db<<Get Journal

Returns a string containing journal source for the box.
db<<Get Picture
  Captures \textit{db} as a Picture Object.

\begin{itemize}
  \item \textbf{db<<Get Script}
  \quad Returns the script for recreating the display box.
  \item \textbf{db<<Save Text("" | "pathname", format)}
  \quad Saves a file containing the text of the box.
  \item \textbf{db<<Save HTML("" | "pathname", format)}
  \quad Saves HTML source and folder of graphics in \emph{format} specified.
  \item \textbf{db<<Save MSWord("pathname", Native)}
  \quad Saves the display box as a Microsoft Word document. (Windows Only)
  \item \textbf{db<<Save RTF("" | "pathname", format)}
  \quad Saves RTF source with graphics in \emph{format} specified.
  \item \textbf{db<<Save Journal("" | "pathname")}
  \quad Saves journal source for the box.
  \item \textbf{db<<Save Picture("" | "pathname", format)}
  \quad Saves a picture of the box.
  \item \textbf{db<<Clone Box}
  \quad Makes a new copy of the display box.
  \item \textbf{db<<Scroll Window(x, y)}
  \quad Scrolls the containing window.
  \item \textbf{db<<Close Window}
  \quad Closes the containing window.
  \item \textbf{db<<Zoom Window}
  \quad Zooms the containing window.
\end{itemize}
Resizes the containing window.

Moves the containing window.

Appends the containing window of the display box to the journal; compare with \texttt{Journal}.

\textbf{Messages for Outline Boxes}

Closes the outline box.

Horizontally aligns the node's children.

Opens all the node's child nodes.

Closes all the node's child nodes.

Opens this node in all report windows from this platform.

Closes this node in all report windows from this platform.

Closes all nodes that do not have children.

Changes the title of the outline box.
outline box<<Get Title

Gets the title of the outline box.

**Messages for Frame Boxes**

_frame box<<Add Graphics Script(script)_

Adds a script to play inside the graphics frame.

_frame box<<Background Color(color)_

Changes the background color.

_frame box<<Edit Graphics Script_

Brings up a dialog box to view, edit, or delete the current graphics scripts.

_frame box<<Frame Size(x, y)_

Resets the size of the frame, in pixel units.

_frame box<<Marker Size(size)_

Changes the marker size.

_frame box<<X Axis(min, max, inc, named arguments)_

Scales the X coordinate system.

_frame box<<Y Axis(min, max, inc, named arguments)_

Scales the Y coordinate system.

_frame box<<Row Colors(color)_
_frame box<<Row Markers(marker)_
_frame box<<Row Exclude(bool)_
_frame box<<Row Hide(bool)_
_frame box<<Row Label(bool)_

Forwards commands to the data table associated with the report, so that the row states of selected rows can be manipulated.
Messages for Table Boxes

**table box<<Get**

Gets the entries of the table in list form.

**table box<<Get As Matrix**

Gets the numeric entries of the table in matrix form.

**table box<<Make Data Table(name)**

Turns the table entries into a new data table.

**table box<<Make Combined Data Table**

Same as Make Data Table, but also searches the report for report tables with the same columns and combine all of these into the new data table.

**table box<<Table Style(style)**

Sets the presentation style.

Messages for Text Boxes

**text box<<Font Color(n)**

Sets the color for Text strings.

**text box<<Get Text**

Returns the string content of the box.

**text box<<Rotate Text(direction)**

Rotates the text 90 degrees left or right, or returns it to the horizontal.

**text box<<Set Font Size(n)**

Sets the font size in points for text strings.

**text box<<Set Font Style(style)**

Sets the font style for text strings.

**text box<<Set Font(fontname)**

Sets the font for text strings.
text box<<Set Script(script)
Associate a script to a text box. The script executes when the user hits enter (or the text edit box otherwise loses focus).

text box<<Set Text("string")
Changes the string content of the box.

text box<<Set Wrap(n)
Set the wrap point, in pixels, in pixels (n).

Messages for Axis Boxes

axis box<<Add Axis Label("string")
Adds an axis label with the string given.

axis box<<Add Ref Line(number, linestyle, color)
Adds a reference line at number in the linestyle and color specified.

axis box<<Axis Settings(choices)
Applies numerous axis changes in a single message. Choose from the messages below for choices arguments.

axis box<<Decimal(width, decimalplaces)
Changes the numeric format for axis values.

axis box<<Format("name")
Changes to the numeric format given by name.

axis box<<Inc(n)
Set the increment between ticks.

axis box<<Interval(choice)
Specifies the units used for Inc() with date/time data formats.

axis box<<Max(max)
Changes just the maximum of the axis scale.
\texttt{axis box<<Min(min)}

Changes just the minimum of the axis scale.

\texttt{axis box<<Minor Ticks(number)}

Specifies the \textit{number} of minor tick marks between major tick marks.

\texttt{axis box<<Remove Axis Label}

Removes any label added with \texttt{Add Axis Label}.

\texttt{axis box<<Revert Axis}

Restores the axis' original settings (from time of creation).

\texttt{axis box<<Rotated Labels(bool)}

Rotates or un-rotates the labels for each tick value.

\texttt{axis box<<Scale(type)}

Changes the scale of the axis to \texttt{type} (Log or Linear).

\texttt{axis box<<Show Labels(boolean)}

Shows or hides labels for the axis values.

\texttt{axis box<<Show Major Grid(bool)}

Adds or removes grid lines at the major tick values.

\texttt{axis box<<Show Major Ticks(bool)}

Shows or hides major tick marks.

\texttt{axis box<<Show Minor Grid(bool)}

Adds or removes grid lines at the minor tick values.

\texttt{axis box<<Show Minor Ticks(bool)}

Shows or hides minor tick marks.
Messages for Number Col Boxes

```plaintext
number col box<<Get As Matrix
    Gets the values in a matrix, specifically a column vector.

number col box<<Get Format
    Returns the current format (width, decimalplaces). A decimalplaces > 100 indicates date/time values.

number col box<<Get Heading
    Returns the column heading text.

number col box<<Set Format(width, decimalplaces)
    Sets the format. Set decimalplaces > 100 for date/time values. Set decimalplaces = 97 for p-value format.

number col box<<Set Heading("text")
    Changes the column heading text.

number col box<<Get
number col box<<Get(i)
    Gets the values in a list, or the ith value.
```

Messages for String Col Boxes

```plaintext
string col box<<Get
string col box<<Get(i)
    Gets the values in a list, or the ith value.

string col box<<Get Heading
    Returns the column heading text.

string col box<<Set Heading("text")
    Changes the column heading text.
```
Messages for Matrix Boxes

matrix box<<Get
   Returns the matrix contents.

matrix box<<Make Into Data Table
   Turns the matrix into a new data table.

matrix box<<Set Format(width, decimalplaces)
   Sets the numeric format for matrix elements.

Messages for Nom Axis Boxes

nom axis box<<Divider Lines(boolean)
   Adds or removes divider lines between labels in the axis box.

nom axis box<<Lower Frame(boolean)
   Adds or removes a lower frame around the axis.

nom axis box<<Rotated Tick Labels(boolean)
   Rotates or un-rotates the labels at each tick value.

Modal Dialogs

Column Dialog(ColList("rolename"), specifications)
Function:
   Draws a dialog box for column role assignments.
Returns:
   A list of commands that were sent and the button that was clicked.
Arguments:
   ColList  Specifies the name of at least one list to add variables to.
   specifications  Any additional Dialog items (for example, Max Col, Datatype).
Dialog(contents)

Function:
   Draws a dialog box composed of contents given.

Returns:
   A reference to the dialog box.

Arguments:
   contents  Any number of possible JSL commands for a dialog.

Notes:
   To set the title of the dialog box, put Title() as the first item of the dialog; for example,
   Title("Select Options").

Graphs

Arc(x1, y1, x2, y2, startangle, endangle)

Function:
   Inscribes an arc in the rectangle described by the arguments.

Returns:
   A display box object.

Arguments:
   x1, y1  The point at the top left of the rectangle
   x2, y2  The point at the bottom right of the rectangle
   startangle, endangle  The starting and ending angle in degrees, where 0 degrees is 12 o’clock
                          and the arc or slice is drawn clockwise from startangle to endangle.

Arrow(<pixellength>, {x1, y1}, {x2, y2})

Function:
   Draws an arrow from the first point to the second point. The optional first argument specifies the
   length of the arrow’s head lines (in pixels).

Returns:
   A display box object.

Arguments:
   pixellength  Optional: specifies the length of the arrowhead in pixels.
   {x1, y1}, {x2, y2}  Two lists of two numbers that each specify a point in the graph.

Notes:
   The two points may also be enclosed in square brackets: Arrow(<pixellength>, [x1, x2],
   [y1, y2]).
Back Color(name)

Function:
Sets the color used for filling the graph's background.

Returns:
A display box object.

Argument:
name A name color or a color index.

Circle({x, y}, radius|PixelRadius(n), <...>, <fill>)

Function:
Draws a circle centered at {x, y} with the specified radius.

Returns:
Null.

Arguments:
{x, y} A number that describes a point in the graph
radius A number that describes the length of the circle's radius in relation to the vertical axis. If the vertical axis is resized, the circle will also be resized.
PixelRadius(n) A number that describes the length of the circle's radius in pixels. If the vertical axis is resized, the circle will not be resized.
fill Optional: A color which is used to fill the circle.

Note:
The center point and the radius can be placed in any order. You can also add additional center point and radius arguments and draw more than one circle in one statement. One point and several radii results in a bull's-eye. Adding another point still draws all previous circles, and then adds an additional circle with the last radius specified. This means that this code:
graphbox(circle({20, 30}, 5, {50, 50}, 15))
results in three circles, not two. First, a circle with radius 5 is drawn at 20, 30. Second, a circle with radius 5 is drawn at 50, 50. Third, a circle with radius 15 is drawn at 50, 50. Note that fill must be the last argument, and it is applied to all circles defined in this function.

Contour(xVector, yVector, zGridMatrix, zContour, <zColors>)

Function:
Draws contours given a grid of values.

Returns:
None.

Arguments:
xVector The n values that describe zGridMatrix.
yVector The m values that describe zGridMatrix.
zGridMatrix An n x m matrix of values on some surface.
zContour Optional: Definition of values for the contour lines.
**zColors**  Optional: Definition of colors to use for the contour lines.

**Contour Function**(expr, xName, yName, z, <XGrid(min, max, incr)>, <YGrid(min, max, incr)>, <zColor(color)>)

**Function:**
- Draws sets of contour lines of the expression, a function of the two symbols. The z argument can be a single value or an index or matrix of values.

**Returns:**
- None.

**Arguments:**
- **expr** Any expression; for example, \( \text{Sine}(y)+\text{Cosine}(x) \).
- **xName**, **yName** Values to use in the expression.
- **XGrid**, **YGrid** Defines a box, beyond which the contour lines are not drawn.
- **zColor** Defines the color in which to draw the contour lines.

**Drag Line**(xMatrix, yMatrix, <dragScript>, <mouseupScript>)

**Function:**
- Draws line segments between draggable vertices at the coordinates given by the matrix arguments.

**Returns:**
- None.

**Arguments:**
- **xMatrix** A matrix of \( x \)-coordinates.
- **yMatrix** A matrix of \( y \)-coordinates.
- **dragScript** Any valid JSL script; it is run at drag.
- **mouseupScript** Any valid JSL script; it is run at mouseup.

**Drag Marker**(xMatrix, yMatrix, <dragScript>, <mouseupScript>)

**Function:**
- Draws draggable markers at the coordinates given by the matrix arguments.

**Returns:**
- None.

**Arguments:**
- **xMatrix** A matrix of \( x \)-coordinates.
- **yMatrix** A matrix of \( y \)-coordinates.
- **dragScript** Any valid JSL script; it is run at drag.
- **mouseupScript** Any valid JSL script; it is run at mouseup.
Drag Polygon(xMatrix, yMatrix, <dragScript>, <mouseupScript>)

Function:
Draws a filled polygon with draggable vertices at the coordinates given by the matrix arguments.

Returns:
None.

Arguments:
- xMatrix: A matrix of x-coordinates.
- yMatrix: A matrix of y-coordinates.
- dragScript: Any valid JSL script; it is run at drag.
-mouseupScript: Any valid JSL script; it is run at mouseup.

Drag Rect(xMatrix, yMatrix, <dragScript>, <mouseupScript>)

Function:
Draws a filled rectangle with draggable vertices at the first two coordinates given by the matrix arguments.

Returns:
None.

Arguments:
- xMatrix: A matrix of 2 x-coordinates.
- yMatrix: A matrix of 2 y-coordinates.
- dragScript: Any valid JSL script; it is run at drag.
-mouseupScript: Any valid JSL script; it is run at mouseup.

Note:
xMatrix and yMatrix should each contain exactly two values. The resulting coordinate pairs should follow the rules for drawing a rect(); the first point (given by the first value in xMatrix and the first value in yMatrix) must describe the top, left point in the rectangle, and the second point (given by the second value in xMatrix and the second value in yMatrix) must describe the bottom, right point in the rectangle.

Drag Text(xMatrix, yMatrix, "text", <dragScript>, <mouseupScript>)

Function:
Draws the text (or all the items if a list is specified) at the coordinates given by the matrix arguments.

Returns:
None.

Arguments:
- xMatrix: A matrix of x-coordinates.
- yMatrix: A matrix of y-coordinates.
- text: A quoted string to be drawn in the graph.
dragScript Any valid JSL script; it is run at drag.

mouseupScript Any valid JSL script; it is run at mouseup.

**Fill Color(n)**

**Function:**
Sets the color used for filling solid areas.

**Returns:**
None.

**Argument:**
- n Index for a color or a quoted color name.

**Fill Pattern()**

**Function:**
Obsolete, as patterns have been removed. Use Transparency() instead.

**Global Box(global)**

**Function:**
Constructs a box for editing global value directly.

**Gradient Function(zexpr, xname, yname, [zlow, zhigh], zcolor([colorlow, colorhigh]), <XGrid(min, max, incr)>, <YGrid(min, max, incr)>)**

**Function:**
Fills a set of rectangles on a grid according to a color determined by the expression value as it crosses a range corresponding to a range of colors.

**Example:**
Gradient Function(Log(a * a + b * b), a, b, [2 10], ZColor([4, 6]))

*Zexpr* is an function in terms of the two following variables (a and b), whose values range from *zlow* to *zhigh* (2 to 10). *ZColor* defines the two colors that are blended together (4 is green, 6 is orange).

**Graph()**

**See:**
“Graph Box(properties, script),” p. 554

**Graph 3D Box(properties)**

**See:**
“Graph 3D Box(properties),” p. 559
Graph Box(properties, script)

Graph(properties, script)

Function:
Constructs a graph with axes.

Returns:
The display box (Graph Box).

Arguments:
properties Named property arguments: title("title"), XScale(low, high), YScale(low, high), FrameSize(h, v), XName("x"), YName("y"), DoubleBuffer.

script Any script to be run on the graph box.

Handle(a, b, dragScript, mouseupScript)

Function:
Places draggable marker at coordinates given by a, b. The first script is executed at drag and the second at mouseup.

H Line(<x1, x2>, y)

Function:
Draws a horizontal line at y across the graph. If you specify start and end points on the x-axis (x1 and x2), the line is drawn horizontally at y from x1 to x2. You can also draw multiple lines by using a matrix of values in the y argument.

H List Box(display box, ...)

Function:
Creates a display box that contains other display boxes and displays them horizontally.

H Size()

Function:
Returns the horizontal size of the graphics frame in pixels.

In Polygon(x, y, xx, yy)

In Polygon(x, y, xyPolygon)

Function:
Returns 1 or 0, indicating whether the point (x, y) is inside the polygon that is defined by the xx and yy vector arguments.
The vector arguments (xx, yy) can also be combined into a 2-column matrix (xyPolygon), allowing you to use three arguments instead of four. Also, x and y can be conformable vectors, and then a vector of 0s and 1s are returned based on whether each (x, y) pair is inside the polygon.
Line([x1, y1], [x2, y2], ...)
Line([x1, x2, ...], [y1, y2, ...])

Function:
Draws a line from point to point to point.

Arguments:
Can be any number of lists of two points, separated by commas; or a matrix of x's and a matrix of y's.

___

Line Style(n)

Function:
Sets the line style used to draw the graph.

Argument:

n Can be either a style name or the style's number:
- 0 or Solid
- 1 or Dotted
- 2 or Dashed
- 3 or DashDot
- 4 or DashDotDot

___

Marker(<markerState>, {x1, y1}, {x2, y2}, ...)
Marker(<markerState>, [x1, x2, ...], [y1, y2, ...])

Function:
Draws one or more markers at the points described either by lists or matrices. The optional markerState argument sets the type of marker.

___

Marker Size(n)

Function:
Sets the size used for markers.

___

Mousetrap(dragscript, mouseupscript)

Function:
Captures click coordinates to update graph parameters. The first script is executed at drag and the second at mouseup.
Normal Contour(prob, meanMatrix, stdMatrix, corrMatrix, <colorsMatrix>, <fill=x>)

Function:
Draws normal probability contours for \( k \) populations and two variables.

Arguments:
- prob A scalar or matrix of probabilities.
- meanMatrix A matrix of means of size \( k \) by 2.
- stdMatrix A matrix of standard deviations of size \( k \) by 2.
- corrMatrix A matrix of correlations of size \( k \) by 1.
- colorsMatrix Optional. Specifies the color(s) for the \( k \) contour(s). The colors must be specified as JSL colors (either JSL color integer values or return values of JSL Color functions such as RGB Color or HLS Color).
- fill=x Optional. Specifies the amount of transparency for the contour fill color.

Oval(x1, y1, x2, y2, fill)
Oval({x1, y1}, {x2, y2}, fill)

Function:
Draws an oval inside the rectangle whose diagonal has the coordinates \((x1, y1)\) and \((x2, y2)\). The oval is filled with the color \(fill\).

Pen Color(n)

Function:
Sets the color used for the pen.

Pen Size(n)

Function:
Sets the thickness of the pen in pixels.

Pie(x1, y1, x2, y2, startangle, endangle)

Function:
Draws a filled pie slice. The two points describe a rectangle, within which is a virtual oval. Only the slice described by the start and end angles is drawn.

Pixel Line To(x, y)

Function:
Draws a one-pixel-wide line from the current pixel location to the location given in pixel coordinates. Set the current pixel location using the Pixel Origin and Pixel Move To commands.
Pixel Move To(x, y)

Function:
Moves the current pixel location to a new location given in pixel coordinates.

Pixel Origin(x, y)

Function:
Sets the origin, in graph coordinates, for subsequent Pixel Line To or Pixel Move To commands.

Polygon({x1, y1}, {x2, y2}, ...)

Polygon(xmatrix, ymatrix)

Function:
Draws a filled polygon defined by the listed points.

Rect(x1, y1, x2, y2, <fill>)
Rect({x1, y1}, {x2, y2}, <fill>)

Function:
Draws a rectangle whose diagonal has the coordinates (x1, y1) and (x2, y2). Fill is boolean: if fill is 0, the rectangle is empty. If fill is non-zero, the rectangle is filled with the current fill color. The default value for fill is 0.

Text({x, y}, text, <properties>)

Function:
Draws the quoted string text at the given point. Properties can be any of several named arguments: Center Justified, Right Justified, Erased, Boxed, Counterclockwise, Position, and named arguments. Strings can be added anywhere, but the position and properties apply to all the strings.

Text Color(n)

Function:
Sets the color for Text strings.

Text Size(n)

Function:
Sets the font size in points for Text strings.
Transparency(alpha)

Function:
Sets the transparency of the current drawing, with alpha between 0 and 1 where 0 is clear (no drawing) and 1 is completely opaque (the default).

Note:
Not all operating systems support transparency.

V Line(x, <y1, y2>)

Function:
Draws a vertical line at x across the graph. If you specify start and end points on the y-axis (y1 and y2), the line is drawn vertically at x from y1 to y2. You can also draw multiple lines by using a matrix of values in the x argument.

X Function(expr, symbol, <Min(min), Max(max), Fill(value), Inc(bound), Show Details(n)>)

Function:
Draws a plot of the function as the symbol is varied over the y-axis of the graph.

X Scale(xmin, xmax)

Function:
Sets the range for the horizontal scale.

Y Function(expr, symbol, <Min(min), Max(max), Fill(value), Inc(bound), Show Details(n)>)

Function:
Draws a plot of the function as the symbol is varied over the x-axis of the graph.

Y Scale(ymin, ymax)

Function:
Sets the range for the vertical scale. If you don’t specify a scale, it defaults to (0, 1).

V Size()

Function:
Returns the vertical size of the graphics frame in pixels

X Origin()

Function:
Returns the x-value for the left edge of the graphics frame
X Range()
Function:
Returns the distance from the left to right edge of the display box. For example, X Origin() + X Range() is the right edge.

Y Origin()
Function:
Returns the y-value for the bottom edge of the graphics frame

Y Range()
Function:
Returns the distance from the bottom to top edges of a display box. For example, Y Origin() + Y Range() is the top edge.

3d Scenes

Graph 3D Box(properties)
Function:
Constructs a display box with 3D content.
Returns:
The display box.
Arguments:
Properties can include: framesize(x, y), Xname("title"), Yname("title"), Zname("title").

Note:
This display box constructor is experimental.

Statistical and Probability Functions

This section gives abbreviated descriptions for statistical and probability functions.
Probability Functions

**Beta Density**

**Function:**
Calculates the beta probability density function (pdf).

**Returns:**
The density function at quantile \( x \) for the beta distribution for the given arguments.

**Arguments:**
- \( x \) A quantile between 0 and 1.
- \( \text{alpha}, \beta \) Shape parameters which must both be greater than 0.
- \( \theta \) optional threshold. The allowable range is \( -\infty < \theta < \infty \). The default is 0.
- \( \sigma \) optional scale parameter, which must be greater than 0. The default is 1.

**See Also:**
The section on probability functions in the *JMP User Guide*.

**Beta Distribution**

**Function:**
Calculates the cumulative distribution function for the beta distribution.

**Returns:**
Returns the cumulative distribution function at quantile \( x \) for the beta distribution with shape arguments \( \text{alpha} \) and \( \beta \).

**Arguments:**
- \( x \) A quantile between 0 and 1.
- \( \text{alpha}, \beta \) Shape parameters which must both be greater than 0.

**See Also:**
The section on probability functions in the *JMP User Guide*.

**Beta Quantile**

**Function:**
Calculates the requested quantile for the beta distribution.

**Returns:**
Returns the \( p \)th quantile from the beta distribution with shape arguments \( \text{alpha} \) and \( \beta \).

**Arguments:**
- \( p \) The probability of the quantile desired; \( p \) must be between 0 and 1.
- \( \text{alpha}, \beta \) Shape parameters which must both be greater than 0.

**See Also:**
The section on probability functions in the *JMP User Guide*. 
ChiSquare Density(q, df, <center>)

Function:
The chi square density at q of the chi square with df degrees of freedom and optional non-centrality parameter center.

Returns:
The chi square density.

Arguments:
q  quantile
df  degrees of freedom.
center  non-centrality parameter

ChiSquare Distribution(q, df, <center>)

Function:
Returns cumulative distribution function at quantile x for chi square with df degrees of freedom centered at center.

ChiSquare Quantile(q, df, <center>)

Function:
Returns the pth quantile from the chi square distribution with df degrees of freedom, centered at center.

F Density(x, dfnum, dfden, <center>)

Function:
Returns the F density at x for the F distribution with numerator and denominator degrees of freedom dfnum and dfden, with optional noncentrality parameter center.

F Distribution(x, dfnum, dfden, <center>)

Function:
Returns cumulative distribution function at quantile x for F distribution with numerator and denominator degrees of freedom dfnum and dfden and noncentrality parameter center.

F Power(alpha, dfh, dfm, d, n)

Function:
Calculates the power from a given situation involving an F test or a t test.

F Quantile(x, dfnum, dfden, <center>)

Function:
Returns the pth quantile from the F distribution with numerator and denominator degrees of freedom dfnum and dfden and noncentrality parameter center.
**F Sample Size**

Function:
Calculates the sample size from a given situation involving an F test or a t test.

**Gamma Density**

Function:
Calculates the density at q of a Gamma probability distribution.

Returns:
The density function at quantile q for the gamma density distribution for the given arguments.

Arguments:
- q A quantile.
- alpha Shape parameters which must be greater than 1.
- scale Optional scale. The allowable range is $-\infty < 0 < \infty$. The default is 0.
- threshold Optional threshold parameter, which must be greater than 0. The default is 1.

See Also:
The section on probability functions in the *JMP User Guide*.

**Gamma Distribution**

Function:
Returns cumulative distribution function at quantile x for the gamma distribution with shape, scale, and threshold given.

**Gamma Quantile**

Function:
Returns the pth quantile from the gamma distribution with the shape, scale, and threshold parameters given.

**GLog Density**

Function:
Returns the density at q of a generalized logarithmic distribution with location mu, scale sigma, and shape lambda.

**GLog Distribution**

Function:
Returns the probability that a generalized logarithmically distribution random variable is less then q.

**GLog Quantile**

Function:
Returns the quantile for whose value the probability is p that a random value would be lower.
Johnson Sb Density(q, gamma, delta, theta, sigma)

Function:
Returns the density at $q$ of a Johnson Sb distribution.

Arguments:
- $q$ A value that is in the interval $\theta$ to $\theta + \sigma$.
- $\gamma$ Shape parameter that may be any value.
- $\delta$ Shape parameter that must be greater than 0.
- $\theta$ Location parameter that may be any value.
- $\sigma$ Scale parameter that must be greater than 0.

Johnson Sb Distribution(q, gamma, delta, theta, sigma)

Function:
Returns the probability that a Johnson Sb-distributed random variable is less than $q$.

Arguments:
- $q$ A value that is in the interval $\theta$ to $\theta + \sigma$.
- $\gamma$ Shape parameter that may be any value.
- $\delta$ Shape parameter that must be greater than 0.
- $\theta$ Location parameter that may be any value.
- $\sigma$ Scale parameter that must be greater than 0.

Johnson Sb Quantile(p, gamma, delta, theta, sigma)

Function:
Returns the quantile whose value for which the probability is $p$ that a random value would be lower.

Arguments:
- $p$ The probability of the quantile desired; $p$ must be between 0 and 1.
- $\gamma$ Shape parameter that may be any value.
- $\delta$ Shape parameter that must be greater than 0.
- $\theta$ Location parameter that may be any value.
- $\sigma$ Scale parameter that must be greater than 0.

Johnson Sl Density(q, gamma, delta, theta, sigma)

Function:
Returns the density at $q$ of a Johnson Sl distribution.

Arguments:
- $q$ A value that is in the interval $\theta$ to $+\infty$.
- $\gamma$ Shape parameter that may be any value.
- $\delta$ Shape parameter that must be greater than 0.
theta Location parameter that may be any value.
sigma Parameter that defines if the distribution is skewed positively or negatively. Sigma must be equal to either +1 (skewed positively) or -1 (skewed negatively).

Johnson Sl Distribution(q, gamma, delta, theta, sigma)
Function:
Returns the probability that a Johnson Sl-distributed random variable is less than q.
Arguments:
q A value that is in the interval theta to +infinity.
gamma Shape parameter that may be any value.
delta Shape parameter that must be greater than 0.
theta Location parameter that may be any value.
sigma Parameter that defines if the distribution is skewed positively or negatively. Sigma must be equal to either +1 (skewed positively) or -1 (skewed negatively).

Johnson Sl Quantile()
Function:
Returns the quantile whose value for which the probability is p that a random value would be lower.
Arguments:
p The probability of the quantile desired; p must be between 0 and 1.
gamma Shape parameter that may be any value.
delta Shape parameter that must be greater than 0.
theta Location parameter that may be any value.
sigma Parameter that defines if the distribution is skewed positively or negatively. Sigma must be equal to either +1 (skewed positively) or -1 (skewed negatively).

Johnson Su Density(q, gamma, delta, theta, sigma)
Function:
Returns the density at q of a Johnson Su distribution.
Arguments:
q A value that is between -infinity and +infinity.
gamma Shape parameter that may be any value.
delta Shape parameter that must be greater than 0.
theta Location parameter that may be any value.
sigma Scale parameter that must be greater than 0.
Johnson Su Distribution \((q, \gamma, \delta, \theta, \sigma)\)

Function:
Returns the probability that a Johnson Su-distributed random variable is less than \(q\).

Arguments:
- \(q\) A value that is between -infinity and +infinity.
- \(\gamma\) Shape parameter that may be any value.
- \(\delta\) Shape parameter that must be greater than 0.
- \(\theta\) Location parameter that may be any value.
- \(\sigma\) Scale parameter that must be greater than 0.

Johnson Su Quantile()

Function:
Returns the quantile whose value for which the probability is \(p\) that a random value would be lower.

Arguments:
- \(p\) The probability of the quantile desired; \(p\) must be between 0 and 1.
- \(\gamma\) Shape parameter that may be any value.
- \(\delta\) Shape parameter that must be greater than 0.
- \(\theta\) Location parameter that may be any value.
- \(\sigma\) Scale parameter that must be greater than 0.

Normal Biv Distribution \((x, y, r, \mu_1, s_1, \mu_2, s_2)\)

Function:
Computes the probability that an observation \((X, Y)\) is less than or equal to \((x, y)\) with correlation coefficient \(r\) where \(X\) is individually normally distributed with mean \(\mu_1\) and standard deviation \(s_1\) and \(Y\) is individually normally distributed with mean \(\mu_2\) and standard deviation \(s_2\). If \(\mu_1, s_1, \mu_2, s_2\) are not given, the function assumes the standard normal bivariate distribution with \(\mu_1=0, s_1=1, \mu_2=0, \text{ and } s_2=1\).

Normal Density \((x, \mu, s)\)

Function:
Returns the value of the density function at quantile \(x\) for the normal distribution with \(\mu\) and \(s\). The default \(\mu\) is 0. the default \(s\) is 1.

Normal Distribution \((x, \mu, s)\)

Function:
Returns the cumulative distribution function at quantile \(x\) for the normal distribution with \(\mu\) and \(s\). The default \(\mu\) is 0. the default \(s\) is 1.
Normal Quantile\( (p, <\text{mean}>, <\text{stddev}>) \)
Probit\( (p, <\text{mean}>, <\text{stddev}>) \)

Function:
Returns the \( p \)th quantile from the normal distribution with \textit{mean} and \textit{stddev}. The default \textit{mean} is 0. the default \textit{stddev} is 1.

Students t Density\( (x, df) \)
\textit{t} Density\( (x, df, \text{center}) \)

Function:
Returns the value of the density function at quantile \( x \) for the Student's t distribution with degrees of freedom \( df \) and \textit{center}.

Students t Distribution\( (x, df) \)
\textit{t} Distribution\( (x, df, \text{center}) \)

Function:
Returns the cumulative distribution function at quantile \( x \) for Student's \textit{t} distribution with \textit{df} and \textit{center}.

Students t Quantile\( (p, df) \)
\textit{t} Quantile\( (p, df, \text{center}) \)

Function:
Returns the \( p \)th quantile from the Student's \textit{t} distribution with \textit{df} and \textit{center}.

Tukey HSD P Value\( (q, n, \text{dfe}) \)

Function:
Returns the \( p \)-value from Tukey's HSD multiple comparisons test.
Arguments:
- \( q \) The test statistic.
- \( n \) The number of groups in the study.
- \( \text{dfe} \) The error degrees of freedom, based on the total study sample.

Tukey HSD Quantile\( (1-\alpha, n, \text{dfe}) \)

Function:
Returns the quantile needed in Tukey's HSD multiple comparisons test.
Arguments:
- \( 1-\alpha \) The confidence level.
- \( n \) The number of groups in the study.
- \( \text{dfe} \) The error degrees of freedom, based on the total study sample.
Weibull Density(x, shape, <scale, threshold>)

Function:
Returns the value of the density function at quantile x for the Weibull distribution with the parameters given.

Weibull Distribution(x, shape, <scale, threshold>)

Function:
Returns the cumulative distribution function at quantile x for the Weibull distribution with the parameters given.

Weibull Quantile(p, shape, <scale, threshold>)

Function:
Returns the pth quantile from the Weibull distribution with the parameters given.

Discrete Probability Functions

Beta Binomial Distribution(k, p, n, delta)

Function:
Returns the probability that a beta binomially distributed random variable is less than or equal to k.

Beta Binomial Probability(k, p, n, delta)

Function:
Returns the probability that a beta binomially distributed random variable is equal to k.

Beta Binomial Quantile(p, n, delta, cumprob)

Function:
Returns the quantile whose value the probability of observing a beta binomial(p, n, delta) random variables of equal or smaller value is cumprob.

Binomial Distribution(p, n, k)

Function:
The probability that a binomially distributed random variable is less than or equal to k.

Returns:
The cdf for the binomial distribution with n trials, probability p of success for each trial, and k successes.

Arguments:
p  probability of success for each trial
n  number of trials
Statistical and Probability Functions

k  number of successes

See Also:
The section on probability functions in the *JMP User Guide*.

**Binomial Probability**(*p*, *n*, *k*)

Function:
The probability that a binomially distributed random variable is equal to *k*.

Returns:
The probability of getting *k* successes out of *n* trials if the probability is *p*.

Arguments:
- *p*  probability of success for each trial
- *n*  number of trials
- *k*  number of successes

See Also:
The section on probability functions in the *JMP User Guide*.

**Binomial Quantile**(*p*, *n*, *m*)

Function:
The probability of observing a binomial random variable of equal or smaller value.

Returns:
The quantile for which the probability of observing a binomial(*p*, *n*) random variables of equal or smaller value is the percentile *m*.

Arguments:
- *p*  probability of success for each trial
- *n*  number of trials
- *m*  percentile

**Gamma Poisson Distribution**(*k*, *lambda*, *sigma*)

Function:
Returns the probability that a gamma-Poisson distributed random variable is less than or equal to *k*.

Arguments:
- *k*  The count of interest.
- *lambda*  The mean parameter.
- *sigma*  The overdispersion parameter.
Gamma Poisson Probability\((k, \lambda, \sigma)\)

Function:
Returns the probability that a gamma-Poisson distributed random variable is equal to \(k\).

Arguments:
- \(k\) The count of interest.
- \(\lambda\) The mean parameter.
- \(\sigma\) The overdispersion parameter.

Gamma Poisson Quantile\((\lambda, \sigma, \text{percentile})\)

Function:
Returns the quantile whose value for which the probability of observing a Gamma Poisson(\(\lambda, \sigma\)) random variable of equal or smaller value is \(p\).

Hypergeometric Distribution\((N, K, n, x, <r>)\)

Function:
Returns the cumulative distribution function at \(x\) for the hypergeometric distribution with population size \(N\), \(K\) items in the category of interest, sample size \(n\), count of interest \(x\), and optional odds ratio \(r\).

Hypergeometric Probability\((\text{pop}, k, n, x, <r>)\)

Function:
Returns the probability that a hypergeometrically distributed random variable is equal to \(x\).

Arguments:
- \(\text{pop}\) Population size.
- \(k\) The Number of items in the category of interest.
- \(n\) Sample size.
- \(x\) Count of interest.
- \(r\) Optional odds ratio.

Neg Binomial Distribution\((p, n, k)\)

Function:
Returns the cdf for the negative binomial distribution with \(n\) trials, probability \(p\) of success for each trial, and \(k\) successes.

Neg Binomial Probability\((p, n, k)\)

Function:
Returns the probability that a negative binomially distributed random variable is equal to \(k\), where \(n\) is the number of trials and \(p\) is the probability of success for each trial.
Poisson Distribution($\lambda$, $k$)
Function:
Returns the cumulative distribution function at $k$ for the Poisson distribution with mean $\lambda$.

Poisson Probability($\lambda$, $k$)
Function:
Returns the probability that a Poisson distributed random variable with mean $\lambda$ is equal to $k$.

Poisson Quantile($\lambda$, percentile)
Function:
Returns the quantile for which the probability of observing a Poisson($\lambda$) random variable of equal or smaller value is $k$.

Specialty Probability Functions

ChiSquare Noncentrality($x$, $df$, prob)
Function:
Solves the noncentrality such that $\text{prob=ChiSquare Distribution (x, df, nc)}$

F Noncentrality($x$, ndf, ddf, prob)
Function:
Solves the noncentrality such that $\text{prob=F Distribution (x, ndf, ddf, nc)}$

T Noncentrality($x$, $df$, prob)
Function:
Solves the noncentrality such that $\text{prob=T Distribution (x, df, nc)}$

Log Density Functions

ChiSquare Log C Distribution($x$, $df$, <nc>)
Function:
Returns $1 - \text{log (value)}$ of the distribution function at quantile $x$ for the Chi Square distribution.

ChiSquare Log Density($x$, $df$, <nc>)
Function:
Returns the log of the value of the density function at quantile $x$ for the Chi Square distribution.
ChiSquare Log Distribution(x, df, <nc>)

Function:
Returns the log of the value of the distribution function at quantile x for the Chi Square distribution.

F Log CDistribution(x, dfn, dfd, <nc>)

Function:
Returns 1 - log (value) of the normal distribution function at quantile x for the F distribution.

F Log Density(x, dfn, dfd, <nc>)

Function:
Returns the log of the value of the density function at quantile x for the F distribution.

F Log Distribution(x, dfn, dfd, <nc>)

Function:
Returns the log of the value of the distribution function at quantile x for the F distribution.

Gamma Log CDistribution(x, alpha, <scale = 1>, <threshold = 0>)

Function:
Same as Log (1 - Gamma Distribution(x, alpha)) except that it has a much greater range.

Gamma Log Density(x, alpha, <scale = 1>, <threshold = 0>)

Function:
Same as Log(Gamma Density(x, alpha)) except that it has a much greater range.

Gamma Log Distribution(x, alpha, <scale = 1>, <threshold = 0>)

Function:
Same as Log(Gamma Distribution(x, alpha)) except that it has a much greater range.

Normal Log CDistribution(x)

Function:
Returns 1 - log (value) of the distribution function at quantile x for the normal distribution.

Normal Log Density(x)

Function:
Returns the log of the value of the density function at quantile x for the normal distribution.
Normal Log Distribution(x)

Function:
Returns the log of the value of the distribution function at quantile x for the normal distribution.

t Log CDistribution(x, df, <nc>)

Function:
Returns 1 - log (value) of the normal distribution function at quantile x for the t distribution.

t Log Density(x, df, <nc>)

Function:
Returns the log of the value of the density function at quantile x for the t distribution.

t Log Distribution(x, df, <nc>)

Function:
Returns the log of the value of the distribution function at quantile x for the t distribution.

Statistical Functions

Best Partition(xindices, yindices, <<Ordered, <<Continuous Y, <<Continuous X)

Function:
Experimental function to determine the optimal grouping.

Returns:
A list.

Arguments:
xindices, yindices Same-dimension matrices.

CDF(YVec)

Function:
Returns values of the empirical cumulative probability distribution function for YVec. Cumulative probability is the proportion of data values less than or equal to the corresponding entry in QuantVec.

Desirability(yVector, desireVector, y)

Function:
Fits a function to go through the three points, suitable for defining the desirability of a set of response variables (y's). yVector and desireVector are matrices with three values, corresponding to the
three points defining the desirability function. The actual function depends on whether the desire values are in the shape of a larger-is-better, smaller-is-better, target, or antitarget.

Returns:
The desirability function.

Arguments:
yVector Three input values.
desireVector the corresponding three desirability values.
y the value of which to calculate the desirability.

Maximum(var1, var2, ...)
Max(var1, var2, ...)
Function:
Row-wise maximum of the variables specified.

Mean(var1, var2, ...)
Function:
Row-wise mean of the variables specified.

Minimum(var1, var2, ...)
Min(var1, var2, ...)
Function:
Row-wise minimum of the variables specified.

Multivariate Normal Impute(y, mean, covariance)
Function:
Transforms the responses to the principal component space.

Arguments:
y The vector of responses.
mean The vector of response means.
covariance A symmetric matrix containing the response covariances.

N Missing(expression)
Function:
Row-wise number of missing values in variables specified.

Number(var1, var2, ...)
Function:
Row-wise number of non-missing values in variables specified.
Product(i=initialValue, limitValue, bodyExpr)
Function:
Multiplies the results of bodyExpr over all i until the limitValue and returns a single product.

Quantile(p, arguments)
Function:
Quantile of the arguments, where p is a value between 0 and 1, and the arguments are numbers, lists of numbers, or matrices.

SbInv(z, gamma, delta, theta, sigma)
Function:
Johnson Sb inverse transformation. If argument is normal, the result is Johnson Sb.

SbTrans(x, gamma, delta, theta, sigma)
Function:
Johnson Sb transformation from a doubly-bound variable to a more normal (0, 1) distribution.

SlInv(z, gamma, delta, theta, sigma)
Function:
Johnson Sl inverse transformation. If argument is normal, the result is Johnson Sl.

SlTrans(x, gamma, delta, theta, sigma)
Function:
Johnson Sl transformation from a doubly-bound variable to a more normal (0, 1) distribution.

Spline Coef(x, y, lambda)
Function:
Returns a five column matrix of the form knots||a||b||c||d where knots is the unique values in x.

Spline Eval(x, coef)
Function:
Evaluates the spline predictions using the coef matrix, where coef is in the same form as Spline Coef().

Spline Smooth(x, y, lambda)
Function:
Returns the smoothed predicted values from a spline fit.
**Statistical and Probability Functions**

**Std Dev(var1, var2, ...)**

Function:
Row-wise standard deviation of the variables specified.

**SuInv(z, gamma, delta, theta, sigma)**

Function:
Johnson Su inverse transformation. If argument is normal, the result is Johnson Su.

**Sum(var1, var2, ...)**

Function:
Row-wise sum of the variables specified.

**Summation(init, limitvalue, body)**

Function:
Summation sums the results of the body statement(s) over all $i$ to return a single value.

**Summarize(<by>, <count>, <sum>, <mean>, <min>, <max>, <stddev>, <quantile>)**

Function:
Gathers summary statistics for a data table and stores them in global variables.

Returns:
None.

Arguments:
All arguments are optional and can be included in any order. Typically, each argument is assigned to a variable so you can display or manipulate the values further.

- name=By(col) Using a By variable changes the output from single values for each statistic to a list of values for each group in the by variable.
- c=Count The number of values in the column (or for each By group).
- sum=Sum(col) The sum of values in the column (or for each By group).
- mean=Mean(col) The mean of the column (or of each By group).
- min=Min(col) The minimum value in the column (or in each By group).
- max=Max(col) The maximum value in the column (or in each By group).
- stddev=StdDev(col) The standard deviation of the column (or of each By group).
- quantile=Quantile(col, q) The quantile specified by q for the column (or for each By group).

Note:
If all data are excluded, missing values are returned by the summary statistics commands. If all rows are deleted, the summary statistics commands return empty.
SuTrans(x, gamma, delta, theta, sigma)

Function:
Johnson Su transformation from a doubly-bound variable to a more normal (0, 1) distribution.

Column Statistics

Col Maximum(name)
Col Max(name)

Function:
Calculates the maximum value across all rows of the specified column.

Returns:
The maximum value that appears in the column.

Argument:
name a column name

Col Mean()

Function:
Calculates the mean across all rows of the specified column.

Returns:
The mean of the column.

Argument:
name a column name

Col Minimum(name)
Col Min(name)

Function:
Calculates the minimum value across all rows of the specified column.

Returns:
The minimum value that appears in the column.

Argument:
name a column name
Col N Missing(name)

Function:
Calculates the number of missing values across all rows of the specified column.

Returns:
The number of missing values in the column.

Argument:
name a column name.

Col Number(name)

Function:
Calculates the number of non-missing values across all rows of the specified column.

Returns:
The number of non-missing values in the column.

Argument:
name a column name.

Col Quantile(name, q)

Function:
Calculates the specified quantile q of across all rows of the specified column.

Returns:
The value of the quantile.

Argument:
name a column name.
q a number between 0 and 1 that represents a quantile.

Example:
Using Big Class.jmp:
colquantile(:height, .5)
63
63 is the 50th percentile, or the median, of all rows in the height column.
Col Standardize(name)

Function:
Calculates the column mean divided by the standard deviation across all rows of the specified column.

Returns:
The standardized mean.

Argument:
name a column name.

Notes:
Standardizing centers the variable by its sample standard deviation. Thus, the following commands are equivalent:
dt<<New Column("stdht", Formula(Col Standardize(height)));

dt<<New Column("stdht2", Formula((height-Col Mean(height))/Col Std Dev(height)));

Col Std Dev(name)

Function:
Calculates the standard deviation across rows in a column.

Returns:
The standard deviation.

Argument:
name a column name.

Col Sum(name)

Function:
Calculates the sum across rows in a column.

Returns:
The sum.

Argument:
name a column name.
**Additional Statistical Functions**

**FFT(list, <named arguments>)**

**Function:**
Conducts a Fast Fourier Transformation (FFT) a list of matrices.

**Argument:**
- **List** A list of one or two matrices. If two are provided, the second one is the imaginary part. Both matrices must have the same dimensions, and both must have more than one row.

**Named Arguments:**
- Note: all named arguments are optional.
  - <<inverse(boolean)> If true (1), an inverse FFT is conducted.
  - <<multivariate(boolean)> If true (1), a multivariate FFT is conducted. If false(0), a spacial FFT is conducted.
  - <<scale(number)> Multiplies the return values by the specified number.

**Hier Clust(x)**

**Function:**
Returns the clustering history for a hierarchical clustering using Ward’s method (without standardizing data).

**Argument:**
- **x** A data matrix.

**IRT Ability(Q1, <Q2, Q3, ...,> parameter matrix)**

**Function:**
Solves for ability in an Item Response Theory model.

**KDE(vector, <named arguments>)**

**Function:**
Returns a kernel density estimator with automatic bandwidth selection.

**Argument:**
- **vector** A vector.

**Named Arguments:**
All the named arguments are optional.
- <<weights> Must be a vector of the same length as vector, and can contain any non-negative real numbers. Weights represents frequencies, counts, or similar concepts.
- <<bandwidth(n)> A non-negative real number. Enter a value of 0 to use the bandwidth selection argument.
- <<bandwidth scale(n)> A positive real number.
<<bandwidth selection(n) >
  n must be 0, 1, 2, or 3, corresponding to Sheather and Jones, Normal Reference, Sliverman rule of thumb, or Oversmoother, respectively.
<<kernel(n) >
  n must be 0, 1, 2, 3, or 4, corresponding to Gaussian, Epanechnikov, Biweight, Triangular, or Rectangular, respectively.

LenthPSE(x)
Function:
  Returns Lenth’s pseudo-standard error of the values within a vector.
Argument:
  x A vector.

LPSolve(A, b, c, L, U, neq, nle, nge, <slackVars=0>)
Function:
  Returns a list containing the decision variables (and slack variables if applicable) in the first list item and the optimal objective function value (if one exists) in the second list item.
Arguments:
  A A matrix of constraint coefficients.
  b A matrix that is a column of right hand side values of the constraints.
  c A vector of cost coefficients of the objective function.
  L, U Matrices of lower and upper bounds for the variables.
  neq The number of equality constraints.
  nle The number of less than or equal inequalities.
  nge The number of greater than or equal inequalities.
  slackVars True (1) or false (2).
Note:
  The constraints must be listed as equalities first, less than or equal inequalities next, and greater than or equal inequalities last.

Mixture Moment(powers, <L>, <U>)
Function:
  Computes the exact moments of monomials over bounded subsets of the face of the unit simplex.
Arguments:
  powers The vector of polynomial powers in the monomial.
  L, U Optional lower and upper bounds that must be between 0 and 1. The default values for L and U are 0 and 1, respectively.
Multivariate Gaussian Quadrature(nDim, nStrata, nSim)

Function:
Returns a list of two elements: values and weights for radial-spherical integration for smooth functions of independent standard normal variables.

Arguments:
- nDim Number of dimensions.
- nStrata Number of strata.
- nSim Number of simulations.

Normal Integrate(muVector, sigmaMatrix, expr, x, nStrata, nSim)

Function:
Returns the result of radial-spherical integration for smooth functions of multivariate, normally-distributed variables.

Arguments:
- muVector A vector.
- sigmaMatrix A matrix.
- expr An expression in terms of the variable x.
- x The variable used in the expression expr.
- nStrata Number of strata.
- nSim Number of simulations.

Polytope Uniform Random(samples, A, b, L, U, neq, nle, nge, <nwarm=200>, <nstride=25>)

Function:
Generates random uniform points over a convex polytope.

Arguments:
- Samples The number of random points to be generated.
- A The constraint coefficient matrix.
- B The right hand side values of constraints.
- L, U The lower and upper bounds for the variables.
- neq The number of equality constraints.
- nle The number of less than or equal inequalities.
- nge The number of greater than or equal inequalities.
- nwarm Optional: The number of warm-up repetitions before points are written to the output matrix.
Optional: The number of repetitions between each point that is written to the output matrix.

**Note:**

The constraints must be listed as equalities first, less than or equal inequalities next, and greater than or equal inequalities last.

**Tolerance Limit**(1-alpha, p, n)

**Function:**

Constructs a 1-alpha confidence interval to contain proportion p of the means with sample size n.

---

### Advanced Concepts

This section gives abbreviated descriptions for advanced programming functions.

### Advanced Programming Functions

**Derivative**(expr, {name, ...}, ...)

**Function:**

Calculates the derivative of the expr with respect to name.

**Returns:**

Returns the derivative.

**Arguments:**

- expr Any expression.
- name May be a single variable or a list of variables.

**Note:**

Adding an additional variable (derivative(expr, name, name2)) takes the second derivative.

**Function**({arguments}, script)

**Function:**

Stores the body script with arguments as local variables.

**Returns:**

The function as defined.

When called later, it returns the result of the script given the specified arguments.

**Arguments:**

- {arguments} List of arguments.
- script Any valid JSL script.
LnZ(x)

Function:
Returns the natural logarithm of x, except when x=0; in that case, 0 is returned. This function is intended for internal use by derivatives.

Num Deriv(expr)

Function:
Returns the first numeric derivative of the expr with respect to the first argument in the expression.

Num Deriv2(expr)

Function:
Returns the second numeric derivative of the expr with respect to the first argument in the expression.

Parameter({name=value, ...}, model expression)

Function:
Defines formula parameters for models for the Nonlinear platform.

Recurse(function)

Function:
Makes a recursive call of the defining function.

Throw("text")

Function:
Returns a Throw. If you include text, throwing stores text in a global exception_msg. If text begins with “!”, throwing creates an error message about where the exception was caught.

Try(expr1, expr2)

Function:
Evaluates expr1. If that returns a Throw, stops, returns nothing, evaluates expr2 and returns result.

External Files

Includes

Include(pathname)

Function:
Opens the script file identified by the quoted string pathname, parses the script in it, and executes it.
Load DLL(path)

Function:
Loads the DLL specified in *path*.

Text Files

Load Text File(path, <XMLParse|SASODSXML>)

Function:
Opens the text file stored at *path*.

Returns
The text contained in the file.

Arguments:
- **path** A pathname that points to a text file. This can be a URL.
- **XMLParse** Optional argument that causes an XML text file to be converted into JSL.
- **SASODSXML** Optional argument that causes the text file to be parsed as SAS ODS default XML.

Save Text File(path, text)

Function:
Saves the JSL variable text into the file specified by *path*.

XML Files

Parse XML(string, On Element("tagname", Start Tag(expr), End Tag(expr)))

Function:
Parses an XML expression using the *On Element* expressions for specified XML tags.

XML Attr("attr name")

Function:
Extracts the string value of an xml argument in the context of evaluating a *Parse XML* command

XML Decode("xml")

Function:
Decodes symbols in XML to ordinary text. For example, &amp; becomes &, and &lt; becomes <.

Argument:
- **xml** A quoted string containing XML.
XML Encode("text")

Function:
Prepares text for embedding in XML. For example, & becomes &amp;, and < becomes &lt;.

Argument:
xml A quoted string containing plain text.

XML Text()

Function:
Extracts the string text of the body of an XML tag in the context of evaluating a Parse XML command.

Pattern Matching

Pat Abort()

Function:
Constructs a pattern that immediately aborts the pattern match. The matcher does not back up and retry any alternatives. Conditional assignments are not made. Immediate assignments that were already made are kept.

Returns:
0 when a match is aborted.

Argument:
none

Pat Altern(pattern1, <pattern 2, ...>)

Function:
Constructs a pattern that matches any one of the pattern arguments.

Returns:
A pattern.

Argument:
One or more patterns.

Pat Any("string")

Function:
Constructs a pattern that matches a single character in the argument.

Returns:
A pattern.

Argument:
string a string.
**Pat Arb()**

**Function:**
Constructs a pattern that matches an arbitrary string. Initially it will match the null string. It will match one additional character each time the pattern matcher backs into it.

**Returns:**
A pattern.

**Argument:**
none

**Example:**
```perl
p = "the beginning" + Pat Arb() >? stuffInTheMiddle + "the end";
P = Pat Match("in the beginning of the story, and not near the end, there are three bears", p);
show(stuffInTheMiddle);

stuffInTheMiddle:" of the story, and not near"
```

---

**Pat Arb No(pattern)**

**Function:**
Constructs a pattern that matches zero or more copies of `pattern`.

**Returns:**
A pattern.

**Argument:**

**Example:**
```perl
adjectives = "large" | "medium" | "small" | "warm" | "cold" | "hot" | "sweet";
rc = Pat Match("I would like a medium hot, sweet tea please",
             Pat ArbNo(adjectives | Pat Any("", "") >> adj +
             ("tea" | "coffee" | "milk"));
show(rc, adj);

adj:" medium hot, sweet "
```
**Pat At(varName)**

**Function:**
Constructs a pattern that matches the null string and stores the current position in the source string into the specified JSL variable (`varName`). The assignment is immediate, and the variable can be used with `expr()` to affect the remainder of the match.

**Returns:**
A pattern.

**Argument:**
`varName` the name of a variable to store the result in.

**Example:**
```
p = "":" + Pat At(listStart) + expr(if(listStart==1, Pat Immediate(Pat Len(3), early), Pat Immediate(Pat Len(2), late))); early = "; late = "
Pat Match(":123456789", p);
show(early,late); early = ""; late = "
Pat Match(" :123456789", p);
show(early,late);
```

First this is produced:
```
early:"123"
late:""
```

and later this:
```
early:"
late:"12"
```

**Pat Break("string")**

**Function:**
Constructs a pattern that matches zero or more characters that are not in its argument; it stops or breaks on a character in its argument. It fails if a character in its argument is not found (in particular, it fails to match if it finds the end of the source string without finding a break character).

**Returns:**
A pattern.

**Argument:**
`string` a string.
Pat Concat(pattern1, pattern2 <pattern 3, ...>)

**Pattern1 + Pattern2 + ...**

**Function:**
Constructs a pattern that matches each pattern argument in turn.

**Returns:**
A pattern.

**Argument:**
Two or more patterns.

Pat Conditional(pattern, type)

**Function:**
Saves the result of the pattern match, if it succeeds, to a variable named as the second argument (type) after the match is finished.

**Returns:**
A pattern.

**Arguments:**
- **pattern** a pattern to match against.
- **varName** the name of a variable to store the result in.

**Example:**
```javascript
var type = "undefined";
rc = Pat Match("green apples", Pat Conditional("red"|"green", type) + " apples");
show(rc, type);
```

```
rc:1
type:"green"
```

Pat Fail()

**Function:**
Constructs a pattern that fails whenever the matcher attempts to move forward through it. The matcher backs up and tries different alternatives. If and when there are no alternatives left, the match will fail and Pat Match will return 0.

**Returns:**
0 when a match fails.

**Argument:**
none
**Pat Fence()**

**Function:**
Constructs a pattern that succeeds and matches the null string when the matcher moves forward through it, but fails when the matcher tries to back up through it. It is a one-way trap door that can be used to optimize some matches.

**Returns:**
1 when the match succeeds, 0 otherwise.

**Argument:**
none

**Pat Immediate(pattern, varName)**

**Function:**
Saves the result of the pattern match to a variable named as the second argument (varName) immediately.

**Returns:**
A pattern.

**Arguments:**
- `pattern` a pattern to match against.
- `varName` the name of a variable to store the result in.

**Example:**
```plaintext
type = "undefined";
rc = Pat Match("green apples", ("red" | "green") >> type + " pears");
show(rc, type);
```

```plaintext
rc:0
  type:"green"
```
Even though the match failed, the immediate assignment was made.

**Pat Len(int)**

**Function:**
Constructs a pattern that matches n characters.

**Returns:**
A pattern.

**Argument:**
- `int` an integer that specifies the number of characters.
**Pat Match**(*SourceText*, *Pattern*, *<ReplacementText>*,, *<ANCHOR>*,, *<FULLSCAN>*
)

**Function:**
Pat Match executes the Pattern against the SourceText. The pattern must be constructed first, either inline or by assigning it to a JSL variable elsewhere.

**Returns:**
1 if the pattern is found, 0 otherwise.

**Arguments:**
- **SourceText** A string or a reference to a string that contains the text to be searched.
- **Pattern** A pattern variable (or reference to one) that describes the text to be searched for.
- **ReplacementText** Optional string that, if present, defines text to replace the pattern in the source text.
- **ANCHOR** Optional command to anchor the pattern match to the beginning of the string.
- **FULLSCAN** Optional command to force Pat Match to try all alternatives.

---

**Pat Not Any**("string")

**Function:**
Constructs a pattern that matches a single character that is not in the argument.

**Returns:**
A pattern.

**Argument:**
- **string** a string.

---

**Pat Pos**(int)

**Function:**
Constructs patterns that match the null string if the current position is int from the left end of the string, and fail otherwise.

**Returns:**
A pattern.

**Argument:**
- **int** an integer that specifies a position in a string.
**Pat R Pos(int)**

**Function:**
Constructs patterns that match the null string if the current position is *int* from the right end of the string, and fails otherwise.

**Returns:**
A pattern.

**Argument:**
*int* an integer that specifies a position in a string.

**Pat R Tab(int)**

**Function:**
Constructs a pattern that matches up to position *n* from the end of the string. It can match 0 or more characters. It fails if it would have to move backwards or beyond the end of the string.

**Returns:**
A pattern.

**Argument:**
*int* an integer that specifies a position in a string.

**Pat Regex(string)**

**Function:**
Constructs a pattern that matches the regular expression in the quoted *string* argument.

**Returns:**
A pattern.

**Argument:**
*string* a string.

**Pat Rem()**

**Function:**
Constructs a pattern that matches the remainder of the string. It is equivalent to *Pat R Tab(0)*.

**Returns:**
A pattern.

**Argument:**
none
Pat Repeat(pattern, minimum, maximum, GREEDY|RELUCTANT)

Function:
    Matches pattern between minimum and maximum times.

Returns:
    A pattern.

Arguments:
    pattern a pattern to match against.
    minimum An integer that must be smaller than maximum.
    maximum An integer that must be greater than minimum.

GREEDY|RELUCTANT If GREEDY is specified, it tries the maximum first and works back to the minimum. If RELUCTANT is specified, it tries the minimum first and works up to the maximum.

Notes:
    Pat Arno(p) is the same as Pat Repeat(p, 0, infinity, RELUCTANT)
    Pat Repeat(p) is the same as Pat Repeat(p, 1, infinity, GREEDY)
    Pat Repeat(p, n) is the same as Pat Repeat(p, n, infinity, GREEDY)
    Pat Repeat(p, n, m) is the same as Pat Repeat(p, n, m, GREEDY)

Pat Span("string")

Function:
    Constructs a pattern that matches one or more (not zero) occurrences of characters in its argument. It is greedy; it always matches the longest possible string. It fails rather than matching zero characters.

Returns:
    A pattern.

Argument:
    string a string.

Pat String("string")

Function:
    Constructs a pattern that matches its string argument.

Returns:
    A pattern.

Argument:
    string a string.
Pat Succeed()

Function:
Constructs a pattern that always succeeds, even when the matcher backs into it. It matches the null string.

Returns:
1 when the match succeeds.

Argument:
none

Pat Tab(int)

Function:
Constructs a pattern that matches forward to position int in the source string. It can match 0 or more characters. It fails if it would have to move backwards or beyond the end of the string.

Returns:
A pattern.

Argument:
int an integer that specifies a position in a string.

Pat Test(expr)

Function:
Constructs a pattern that succeeds and matches the null string if expr is not zero and fails otherwise.

Returns:
A pattern.

Argument:
expr An expression.

Note:
Usually the argument will be wrapped with expr() because the test needs to be made on the current value of variables set by Pat Immediate, Pat Conditional, and Pat At. Without expr, the test is based on values that were known when the pattern was constructed, which means the test will always succeed or always fail at pattern execution time, which is probably not what you want.

Example
nCats = 0;
whichCat = 3;
string = "catch a catnapping cat in a catsup factory";
rc = Pat Match(string, "cat" + Pat Test(expr(nCats=nCats+1; nCats == whichCat)), "dog");
show(rc, string, nCats);

rc:1
string:"catch a catnapping dog in a catsup factory"
nCats:3
Regex Match(source, pattern)

Function:
Executes the pattern match in `pattern` against the quoted string `source`.

Returns:
A pattern.

Argument:
- `source` a string.
- `pattern` a pattern.

Expressions

Arg(expr, i)

Arg Expr(expr, i)

Function:
Finds the argument numbered by `i` within the given expression.

Returns:
The `i`th argument within the expression `expr`.
Empty() if that argument doesn’t exist or is not specified.

Arguments:
- `expr` an expression defined previously in the JSL script.
- `i` an integer denoting which argument to return.

Notes:
`Arg Expr()` was deprecated in a previous release of JMP. Use `Arg()` instead.

Eval(expr)

Function:
Evaluates `expr`, then evaluates the result of `expr` (unquoting).

Returns:
The result of the evaluation.

Argument:
- `expr` Any JSL expression.
Eval Expr(expr)

Function:
Evaluates any expressions within expr, but leaves the outer expression unevaluated.

Returns:
An expression with all the expressions inside expr evaluated.

Argument:
expr Any JSL expression.

Eval Insert(string, <startDel>, <endDel>)

Function:
Allows for multiple substitutions.

Returns:
The result.

Arguments:
string A quoted string with embedded expressions.
startDel Optional starting delimiter. The default value is ^.
endDel optional ending delimited. The default value is the starting delimiter.

Eval Insert Into(string, <startDel>, <endDel>)

Function:
Allows for multiple substitutions in place. The same operation as in Eval Insert is performed, and the result is placed into string.

Returns:
The result.

Arguments:
string A quoted string with embedded expressions.
startDel Optional starting delimiter. The default value is ^.
endDel optional ending delimited. The default value is the starting delimiter.

Eval List(list)

Function:
Evaluates expressions inside list.

Returns:
A list of the evaluated expressions.

Arguments:
list A list of valid JSL expressions.
Expr(x)
Function:
  Returns the argument unevaluated (expression-quoting).
Returns:
  The argument, unevaluated.
Argument:
  x  Any valid JSL expression.

Expr As Picture(expr(...))
Function:
  Converts expr() to a picture as it would appear in the Formula Editor.
Returns:
  Reference to the picture.
Argument:
  expr(...) Place any valid JSL expression that can be displayed as a picture inside expr().

Extract Expr(expr, pattern)
Function:
  Find expr matching pattern.
Returns:
  A pattern that matches the specified pattern.
Arguments:
  expr  Any expression.
  pattern Any pattern.

First(expr, <expr>, ...)
Function:
  Evaluates all expressions provided as arguments.
Returns:
  Only the result of the first evaluated expression.
Arguments:
  expr  Any valid JSL expression.

Head(exprArg)
Head Expr(exprArg)
Function:
  Returns the head of the expression, without its arguments.
Notes:
  Head Expr() is deprecated: use Head() instead.
Head Name(exprArg)

Head Name Expr(exprArg)

Function:
Returns the head of the evaluated expression as a string.

Notes:
Head Name Expr() is deprecated: use Head Name() instead.

Invert Expr(expr, name)

Function:
Attempts to unfold expr around name.

Maximize(expr, {x1, x2, ...}, messages)

Function:
Finds the values for the listed x's that maximize the expr.

Messages:
<<Max Iter(int) An integer that specifies the maximum number of iterations to be performed.
<<Tolerance(p) p is a number between 0 and 1.
<<Method(NR|BFGS) Specify either the Newton-Raphson method or the Quasi-Newton method with BFGS update.
<<Limits(arguments) The arguments specify ranges for each x (for example, x1<3)

Minimize(expr, {x1, x2, ...}, messages)

Function:
Finds the values for the listed x's that minimize the expr.

Messages:
<<Max Iter(int) An integer that specifies the maximum number of iterations to be performed.
<<Tolerance(p) p is a number between 0 and 1.
<<Method(NR|BFGS) Specify either the Newton-Raphson method or the Quasi-Newton method with BFGS update.
<<Limits(arguments) The arguments specify ranges for each x (for example, x1<3)

Name Expr(x)

Function:
Returns the unevaluated expression of x rather than the evaluation of x.
JSL Syntax Reference

SAS Integration

N Arg(exprArg)
N Arg Expr(exprArg)
Function:
Returns the number of arguments to the expression head.
Note:
N Arg Expr() is deprecated: use N Arg() instead.

Simplify Expr(expr(expression))
Simplify Expr(nameExpr(global))
Function:
Algebraically simplifies an expression

Wild()
Function:
Only used with Extract Expr() for expression matching to denote a wildcard position that matches any expression.

Wild List()
Function:
Only used with Extract Expr() for expression matching to denote a series of wildcard arguments that match any expression.

SAS Integration

JMP6 SAS Compatibility Mode(bool)
Function:
Setting this to 1 (true) causes SAS-related operators to operate in a mode compatible with JMP 6 capabilities.

Global Metadata Server

Current Metadata Connection()
Function:
Returns the active SAS metadata connection, if any, as a scriptable object.
Get SAS Version Preference()

Function:
Returns the SAS version selected in Preferences as a string.

Meta Connect(<machine, port>, <authDomain>, <username>,<password>, named arguments)

Meta Connect(<Profile(profile name)>, <Password(password)>, named arguments)

Function:
Connects to a SAS Metadata Server. If no arguments are specified, an empty connection dialog appears. If some arguments are specified, a dialog partially filled in with the argument values appears. If all arguments are specified, the connection is made and no dialog appears.

Returns:
1 if connection is successful, 0 if not.

Arguments:
machine  Optional: quoted string that contains the DNS name of the machine.
port    Required if machine is specified. Quoted string or integer that contains the port on which the metadata server listens.
authDomain Optional: quoted string that contains the authentication domain for the credentials supplied. Not necessary unless username and password are included.
username Optional: quoted string that contains the username for the connection.
password Optional: quoted string that contains the password for the connection.

Named Arguments
All named arguments are optional.

CheckPreferenceOnly If specified, Meta Connect returns the status of the option I want to connect to a SAS Metadata Server in the SAS Integration page of JMP Preferences. If that box is checked, Meta Connect returns 1; if not, 0.

Repository("string") Takes a quoted string that contains the name of the repository to which to connect.

ProfileLookup If machine and port are specified rather than a profile name, and ProfileLookup is specified, an attempt is made to find a metadata server connection profile with a machine name and port matching those provided. If one is found, other connection information (such as authentication domain, user name and password) is obtained from that profile.

Prompt(Always | Never | IfNeeded) Takes one of the keywords Always (always prompt before attempting to connect), Never (never prompt, just fail), or IfNeeded (the default; prompt if connection with the given parameters fails).

Notes:
If no arguments are included, the Connect to SAS Metadata Server window appears.
Meta Create Profile(profile, HostName("hostname"), Port(port), <named arguments>)

Function:
Creates a metadata server connection profile and adds it to the current user's set of saved metadata server connection profiles.

Returns:
1 if profile was successfully created, otherwise 0.

Arguments:

profile A quoted string that contains the name of the created profile. If a profile by the given name already exists, MetaCreateProfile fails unless Replace is specified.

HostName(name) A quoted string that contains the name of the host computer running the SAS Metadata Server that this profile will connect to.

Port(n) The port number (n) that the SAS Metadata Server is listening for connections on.

Named Arguments
The following named arguments are all optional.

AuthenticationDomain("domain") | AuthDomain("domain") A quoted string that sets the authentication domain to use for the connection.

Description("desc") | Desc("desc") A quoted string that sets a description for this profile.

Password("password") A quoted string that contains the password to store in this profile.

Replace If name matches a profile that already exists, Replace must be specified for the existing profile to be replaced by the one provided. The default value is False (0).

Repository("repository") A quoted string that contains the name of the repository in the SAS Metadata Server that this profile will connect to. This option is only valid for connections to SAS 9.1.3 Metadata Servers.

UserName("username") A quoted string that contains the user name that this profile will use to connect to the SAS Metadata Server.

UseSingleSignOn If specified, this profile attempts to use Single Sign-On (currently also known as Integrated Windows Authentication) to connect to the SAS Metadata Server. This option is only valid for connecting to SAS 9.2 Metadata Servers. If UseSingleSignOn is True(1), UserName and Password cannot be specified. The default value is False (0).

Meta Delete Profile(name)

Function:
Deletes the named metadata server connection profile from the current user's set of saved metadata server connection profiles

Argument:

name A quoted string that contains the name of the profile to delete.

Returns:
1 if profile was successfully deleted, otherwise 0.
Meta Disconnect()

Function:
   Disconnect the current SAS Metadata Server connection, if any.

Returns:
   Void.

Arguments:
   None.

Meta Is Connected()

Function:
   Ask whether a current connection to a SAS Metadata Server exists.

Returns:
   1 if a connection exists; 0 otherwise.

Arguments:
   None.

Meta Get Repositories()

Function:
   Gets a list of the repositories available on the current SAS Metadata Server connection.

Returns:
   A list of repository names as strings.

Arguments:
   None.

Meta Get Servers()

Function:
   Get a list of the SAS Servers that are registered in the SAS Metadata Repository to which the session
   is currently connected.

Returns:
   A list of server names as strings.

Arguments:
   None.

Meta Get Stored Process(path)

Function:
   Get a stored process object from the currently connected SAS Metadata Repository.

Returns:
   Stored Process scriptable object.

Arguments:
   path  Quoted string that is the path to the stored process in metadata, starting at the BIP Tree.
Meta Set Repository(repositoryName)

Function:
Set the SAS Metadata Repository to use for metadata searches.

Returns:
1 if setting the repository was successful, 0 otherwise.

Arguments:
repositoryName Quoted string that contains the name of the repository to make current.

Global SAS Server

Current SAS Connection()

Function:
Gets the active global SAS server connection, if any, as a scriptable object.

SAS Connect<name>, <port>, <named arguments>)

Function:
Connect to a local, remote, or logical SAS server.

Returns:
SAS Server scriptable object.

Arguments:
name Optional: quoted string that can contain a physical machine name, in which case port must be provided, or the name of a metadata-defined (logical) server, in which case port must not be provided. If neither name nor port are included, and JMP is running on Windows, a connection to SAS on the local machine (via COM) will be attempted, and all named parameters are ignored.

port Optional: quoted string or integer. If name is a physical machine name, this is the port on that machine to connect to. If name is a metadata-defined (logical) server, port must not be included.

Named Arguments
All the named arguments are optional.

UserName(name) A quoted string that contains the username for the connection.
Password(password) A quoted string that contains the password for the connection.
ReplaceGlobalConnection A boolean. The default value is True. If True, and a successful SAS server connection is made, this connection will replace the active SAS connection that becomes the target of other global SAS JSL function calls. If False, the global SAS connection will not be changed, and the returned SASServer scriptable object should be used to send messages to this server connection.

ShowDialog A boolean. The default value is False. If True, other arguments (except ReplaceGlobalConnection) are ignored and the SAS Server Connection dialog appears. This provides the JSL programmer a way to bring up the SAS Connect dialog.
Prompt(Always|Never|IfNeeded) A Keyword. Always means always prompt before attempting to connect; Never means never prompt even if the connection attempt fails (just fail and send an error message to the log), and IfNeeded (the default value) means prompt if the attempt to connect with the given parameters fails (or is not possible with the information given).

SAS Disconnect()

Function:
Disconnect the active global SAS connection, if any.

Returns:
Void.

Arguments:
None.

SAS Is Connected()

Function:
Check whether there is an active global SAS server connection.

Returns:
1 if an active global SAS connection exists, 0 otherwise.

Arguments:
None.

SAS Submit("sas code", <named arguments>)

Function:
Submit some SAS code to the active global SAS server connection.

Returns:
1 if successful, 0 otherwise.

Arguments:
sasCode Quoted string that contains the SAS code to submit.

Named Arguments:
All named arguments are optional.

Async A boolean. If True (1), the submit occurs asynchronously (in the background). Use the Get Submit Status() message on the SAS Server Scriptable Object to determine the status of the submit. The default value is False (0).

ConvertCustomFormats(bool) A boolean. When SAS data sets generated by submitted SAS code are imported into JMP after the submit completes (see Open Output Datasets), the value of ConvertCustomFormats determines whether an attempt will be made to convert any custom formats found on columns in the SAS data to JMP value labels. The default value is True (1).
DeclareMacros(var1, var2, ...)  JSL variable names. Provides a simple way to pass the values of JSL variables to SAS as macro variables. Each JSL variable specified should evaluate to a string or numeric value.

GetSASLog(<bool|OnError>, <JMPLog|Window>)  A boolean. If no arguments are supplied, the SAS Log is retrieved and displayed in the location indicated in SAS Integration Preferences. The first argument to GetSASLog can be either a Boolean value or the keyword OnError. If a Boolean value is supplied, true means display the SAS Log, and false means not to display it. OnError instructs JMP to only show the SAS Log if an error occurred in the submit. The second argument to GetSASLog tells JMP where to display the SAS Log. If JMPLog is specified, the SAS Log is appended to the JMP Log. If Window is specified, the SAS Log is opened in a separate window.

GraphicsDevice(string) or GDevice(string)  A string that specifies a value for the GDEVICE SAS option to be used for graphics generated by the submitted SAS code. The value must be a valid SAS graphics device. The default value is determined in Preferences.

NoOutputWindow  A boolean. If True, the SAS Output window containing the listing output from the submission does not appear. The default value is False.

ODS(bool)  If true, additional SAS code is submitted causing ODS results to be generated for the submitted SAS code. The default value is determined in Preferences.

ODSFormat(string)  A quoted string that determines the format of generated ODS results. Valid values are “HTML”, “RTF”, and “PDF”. The default value is determined in Preferences.

ODSGraphics(bool)  If true, ODS statistical graphics are generated for the submitted SAS code. Setting ODSGraphics to true causes ODS to also be set to true. The default value is determined in Preferences.

ODSStyle(string)  A quoted string that specifies the ODS Style to use when generating ODS results. String must be a valid SAS Style. The default value is determined in Preferences.

ODSStyleSheet(path)  A quoted string that specifies a local CSS style sheet to use when formatting generated ODS results. Path must be a path to a CSS file valid for the client machine (the machine running JMP). The default value is determined in Preferences.

OnSubmitComplete(script)  A quoted string that specifies a JSL script that should be run when the submit completes. This is especially useful for asynchronous submits. If script is the name of a defined JSL function, that function is executed, with the SAS Server scriptable object passed as the first parameter.

OpenODSResults(bool)  If true, ODS results that are generated by the submitted SAS code (due to ODS being true) are automatically opened after the submit completes. The default value is True (1).

OpenOutputDatasets(<All|None|dataset1, dataset2, ...>)  JMP detects when submitted SAS code creates new SAS data sets. OpenOutputDatasets (which can be abbreviated OutData) determines what, if anything, is done with those data sets with the SAS Submit completes. If All is specified, all data sets generated by the SAS code are imported into JMP when the SAS Submit completes. If None is specified, none of the generated data sets are imported. If there are specific data sets known to be generated by the submitted SAS code that you want to be imported into JMP when the SAS submit completes, you can alternative provide their names, and only the requested data sets will be imported. The default value is determined in Preferences.
Title(string) A quoted string that specifies the window title to use for the window that displays Listing output from the submit.

SAS Submit File(filename, <named arguments>)

Function:
Submit a SAS code file to the active global SAS server connection.

Returns:
1 if successful; 0 otherwise.

Arguments:
filename Quoted string that contains the name of file containing SAS code to submit.

Named Arguments
Same as for SAS Submit.

SAS Get Output()

Function:
Retrieve the listing output from the last submission of SAS code to the current global SAS server connection.

Returns:
A string.

Arguments:
None.

SAS Get Log()

Function:
Retrieve the SAS Log from the active global SAS server connection.

Returns:
A string.

Arguments:
None.

SAS Get Lib Refs(<named arguments>)

Function:
Get a list of the currently defined SAS librefs on the current global SAS server connection.

Returns:
List of strings.

Named Arguments:
Friendly Names Optional: Boolean. If True, then for any libraries that have friendly names (metadata-defined libraries), the friendly name will be returned rather than the 8-character libref.
SAS Assign Lib Refs(libref, path, <engine>, <engine options>)

Function:
Assign a SAS libref on the active global SAS server connection.

Returns:
1 if successful, 0 otherwise.

Arguments:
libref Quoted string that contains a library reference (8-character maximum) to assign.
path Quoted string that contains the full path on the SAS server to the library being assigned.
engine Optional: quoted string that contains the engine for the SAS server to use when accessing members of this library.
engine options Optional: quoted string that contains the options needed for the engine being used.

SAS Deassign Lib Refs(libref)

Function:
De-assign a SAS libref on the active global SAS server connection.

Returns:
1 if successful; 0 otherwise.

Arguments:
libref Quoted string that contains the library reference to de-assign.

SAS Get Data Sets(libref)

Function:
Returns a list of the data sets defined in a SAS library.

Returns:
List of strings.

Arguments:
libref Quoted string that contains the SAS libref or friendly library name associated with the library for which the list of defined SAS data sets will be returned.
**SAS Get Var Names**

**Function:**
Retrieves the variable names contained in the specified data set on the current global SAS server connection.

**Returns:**
List of strings.

**Arguments:**

- **string**
  A quoted string that contains one of the following:
  - The name of the SAS Library containing the SAS data set to be imported, in which case the *dataset name* parameter is required.
  - The full member name of the SAS data set to be imported, in the form "libname.member-name".
  - The SAS Folders tree path to a logical SAS data table to be imported. This option requires a connection to a SAS 9.2 Metadata Server.

- **dataset**
  Optional: quoted string that contains the name of the data set from which to retrieve variable names.

- **password("password")**
  Optional: quoted string that contains the read password for the data set. If this is not provided and the data set has a read password, the user will be prompted to enter it.

**SAS Get Results()**

**Function:**
Retrieve the results of the previous SAS Submit as a scriptable object, which allows significant flexibility in what to do with the results.

**Returns:**
A SAS Results Scriptable object.

**SAS Import Data**

**Function:**
Import a SAS data set from the active global SAS server connection into a JMP table.

**Returns:**
JMP Data Table object.

**Arguments:**

- **string**
  A quoted string that contains one of the following:
  - The name of the SAS Library containing the SAS data set to be imported, in which case the *dataset name* parameter is required.
  - The full member name of the SAS data set to be imported, in the form "libname.member-name".
  - The SAS Folders tree path to a logical SAS data table to be imported. This option requires a connection to a SAS 9.2 Metadata Server.
dataset  Optional: quoted string that contains the name of the data set from which to retrieve variable names.

Named Arguments
All named arguments are optional.

Columns(list) or Columns(col1, col2, ...) A quoted string list or multiple strings that contain the names of columns to include in the import.

ConvertCustomFormats  The default value is True (1). If True and custom formats are found in the SAS data set being imported, an attempt will be made to convert the SAS custom formats to JMP value labels for those columns.

Invisible(bool)  The default value is False (0). If true, the JMP data table is created as a hidden table. Hidden data tables that never become visible or linked to an analysis remain in memory until they are explicitly closed, reducing the amount of memory that is available to JMP. To explicitly close the hidden data table, call Close(dt), where dt is the data table reference returned by SASImportData.

Where(filter) A quoted string that contains the filter to use when importing data, as in Where("salary<50000").

Password(password) A quoted string that contains the read password for the data set. If this is not provided and the data set has a read password, the user will be prompted to enter it.

UseLabelsForVarNames(bool) If True, the labels from the SAS data set will become the column names in the resulting JMP table. If False, the variable names from the SAS data set will become the column names in the JMP table. The default value is True.

Sample(named arguments) optional, named. Allows a random sample of the SAS data set to be imported into JMP. If both Where and Sample are specified, the Where clause is used to filter the SAS data set first, and then a random sample of the resulting rows is taken based on the parameters supplied to Sample. Note that Sample uses PROC SURVEYSELECT on the SAS server, which is only available if the SAS/STAT package is licensed and installed on that server. Documentation of PROC SURVEYSELECT may be helpful in understanding how sampling is
performed. By default (if no parameters are supplied), a 5% simple random sample is taken. Available parameters (all optional) to Sample are as follows:

- **Simple | Unrestricted**: If `Simple` is specified, sampling is performed without replacement. If `Unrestricted` is specified, sampling is performed with replacement. These two options are mutually exclusive and only one may be specified.

- **SampleSize(int) or N(int)**: Total number of rows for the sample, or number of rows per strata level for stratified sampling.

- **SampleRate(number) or Rate(number) or Percent(number)**: Specifies the sampling rate. For stratified sampling, the rate is applied to each strata level. Note that the supplied value is assumed to be a percentage, so `SampleRate(3.5)` means a 3.5% sampling rate.

- **Strata({col1, col2, ...}) or Strata(col1, col2, ...)**: Perform stratified random sampling using the column names supplied as Strata variables.

- **NMin(int)**: Minimum number of rows (either overall or per strata level for stratified sampling) to return. Only applies to rate-based sampling.

- **NMax(int)**: Maximum number of rows (either overall or per strata level for stratified sampling) to return. Only applies to rate-based sampling.

- **Seed(int)**: Number to use as the seed for sampling. Useful for replicating the same sample. By default, the seed is a random number based on time of day. See PROC SURVEYSELECT documentation for more information.

- **OutputHits(bool)**: Boolean; the default value is false. When doing Unrestricted sampling, if the same row of the input data set is selected more than once, by default that row will still only appear once in the resulting JMP data table, with the NumberHits column indicating the number of times that the row was selected. Setting OutputHits to true causes an input row that is selected multiple times to appear multiple times in the resulting JMP data table.

- **SelectAll(bool)**: Boolean, the default value is true. If `SelectAll` is true, PROC SURVEYSELECT will select all stratum rows whenever the stratum sample size exceeds the total number of rows in the stratum. If `SelectAll` is false and PROC SURVEYSELECT finds a case where the stratum sample size exceeds the total number of rows in a given stratum, an error results and sampling fails. `SelectAll` only applies to Simple random sampling.

- **SQLTableVariable(bool)** If `True`, a SQL table variable will be created in the resulting JMP table that shows the SQL that was submitted to SAS to obtain the data. If `False`, the SQL table variable will not be created. The default value is `True`. If a SQL table variable is created and the data set required a read password, the password will be masked.
**SAS Import Query(sqlquery, <named arguments>)**

**Function:**
Execute the requested SQL query on the current global SAS server connection, importing the results into a JMP data table.

**Returns:**
JMP Data Table object.

**Arguments:**
- sqlquery: Quoted string that contains the SQL query to perform and from which to import the result.
- Named Arguments:
  - ConvertCustomFormats: The default value is true. If true and custom formats are found in the SAS data set being imported, an attempt will be made to convert the SAS custom formats to JMP value labels for those columns.
  - Invisible(bool): The default value is false. If true, the JMP data table is created hidden. Hidden data tables that never become visible or linked to an analysis remain in memory until they are explicitly closed, reducing the amount of memory that is available to JMP. To explicitly close the hidden data table, call `Close(dt)`, where `dt` is the data table reference returned by `SAS Import Query`.
  - UseLabelsForVarNames(bool): The default value is true. If True, the labels from the SAS data set will become the column names in the resulting JMP table. If False, the variable names from the SAS data set will become the column names in the JMP table.
  - SQLTableVariable(bool): The default value is true. If True, a SQL table variable will be created in the resulting JMP table that shows the SQL that was submitted to SAS to obtain the data. If False, the SQL table variable will not be created. If a SQL table variable is created and the data set required a read password, the password will be masked.

**SAS Get File(source, dest)**

**Function:**
Get a file from the active global SAS server connection.

**Returns:**
Void.

**Arguments:**
- source: Quoted string that contains the full path of file on the server to be downloaded to the client machine.
- dest: Quoted string that contains the full path on the client machine for where to put the copy of the file downloaded from the server.
SAS Send File(source, dest)

Function:
Send a file from the client machine to the active global SAS server connection.

Returns:
Void.

Arguments:
source Quoted string that contains the full path of the file on the client machine to be uploaded to the server.
dest Quoted string that contains the full path on the server that will receive the file uploaded from the client machine.

SAS Get File Refs()

Function:
Get a list of the currently defined SAS filerefs on the active global SAS server connection.

Returns:
List of two lists. The first list is a list of quoted strings of fileref names. The second is a corresponding list of quotes strings of physical names.

Arguments:
None.

SAS Get File Names(fileref)

Function:
Get a list of filenames found in the given fileref on the active global SAS server connection.

Returns:
List of strings.

Arguments:
fileref Quoted string that contains the name of the fileref from which to retrieve filenames.

SAS Get File Names In Path(path)

Function:
Get a list of filenames found in the given path on the active global SAS server connection.

Returns:
List of strings.

Arguments:
path Quoted string that contains the directory path on the server from which to retrieve filenames.
SAS Load Text File(path)

Function:
Download the file specified in path from the active global SAS server connection and retrieve its contents as a string.

Returns:
String.

Arguments:
path Quoted string that contains the full path on the server of the file to download and retrieve the contents as a string.

SAS Export Data(dt, library, dataset, <named_arguments>)

Function:
Exports a JMP data table to a SAS data set in a library on the active global SAS server connection.

Returns:
1 if the data table was exported successfully; 0 otherwise.

Arguments:
dt data table or a reference to a data table.
library the library to which to export the data table.
dataset the name of the new SAS data set.

Named Arguments:
Note: all the named arguments are optional.
Columns A list of columns or a comma-separated list of columns.
Password A string that contains the password to serve as the READ, WRITE, and ALTER password for the exported SAS data set. If the exported data set is replacing an existing data set with an ALTER password, this password will be used as the ALTER password for overwriting the data set. If Password is specified, values for ReadPassword, WritePassword, and AlterPassword are ignored.
ReadPassword A string that contains the password to serve as the READ password for the exported SAS data set.
WritePassword A string that contains the password to serve as the WRITE password for the exported SAS data set.
AlterPassword A string that contains the password to serve as the ALTER password for the exported SAS data set. If the exported data set is replacing an existing data set with an ALTER password, this password will be used as the ALTER password for overwriting the data set.
PreserveSASColumnNames A boolean. If true and the JMP data table originally came from SAS, the original SAS column names are used in the exported SAS data set. The default value is true.
PreserveSASFormats A boolean. If true and the JMP data table originally came from SAS, the original SAS formats and informats are applied to the columns in the exported SAS data set. The default value is true.
ReplaceExisting A boolean. If true, an existing SAS data set with the specified name in the specified library will be replaced by the exported SAS data set. If false, a data set with the specified name already exists in the specified library; the export will be aborted. The default value is false.

HonorExcludedRows(bool) A boolean. If true, any rows in the JMP data table that are marked as excluded are not exported. The default value is false.

Note: Information on the export is sent to the log.

SAS Name(name)
SAS Name({list of names})

Function: Converts JMP variable names to SAS variable names by changing special characters and blanks to underscores and various other transformations to produce a valid SAS name.

Returns: A valid SAS name or a list of valid SAS names.

Argument: name A quoted string that represents a JMP variable name; or a list of quoted JMP variable names.

SAS Open For Var Names(path)

Function: Opens a SAS data set only to obtain the names of its variables, returning those names as a list of strings.

Returns: A list of variable names in the file.

Argument: path A quoted string that is a pathname of a SAS data set.

Messages for Metadata Server Objects

metaserver << Disconnect()

Function: Disconnects the metadata server.

Returns: Void.
metaserver << Get Display Name()

Function:
   Gets the display name of the metadata server.

Returns:
   A string.

metaserver << Get Host Name()

Function:
   Gets the host (machine) name of the metadata server.

Returns:
   A string.

metaserver << Get Port()

Function:
   Gets the port used for the metadata server connection.

Returns:
   An integer.

metaserver << Get Use Identity()

Function:
   Gets the identify of the connected user as defined in metadata.

Returns:
   A string.

metaserver << Get User Name()

Function:
   Gets the user name (login ID) that was used for the metadata server connection.

Returns:
   A string.
Messages for SAS Server Objects

sasconn << Get Submit Status()

Function:  
Gets the current status of a SAS Submit for this server that is presumably running asynchronously.

Arguments:
None

Returns:
1 if the submit has not started; 2 if the submit is running; 3 if the submit has been canceled; 10 if the submit has completed successfully; 11 if the submit has completed with errors.

sasconn << Cancel Submit()

Function:  
Cancels the currently-running SAS Submit for this server that is presumably running asynchronously.

Arguments:
None

Returns:
1 if a running submit was found and canceled; 0 otherwise.

sasconn << Get Error Count()

Function:  
Gets the count of the number of errors encountered in the previous SAS Submit.

Arguments:
None

Returns:
An integer.

sasconn << Get Results()

Function:  
Retrieve the results of the previous SAS Submit as a scriptable object, which allows significant flexibility in what to do with the results. See the SAS Results Scriptable Object for details.

Arguments:
None

Returns:
A SAS Results Scriptable Object.
\begin{verbatim}
sasconn << Get Work Folder()
Function:
Returns the full path of the folder/directory corresponding to the WORK library for this server.
Arguments:
None
Returns:
A string that contains the work folder/directory path.
\end{verbatim}

\begin{verbatim}
sasconn << Get Version()
Function:
Returns the SAS version as a string such as “9.1” or “9.2”.
Arguments:
None
Returns:
A string that contains the SAS version.
\end{verbatim}

\begin{verbatim}
sasconn << Open SAS Results()
Function:
Open the results from the previous SAS Submit. Intended to be used with asynchronous SAS submits or the use of the OnSubmitComplete option to SAS Submit to give the JSL author a way to conditionally open the results of a submit.
Arguments:
None
Returns:
Void.
\end{verbatim}

\begin{verbatim}
sasconn << Open Log Window()
Function:
Opens, or pops to the top of the Z-order if already open, the SAS Log window for this server.
Arguments:
None
Returns:
Void.
\end{verbatim}
sasconn << Open Output Window()

Function:
    Opens, or pops to the top of the Z-order if already open, the SAS Output window for this server.

Arguments:
    None.

Returns:
    Void.

sasconn << Clear Log History()

Function:
    Clears the SAS Log history for this server.

Arguments:
    None

Returns:
    Void.

sasconn << Clear Output History()

Function:
    Clears the SAS Output history for this server.

Arguments:
    None

Returns:
    Nothing

sasconn<<Connect(<named arguments>)

Function:
    Attempt to reconnect a SAS server connection object that has become disconnected.

Returns:
    1 if the connection was successful, 0 otherwise.

Named Arguments:
    All named arguments are optional.

    UserName(name)  A quoted string that contains the username for the connection.

    Password(password) A quoted string that contains the password for the connection.

    Prompt(Always | Never | IfNeeded) A keyword. Always means always prompt before attempt to connect; Never means never prompt even if the connection attempt fails (just fail with an error message going to the log), and IfNeeded (the default) means prompt if the attempt to connect with the given parameters fails (or is not possible with the information given).
sasconn<<Disconnect()

Function:
    Disconnect this SAS server connection.

Returns:
    Void.

Arguments:
    None.

sasconn<<Export Data(dt, library, dataset, <named arguments>)

Function:
    Exports a JMP data table to a SAS data set in a library on the current global SAS server connection.

Returns:
    1 if the data table was exported successfully; 0 otherwise.

Arguments:
    See the entry for SAS Export Data on p. 612.

sasconn<<Is Connected()

Function:
    Determine whether this SAS Server object is currently connected to SAS.

Returns:
    1 if sasconn is connect, 0 otherwise.

Arguments:
    None.

sasconn<<Submit(sasCode, <named arguments>)

Function:
    Submit some SAS code to this SAS server connection.

Returns:
    Void.

Arguments:
    See SAS Submit().

sasconn<<Submit File(filename, <named arguments>)

Function:
    Submit a SAS code file to this SAS server connection.

Returns:
    Void.

Arguments:
    See SAS Submit File().
**sasconn<<Get Output()**

*Function:* Retrieve the listing output from the last submission of SAS code to this SASServer object.

*Returns:* String.

*Arguments:* None.

**sasconn<<Get Log()**

*Function:* Retrieve the SAS Log from the last SAS Submit from this SAS server connection.

*Returns:* String.

*Arguments:* None.

**sasconn<<Get Librefs(<named arguments>)**

*Function:* Get a list of the currently defined SAS librefs on this SAS server connection.

*Returns:* List of strings.

*Arguments:* See SAS *Get Librefs* global JSL function above.

**sasconn<<Assign Libref(libref, path, engine, engine options)**

*Function:* Assign a SAS libref on this SAS server connection.

*Returns:* Void.

*Arguments:* See SAS *Assign Libref* global JSL function above.

**sasconn<<Deassign Libref(libref)**

*Function:* De-assign a SAS libref on this SAS server connection.

*Returns:* Void.

*Arguments:* libref Quoted string that contains the library reference to de-assign.
sasconn<<Get Data Sets(libref)

Function:
Returns a list of the data sets defined in a SAS library on this SAS server connection.

Returns:
List of strings.

Arguments:
libref  Quoted string that contains the SAS libref or friendly library name associated with the library for which the list of defined SAS data sets will be returned.

sasconn<<Get Var Names(libref, dataset, <named arguments>)

Function:
Retrieves the variable names contained in the specified data set on the current global SAS server connection.

Returns:
List of strings.

Arguments:
See SAS Get Var Names global JSL function above.

sasconn<<Import Data(library, dataset, <named arguments>)

Function:
Import a SAS data set from this SAS server connection into a JMP table.

Returns:
JMP Data Table object.

Arguments:
See SAS Import Data global JSL function above.

sasconn<<Import Query(sqlquery, <named arguments>)

Function:
Execute the requested SQL query on this SAS server connection, importing the results into a JMP data table.

Returns:
JMP data table object.

Arguments:
See SAS Import Query global JSL function above.
sasconn<<Get File(source, dest)
Function:
    Download a file from this SAS server connection.
Returns:
    Void.
Arguments:
    See SAS Get File global JSL function above.

sasconn<<Send File(source, dest)
Function:
    Upload a file to this SAS server connection.
Returns:
    Void.
Arguments:
    See SAS Send File global JSL function above.

sasconn<<Get File Refs()
Function:
    Get a list of the currently defined SAS filerefs on this SAS server connection.
Returns:
    List of strings.
Arguments:
    None.

sasconn<<Get File Names(fileref)
Function:
    Get a list of filenames found in the given fileref on this SAS server connection.
Returns:
    List of strings.
Arguments:
    fileref  Quoted string that contains the name of fileref from which to retrieve filenames.

sasconn<<Get File Names In Path(path)
Function:
    Get a list of filenames found in the given path on this SAS server connection.
Returns:
    List of strings.
Arguments:
    path  Quoted string that contains the directory path on the server from which to retrieve filenames.
sasconn<<Load Text File(path, <named arguments>)

Function:
Download the file specified in path from the active global SAS server connection and retrieve its contents as a string.

Returns:
String.

Arguments:
See SAS Load Text File global JSL function above.

Messages for Stored Processes

stp<<Run(<named arguments>)

Function:
Execute this stored process object in the foreground.

Returns:
An integer; stored process execution status; see the values for Get Status.

Arguments:
AutoOpenResults Optional: Boolean. If True, results are automatically opened when the stored process completes. If False, results are not auto-opened, and can be manually opened via the object returned by the GetResults message. The default value is True.

UserName(username) Optional: quoted string that contains the username under which to run the stored process.

Password(password) Optional: quoted string that contains the password for UserName.

AuthDomain(authDomain) Optional: quoted string that contains the authentication domain of the credentials (username, password) given.

ODSDest(dest) Optional: quoted string that contains an ODS destination (HTML, PDF, tagsets.SASReport12) for any ODS-generated results from the stored process. This requires the stored process SAS code to call %STPBEGIN. The default value is HTML.

GraphicsDevice(device) Optional: quoted string that contains the SAS graphics device to use when generating graphics in ODS results. This requires the stored process SAS code to call %STPBEGIN. The default value is GIF.

ODSStyle(styleName) Optional: quoted string that contains an ODS style to apply to the results. This requires the stored process SAS code to call %STPBEGIN. There is no default value.

ODSStyleSheet(cssFile) Optional: quoted string that contains the full path to a CSS file on the client machine that is to be applied to generated ODS results. This requires the stored process SAS code to call %STPBEGIN. There is no default value.

NoAlerts Optional: Boolean. If True, error messages will be sent to the JMP log rather than message boxes. The default value is False.
stp<<Begin Run(<named arguments>)

Function:
Start this stored process executing in the background. This message is paired with End Run, which should also be called.

Returns:
An integer; stored process execution status; see the values for Get Status. At some point after Begin Run to wait for the stored process to complete.

Arguments:
Same as Run, except AutoOpenResults and NoAlerts are not supported. They are available on EndRun.

AutoResume(<filename>) Optional: quoted string. If specified with no argument, it specifies that the stored process results should be auto-opened when the stored process completes. If filename is specified, filename is opened rather than all results of the stored process being auto-opened.

AutoResumeScript(script) Optional: quoted string that specifies that after stored process execution completes, script should be evaluated. If the script is a function taking at least one argument, the function is evaluated with the scriptable stored process object passed as the first (and only) parameter. AutoResume and AutoResumeScript are mutually exclusive.

stp<<End Run(<named arguments>)

Function:
Wait a specified amount of time (or forever) for a stored process started with Begin Run to complete. If the stored process *is* complete, retrieve the results and optionally open them.

Returns:
An integer; stored process execution status; see the values for Get Status.

Arguments:
AutoOpenResults Optional: Boolean. If True, results are automatically opened if the stored process completes in the time specified by MaxWait. If False, results are not automatically opened, and can be manually opened via the object returned by the Get Results message. Default is True.

MaxWait Optional: an integer. Specifies the maximum amount of time in milliseconds to wait for the stored process to complete. If MaxWait is not specified, End Run will wait forever for the stored process to complete.

NoAlerts Optional: Boolean. If True, error messages will be sent to the JMP log rather than message boxes. The default value is False.
**Stp<<Get Name()**

**Function:**
- Returns the name of the stored process.

**Returns:**
- String.

**Arguments:**
- None.

**Stp<<Get Metadata Id()**

**Function:**
- Returns the metadata ID of the stored process.

**Returns:**
- String.

**Arguments:**
- None.

**Stp<<Get Metadata Path()**

**Function:**
- Returns the full metadata path of the stored process.

**Returns:**
- String.

**Arguments:**
- None.

**Stp<<Get Param Names(<named arguments>)**

**Function:**
- Get a list of parameter names for this stored process, optionally of specific types.

**Returns:**
- List of strings.

**Arguments:**

- **Visible** Optional: Boolean. If `True`, get only visible parameters. If `False`, get only non-visible parameters. If not specified, get both visible and non-visible parameters.

- **Modifiable** Optional: Boolean. If `True`, get only modifiable parameters. If `False`, get only non-modifiable parameters. If not specified, get both modifiable and non-modifiable parameters.

- **Required** Optional: Boolean. If `True`, get only required parameters. If `False`, get only non-required parameters. If not specified, get both required and non-required parameters.

- **Expert** Optional: Boolean. If `True`, get only expert parameters. If `False`, get only non-expert parameters. If not specified, get both expert and non-expert parameters.
stp<<Get Param Value(name)
Function:
Get the current value of the specified parameter.
Returns:
String.
Arguments:
name Quoted string that contains the name of the parameter whose value to retrieve.

stp<<Set Param Value(name, value)
Function:
Set the value of the specified stored process parameter to the specified value.
Returns:
1 if successful, 0 otherwise (value may violate the parameter's constraints).
Arguments:
name Quoted string that contains the name of the parameter whose value to set.
value Quoted string that contains the value to which to set the parameter.

stp<<Reset Param Values()
Function:
Reset all parameter values to their metadata-defined default values.
Returns:
Void.
Arguments:
None.

stp<<Edit Param Values()
Function:
Invoke the stored process parameter dialog box for interactively setting parameter values.
Returns:
Void.
Arguments:
None.
stp<<Get Param Enum Values(name)

Function:
Get the possible enumerated values for a parameter.

Returns:
List of strings.

Arguments:
name Quoted string that contains the name of the parameter whose possible enumerated values to retrieve.

stp<<Get Param Enum Labels(name)

Function:
Get the enumeration labels for a parameter.

Returns:
List of strings.

Arguments:
name Quoted string that contains the name of the parameter whose enumeration labels to retrieve.

stp<<Get Status()

Function:
Get the execution status of the stored process.

Returns:
-1 = execution failed.
1 = not started.
2 = running.
3 = cancelled.
10 = completed successfully.
11 = completed with errors.

Arguments:
None.

stp<<Get Status Message()

Function:
Get the message associated with the failure of the stored process, if any.

Returns:
String.

Arguments:
None.
stp<<Set Results Directory(directory)

Function:
Set the directory on the client machine where stored process results will be placed.

Returns:
String.

Arguments:
directory  Quoted string that contains the full path of the directory where results of the stored process execution should be placed. The directory must exist or be creatable. If the results directory is not set, a temporary location appropriate for the operating system will be used, and that directory can be retrieved from the stored process Results scriptable object after the stored process executes.

stp<<Get Results()

Function:
Get the results generated by the execution of this stored process as a scriptable object.

Returns:
SAS Results scriptable object.

Arguments:
None.

stp<<Delete Results(<named arguments>)

Function:
Delete all results from the execution of this stored process.

Returns:
1 if deletion is successful, 0 otherwise (error message to JMP log).

Arguments:
NoAlerts  Optional: Boolean. If True, the user is not prompted for confirmation before the attempt is made to delete results.

Messages for SAS Results

results << Delete All Result Files()

Function:
Deletes all files created by the SAS Submit or Stored Process execution. Note that any result files that are still in use will not be deleted.

Arguments:
None

Returns:
1 if the deletion was successful; 0 if some of the files could not be deleted.
results<<Get Log()

Function:
Get the SAS Log from the execution of the stored process or SAS submit.

Returns:
String.

Arguments:
None.

results<<Get Output()

Function:
Gets the SAS Listing output from the execution of the stored process or SAS submit.

Returns:
String.

Arguments:
None.

results<<Open All Results()

Function:
Opens all results generated by the execution of the stored process or SAS submit.

Returns:
Void.

Arguments:
None.

results<<Get Directory()

Function:
Gets the directory where the results generated by the stored process or SAS submit are located.

Returns:
String.

Arguments:
None.

results<<Get Main Result File Name(<named arguments>)

Function:
Gets the full path of the main result file generated by the stored process or SAS submit.

Returns:
String.

Arguments:
FullPath  Optional: Boolean. If True, the main result file name is returned as a full path. The default value is False.
results << Get Output Datasets()

Function:
Get a list of output data set generated by the SAS Submit that created this SAS Results object.

Arguments:
None

Returns:
A list of data set names in the form “libname.membername”.

results << Get Result File Info(<MIMEType("mime-type")), <FullPath>)

Function:
Get information about result files that were generated by the execution of the stored process or SAS submit.

Returns:
List of two lists of strings. The first list is filenames, and the second list is the MIME-type of the corresponding file from the first list.

Arguments:

MIMEType(mime_type) Optional: quoted string that restricts the set of files for which information is returned to only those files with the specified MIME-type. If not specified, information about all generated files is returned.

FullPath Optional: Boolean. If True, the filename returned for each result file is returned as a full path; if False, only the name of the file is returned. The default value is False.

results << Open Result File(filename, <named arguments>)

Function:
Attempts to open the result file with the given name.

Returns:
JMP Data Table if one was opened.

Arguments:

filename Quoted string that contains the name of the file from the generated results to open. filename should just be the name of the file, not the full path. If filename is a filename with no extension, both JMP data tables and JSL scripts in the results will be searched for a match, and if both exist, both will be opened.

RunScript Optional: Boolean. If True, and if filename is a JSL script, the script is executed. If False, filename is just opened, even if it is JSL.
results<<Run_Script(filename)

Function:
Looks for a JSL file in the results with the given file name and runs it if it finds it.

Returns:
Void.

Arguments:

filename Quoted string that contains the name of the JSL file from the generated results to open.
Filename should just be the name of the file, not the full path, and it does not need to include the .jsl extension.

Tools for Production Settings

Datafeed

Datafeed()

See
“Open Datafeed,” p. 630

Open Datafeed()
Datafeed()

Function:
Creates a datafeed object and window.

Returns:
A reference to the datafeed object.

Arguments:
None are required. You usually set up the basic operation of the data feed within the Open Datafeed() command, however.

Messages for Datafeed

feed<<Close
Closes the datafeed object and its window.

feed<<Connect(portSettings)
(Windows only) Sets up port settings for the connection to the device.
feed<<Disconnect

(Windows only) Disconnects the device from the datafeed queue but leaves the datafeed object active.

feed<<Get Line

Returns and removes one line from the datafeed queue.

feed<<Get Lines

Returns as a list and removes all lines from the datafeed queue.

feed<<Queue Line(string)

Sends one line to the end of the datafeed queue.

feed<<Restart

Restarts processing queued lines.

feed<<Set Script(script)

Assigns the script that is run each time a line of data is received.

feed<<Show Window(bool)

Specifies whether to show a status window for the data feed.

feed<<Stop

Stops processing queued lines.

Sockets

Socket(<STREAM | DGRAM>)

Function:
Creates a socket.

Returns:
The socket that was created.

Arguments:
STREAM | DGRAM Optional argument to specify whether the socket is a stream or datagram socket. If no argument is supplied, a stream socket is created.
Messages for Sockets

_**skt<<Connect(socketname, port)**_

**Function:**
Connects to a listening socket.

**Returns:**
A list of two strings. The first string is the command name ("connect") and the second is "ok" for a successful connection or an error sent back by the other socket.

**Arguments:**
- `socketname` the name of the other socket. If you're connecting to a web server, this is the webpage address (the name is preferred to the IP address).
- `port` the port of the other socket to connect through.

_**skt<<Close()**_

**Function:**
Closes a socket.

**Returns:**
A list of two strings. The first string is the command name ("close") and the second is "ok" if successful.

**Arguments:**
None.

_**skt<<Send(stream)**_
_**skt<<SendTo(dgram)**_

**Function:**
Sends the data in the argument to the other socket. **Send** sends a stream and **sendto** sends a datagram.

**Returns:**
A list of three strings. The first string is the command name ("send" or "sendto"). The second is "ok" if successful or an error message if not. The third string is any portion of the stream that could not be sent, or empty if all the data was sent correctly.

**Arguments:**
- `stream` the command to send to the other socket.
- `dgram` the command to send to the other socket.

**Note:**
Either argument may need to contain binary data. JMP represents non-printable ASCII characters with a tilde (~) followed by the hex number. For example,  

```  
skt<<send("GET / HTTP/1.0~0d~0a~0d~0a");  
```

sends a “get request” to an HTTP server.
skt<<recv(n, <callback, timeout>)
skt<<recvfrom(n, <callback, timeout>)

Function:
Receives either a stream message (recv) or a datagram message (recvfrom) from the other socket. If the two optional arguments are used, the data is not received immediately. Instead, the data is received when the function callback is called.

Returns:
A list of three strings. The first string is the command name ("recv" or "recvfrom"). The second is "ok" if successful or an error message if not. The third string is the data that was received. If a callback function is used, a fourth element is the socket that was used in the original recv or recvfrom message.

Arguments:
- n specifies the number of bytes to receive from the other socket.
- callback an optional argument that specifies the name of a function to receive the data.
- timeout if you use a callback, timeout specifies how long the function should wait for an answer.

skt<<ioctl(FIONBIO, 1)

Function:
Controls the socket’s blocking behavior.

Returns:
A list of two strings. The first string is the command name ("ioctl") and the second is "ok" if successful or an error.

Arguments:
- FIONBIO, 1 FIONBIO means Non-Blocking I/O. 1 turns on the behavior and the argument.

skt<<bind("localhost", port)

Function:
Associates a port on the local machine with the socket.

Returns:
A list of two strings. The first string is the command name ("bind") and the second is "ok" if successful or an error.

Argument:
- localhost Specifies the local machine. You cannot bind to another machine.
- port The port that should be used.
skt<<Listen()

Function:
Tells the server socket to listen for connections.

Returns:
A list of two strings. The first echoes the command ("listen") and the second is "ok" or an error message.

Arguments:
N/A

skt<<Accept(<callback, timeout>)

Function:
Tells the server socket to accept a connection and return a new connected socket.

Returns:
A list of up to four items. The first is a string that echoes the command ("accept"). The second is a string—either "ok" or an error. The third is a string that specifies the name of the machine that just connected. The fourth is a reference to the socket that you can send more messages.

Arguments:
callback an optional argument that specifies the name of a function to receive the data.
timeout if you use a callback, timeout specifies how long the function should wait for an answer. For a server socket, 0 is an acceptable values because a server shouldn't shut down because no one has connected to it recently.

skt<<GetPeerName()

Function:
Retrieves the address and port of the socket at the other end of the connection.

Returns:
A list of four strings. The first echoes the command ("getpeername"). The second is either "ok" or an error. The third and fourth are the address and the port.

Arguments:
N/A

skt<<GetSockName()

Function:
Retrieves the address and port of the socket at this end of the connection.

Returns:
A list of four strings. The first echoes the command ("getsockname"). The second is either "ok" or an error. The third and fourth are the address and the port.

Arguments:
N/A
Databases

Open Database("connectInfo", "sqlStatement", "outputTableName")

Function:
Opens the database indicated by connectInfo with the sqlStatement, returning a data table named outputTableName.

Message for Databases

dt << Save Database("connectInfo", "TableName")

Saves a data table to a database.

Scheduling

Schedule(n, script)

Function:
Queues an event to run the script after n seconds.

Message for Schedule

sch<<Clear Schedule()

Cancels all events in a schedule queue.
Appendix B

Glossary

Terms, concepts, and placeholders

In syntax summaries, | means “or” and separates possible choices. Usually choices separated by | are mutually exclusive, i.e. you have to pick one and cannot list several.

**argument**  An argument is something specified inside the parentheses of a JSL operator, function, message, etc., e.g. the 10 in Log(10) or the color name "red" in pen color("red"). See also named argument.

**bool**  In syntax summaries, a placeholder for a Boolean argument, e.g. 1 or 0.

**boolean**  A boolean is a yes/no thing—something that is on or off, shown or hidden, true or false, 1 or 0, yes or no. An operator listed as being a boolean operator is one that evaluates to true or false (or missing).

**col**  In syntax summaries, a placeholder for any reference to a data table column, e.g. Column("age").

**command**  A generic description for a JSL statement that performs an action. This book prefers the more specific terms operator, function, and message when they are applicable.

**current data table**  The current data table is the data table that Current Data Table() either returns or is assigned.

**current row**  The current row for scripting is defined to be zero (no row) by default. You can set a current row with Row() or For Each Row, etc.

**database**  Although the term is much more general, for JMP’s purposes, the word “database” describes any external data source (such as SQL) accessed through ODBC with JSL’s Open Database command.

**datafeed**  A datafeed is a method to read real-time data continuously, such as from a laboratory measurement device connected to a serial port.

**db**  In syntax summaries, a placeholder for any reference to a display box, e.g. report(Bivariate[1]).

**dt**  In syntax summaries, a placeholder for any reference to a data table, e.g. Current Data Table() or Data Table("Big Class.jmp").

**eliding operator**  An eliding operator is one that causes arguments on either side to combine and evaluate differently than if the statement were evaluated strictly left to right. For example, 12<a<13 is a range check to test whether a is between 12 and 13; JMP reads the whole expression before evaluating. If < did not elide, the expression would be evaluated left to right as (12<a)<13; in other words, it would check whether the result of the comparison (1 or 0, for false or true) is below 13, which of course would always yield 1 for true. The << operator (for object<<message, which is equivalent to Send(object<<message)) is another example of an eliding operator.
A function takes an argument or series of arguments inside parentheses after the function name. For example, the infix operator + has a function equivalent `Add()`. The statements `3 + 4` and `Add(3, 4)` are equivalent. All JSL's operators have function equivalents, but not all functions have operator equivalents; e.g., `Sqrt(a)` can only be represented by the function. Also see the Function operator for storing a function under a name.

A global variable is a name to hold values that exists for the remainder of a session. Globals can contain many types of values, including numbers, strings, lists, or references to objects. They are called globals because they can be referred to almost anywhere, not just in some specific context.

In infix operator takes one argument on each side, such as + in arithmetic, `3 + 4`, or the = in an assignment, `a=7`.

A list is a multiple-item data type entered in special brace { } notation or with the List operator. Lists enable scripts to work with many things at once, often in the place of a single thing.

A matrix is a JMP data type for a rectangular array of rows and columns of number. In JSL, matrices are entered in bracket [] notation or with the Matrix operator.

A message is a JSL statement that is directed to an object, which knows how to execute the message.

In JMP data tables, metadata are data about the data, such as the source of the data, comments about each variable, scripts for working with the data, and so on.

An event generated by pressing down the mouse button. See “Handle,” p. 302 and “MouseTrap,” p. 305.

An event generated by releasing the mouse button. See “Handle,” p. 302 and “MouseTrap,” p. 305.

A name is something to call a thing. For instance, when you assign the numeric value 3 to a global variable in the statement `a=3`, “a” is a name.

A named argument is an argument with a given context-sensitive effect chosen from a finite set of predefined choices for a particular JSL operator, function, message, etc. For example, in a Graph Box command, Frame Size is one of numerous possible named arguments that is defined to have a specific meaning that is unique to Graph Box. Frame Size happens to expect (non-named) arguments of its own which are pixel dimensions. JSL could potentially have named arguments with the same names to many different functions, and each function would define the behavior of the named argument for its own purposes.

Microsoft’s standard for Open DataBase Connectivity. JSL supports access to any ODBC-enabled data source through the Open Database command.

In syntax summaries, a placeholder for any reference to an analysis platform, e.g. Bivariate[1].
object An object is a dynamic entity in JMP, such as a data table, a data column, a platform results window, a graph, etc. Most objects can receive messages telling them to act on themselves in some way.

operator Usually operator refers to a one- or two-character symbol such as + for addition or <= for less than or equal to.

POSIX POSIX is an acronym for Portable Operating System Interface and is a registered trademark of the IEEE. POSIX pathnames enable you to use one syntax for paths for any operating system, instead of having to use a different syntax for each.

postfix operator A postfix operator takes an argument on its left side (before the operator), such as a++ for postincrement or a-- for postdecrement.

pre-evaluated statistics Statistics that are calculated once and used as constants thereafter.

prefix operator A prefix operator takes one argument on its right side (after the operator), such as !a for negation.

reference A way to address a scriptable object in order to send it messages, e.g., column(“age”) or Current Data Table() or Bivariate[1]. Typically a reference is stored in a global variable for convenience.

row state A data element type to store any combination of the following attributes for data rows: excluded, hidden, labeled, selected, color, marker, hue, shade.

scalar A simple non-matrix numeric value.

scoping operator A scoping operator forces a name to be interpreted as a particular type of data element, for example the : operator in :name forces name to be resolved as a column; the :: operator in ::name forces name to be resolved as a global variable.

toggle If you omit the argument (1 or 0) for a boolean command, the command toggles the setting—that is, it flips the setting from on to off, or from off to on. Sending such a command repeatedly flips back and forth between on and off. If you include the boolean argument, the command sets an absolute on or off state, and sending the command repeatedly has no further effect.

vector A matrix with only one column or row.
Symbols

! 473
- 444, 447
-- 446
!= 474
* 445
* = 445
+ 442
    patterns 588
++ 446
+= 443
/* */ 439
// 439, 444
//! 439
//= 444
:\ 441
:: 442
:: 475
; 440
< 471
< ... <= 472
<= 472
<= ... < 473
= 443
= = 447
=== 470
> 470
>= 471
\ 446
| 474
|| 459
||= 460

A
Abbrev Date 475
Abs 448
Add 442

Add To 443
All 497
And 469
AndMZ 469
AndV3 469
Any 498
Arc 549
Arc Cos H 452
Arc Cosine 452
ArCos 452
ArcSine 453
ArcSinH 453
ArcTan 453
ArcTangent 453
ArcTanH 453
Arg 594
Arg Expr 594
Arrhenius 449
Arrhenius Inv 449
Arrow 549
ArSin 453
As Column 441
As Date 476
As Global 442
As List 498
As Name 458
As Table 512
Assign 443
Associative Array 510
ATan 453

B
Back Color 550
Beep 489
Best Partition 572
Beta 449
Beta Binomial Distribution 567
Beta Binomial Probability 567
Beta Binomial Quantile 567
Beta Density 560
Beta Distribution 560
Beta Quantile 560
Binomial Distribution 567
Binomial Probability 568
Binomial Quantile 568
Blob MD5 467
Blob Peek 467
Blob to Char 467
Border Box 531
Break 494
Button Box 532

C
Caption 489
CDF 572
Ceiling 448
Char 458
Char to Blob 467
Char to Hex 468
Check Box 532
ChiSquare Density 561
ChiSquare Distribution 561
ChiSquare Log CDistribution 570
ChiSquare Log Density 570
ChiSquare Log Distribution 571
ChiSquare Noncentrality 570
ChiSquare Quantile 561
Chol Update 498
Cholesky 498
Choose 494
Circle 550
Clear Globals 440
Clear Log 491
Close 512
Close All 513
Col List Box 533
Col Max 576
Col Maximum 576
Col Mean 576
Col Min 576
Col Minimum 576
Col N Missing 577
Col Number 577
Col Quantile 577
Col Shuffle 455
Col Standardize 578
Col Std Dev 578
Col Sum 578
Color to HLS 487
Color to RGB 488
Column 526
Column Dialog 548
Column Name 526
Combo Box 533
comments 439
Concat 459
Concat Items 459
Concat To 460
Contains 460
Continue 494
Contour 550
Contour Function 551
Convert File Path 483
Cos 454
CosH 454
Cosine 454
Count 521
Cumulant Quadrature 499
Current Data Table 480
Current Journal 480
Current Metadata Connection 598
Current Report 480
Current SAS Connection 602
Current Window 480

D
Data Table 513
Datafeed 630
Date DMY 476
Date MDY 476
Day 477
Day of Week 477
Day of Year 477
debug run 440
debug step 440
Decode64 Double 480
Derivative 582
Design 499
Design Nom 499
Index

Design Ord 499
DesignF 499
Desirability 572
Det 500
Diag 500
Dialog 549
Dif 522
Digamma 449
Direct Product 500
Distance 500
Divide 444
Divide To 444
Drag Line 551
Drag Marker 551
Drag Polygon 552
Drag Rext 552

E

e 442
E Div 501
E Mult 501
Eigen 501
Empty 481
Encode64 Double 481
Ends With 461
Equal 470
Eval 594
Eval Expr 595
Eval Insert 595
Eval Insert Into 595
Eval List 595
Excerpt Box 533
Exit 491
Exp 450
Expr 596
Expr As Picture 596
Extract Expr 596

F

F Density 561
F Distribution 561
F Log CDistribution 571
F Log Density 571
F Log Distribution 571
F noncentrality 570
F Power 561
F Quantile 561
F Sample Size 562
Factorial 450
FFT 579
File in Directory 484
Fill Color 553
Fill Pattern 553
First 596
Floor 448
For 495
For Each Row 522
Format 477
Format Date 477
Functions 582

G

G Inverse 501
Gamma 450
Gamma Density 562
Gamma Distribution 562
Gamma Log CDistribution 571
Gamma Log Density 571
Gamma Log Distribution 571
Gamma Poisson Distribution 568
Gamma Poisson Probability 569
Gamma Poisson Quantile 569
Gamma Quantile 562
Get Addr Info 484
Get Current Directory 484
Get Default Directory 485
Get File Search Path 485
Get Name Info 485
Get Path Variable 485
Get Project 486
Get SAS Version Preference 599
Global Box 553
GLog Density 562
GLog Distribution 562
GLog Quantile 562
Glue 440
Gradient Function 553
Graph 554
Graph 3D Box 559
Graph Box 554
Greater 470
Greater or Equal 471

H
H Center Box 534
H Direct Product 501
H Line 554
H List Box 554
H Sheet Box 534
H Size 554
Handle 554
Head 596
Head Expr 596
Head Name 597
Head Name Expr 597
Heat Color 488
Hex 468
Hex to Blob 468
Hex to Char 468
Hex to Number 468
Hier Box 534
Hier Clust 579
HLS Color 488
Host Is 484
Hour 478
Hypergeometric Distribution 569
Hypergeometric Probability 569

I
Icon Box 534
Identity 502
If 495
If Box 534
IfMZ 495
IfV3 495
IGamma 450
In Days 478
In Format 478
In Hours 478
In Minutes 478
In Polygon 554
In Weeks 478
In Years 478
Include 583
Index 502
Insert 461

K
KDE 579
Index
L
Lag 522
Left 462
Length 462
LenthPSE 580–581
Less 471
Less LessEqual 472
Less or Equal 472
LessEqual Less 473
LGamma 451
Line 555
Line Style 555
Lineup Box 535
List 496
List Box 535
Ln 451
LnZ 583
Load DLL 584
Load Text File 584
Loc 506
Loc Max 506
Loc Min 506
Loc Sorted 507
Local 440
Lock Globals 441
Log 451
Log10 451
Log1P 451
Logit 451
Long Date 478
Lowercase 462
LPSolve 580
M
Mail 489
Main Menu 491
Marker 555
Marker Size 555
Match 496
MatchMZ 496
MatchV3 496
Matrix 502
Matrix Box 535
Max 573
Maximize 597
Maximum 573
MDYHMS 478
Mean 573
Meta Connect 599
Meta Create Profile 600
Meta Delete Profile 600
Meta Disconnect 601
Meta Get Repositories 601
Meta Get Servers 601
Meta Is Connected 601
Meta Set Repository 602
Meta Stored Process 601
Min 573
Minimize 597
Minimum 573
Minus 444
Minute 479
Mixture Moment 580
Mod 448
Modulo 448
Moment Quadrature 503
Month 479
Mousetrap 555
Multiply 445
Multiply To 445
Multivariate Gaussian Quadrature 581
Multivariate Normal Impute 573
Munger 462
N
N Arg 598
N Arg Expr 598
N Choose K 451
N Col 513
N Cols 513
N Items 497
N Missing 573
N Row 522
N Rows 522
N Table 514
Name Expr 597
Neg Binomial Distribution 569
Neg Binomial Probability 569
New Column 514
New Project 493
New Table 514
New Window 535
Normal Biv Distribution 565
Normal Contour 556
Normal Density 565
Normal Distribution 565
Normal Log CDistribution 571
Normal Log Density 571
Normal Log Distribution 572
Normal Quantile 566
Not 473
Not Equal 474
Num 462
Num Deriv 583
Num Deriv2 583
Number 573
Number Col Box 535

O
Open 514
Open Database 635
Open Datafeed 630
Open Project 494
Or 474
OrMZ 475
Ortho 508
Ortho Poly 509
OrV3 475
Outline Box 535
Oval 556

P
Page Break Box 535
Panel Box 535
Parameter 583
Parse 462
Parse Date 478
Parse XML 584
Pat Abort 585
Pat Altern 585
Pat Any 585
Pat Arb 586
Pat Arb No 586
Pat At 587
Pat Break 587
Pat Concat 588
Pat Conditional 588
Pat Fail 588
Pat Fence 589
Pat Immediate 589
Pat Len 589
Pat Match 590
Pat Not Any 590
Pat Pos 590
Pat R Pos 591
Pat R Tab 591
Pat Regex 591
Pat Rem 591
Pat Repeat 592
Pat Span 592
Pat String 592
Pat Succeed 593
Pat Tab 593
Pat Test 593
Pat Test 593
Pen Color 556
Pen Size 556
Pi 442
Pick Directory 486
Pick File 486
Pie 556
Pixel Line 556
Pixel Move 557
PixelOrigin 557
Platform Preferences 493
Plot Col Box 536
Poisson Distribution 570
Poisson Probability 570
Poisson Quantile 570
Polygon 557
Polytope Uniform Random 581
Popup Box 536
Post Decrement 446
Post Increment 446
Power 446
Pref 492
Preference 492
Preferences 492
Print 489
Print Matrix 503
Probit 566
Product 574
Q
QR 509
Quantile 574
Quit 491

R
Radio Box 536
Random Beta 455
Random Beta Binomial 455
Random Binomial 455
Random Cauchy 455
Random Exp 455
Random Gamma 456
Random Gamma Poisson 456
Random Geometric 456
Random GLog 456
Random Index 503
Random Integer 456
Random Johnson Sb 456
Random Johnson Sl 456
Random Johnson Su 456
Random Lognormal 456
Random Negative Binomial 457
Random Normal 457
Random Poisson 457
Random Reset 457
Random Seed 457
Random Seeded Normal 457
Random Seeded Uniform 457
Random Shuffle 503
Random Triangular 457
Random Uniform 458
Random Weibull 458
Rank 508
Rank Index 508
Ranking 508
Ranking Tie 508
Rect 557
Recurse 583
Regex 462
Regex Match 594
Remove 463
  associative arrays 511
Remove From 463
  associative arrays 511
Repeat 463
Report 536
Resample Freq 458
Reverse 464
Reverse Into 464
Revert Menu 492
RGB Color 489
Right 464
Root 451
Round 448
Row 523
run script without opening 439

S
SAS Assign Lib Refs 606
SAS Connect 602
SAS Desaaign Lib Refs 606
SAS Disconnect 603
SAS Export Data 612
SAS Get Data Sets 606
SAS Get File 610
SAS Get File Names 611
SAS Get File Names In Path 611
SAS Get File Refs 611
SAS Get Log 605
SAS Get Output 605
SAS Get Results 607
SAS Get Var Names 607
SAS Import Data 607
SAS Import Query 610
SAS Is Connected 603
SAS Load Text File 612
SAS Name 613
SAS Open For Var Names 613
SAS Send File 611
SAS Submit 603
SAS Submit File 605
Save Log 490
Save Text File 584
SBIinv 574
SbTrans 574
Scene Box 536
Schedule 635
Scheffe Cubic 451
Script Box 536
Second 479
Send 536
Sequence 523
Set Current Directory 487
Set Default Directory 487
Set File Search Path 487
Set Menu 492
Set Path Variable 487
Shape 504
Sheet Part 537
Shift 464
Shift Into 464
Short Date 479
Show 490
Show Commands 490
Show Globals 441
Show Preferences 493
Show Properties 490
Simplify Expr 598
Sin 454
Sine 454
Sinh 454
Site ID 490
Slider Box 537
SIInv 574
SITrans 574
Socket 631
Solve 509
Sort Ascending 497
Sort Descending 497
Sort List 497
Sort List Into 497
Speak 490
Spline Coef 574
Spline Eval 574
Spline Smooth 574
Sqrt 452
Squash 452
Squish 452
Starts With 465
Status Msg 490
Std Dev 575
Step 496
String Col Box 537
Students t Density 566
Students t Distribution 566
Students t Quantile 566
Subscript 497
Substitute 465
Substitute Into 466
Substr 466
Subtract 447
Subtract To 447
SuInv 575
Sum 575
Summarize 575
Summation 575
Suppress Formula Eval 514
SuTrans 576
SVD 509
Sweep 509

T
t Density 566
t Distribution 566
t Log CDistribution 572
t Log Density 572
t Log Distribution 572
T Noncentrality 570
t Quantile 566
Tab Box 537
Table Box 537
Tan 454
Tangent 454
TanH 454
Text 557
Text Box 538
Text Color 557
Text Edit Box 538
Text Size 557
Throw 583
Tick Seconds 491
Time of Day 479
Today 479
Tolerance Limit 582
Trace 508
Transparency 558
Transpose 505
Trigamma 452
Trim 466
Try 583
Tukey HSD P Value 566
Tukey HSD Quantile 566
Type 483
U
Unlock Globals 441
Uppercase 466

V
V Center Box 538
V Concat 505
V Line 558
V List Box 538
V Max 509
V Mean 509
V Min 509
V Sheet Box 538
V Size 558
V Std 509
V Sum 509
Vec Diag 505
Vec Quadratic 506

W
Wait 491
Watch 441
Web 490
Week of Year 479
Weibull Density 567
Weibull Distribution 567
Weibull Quantile 567
While 496
Wild 598
Wild List 598
Window 491
Word 466
Words 466
Write 491

X
X Function 558
X Origin 558
X Range 559
X Scale 558
XML Attr 584
XML Decode 584
XML Encode 585
XML Text 585

Y
Y Function 558
Y Origin 559
Y Range 559
Y Scale 558
Year 479

Z
Zero or Missing 475
Symbols

! 76
definition 76
" 6, 116
& 76
definition 76
( ) 6, 32
* 353
+ 412
, 6
..\ 115
/ 353
/* */ 37
// 37
//! 6, 37
: 42-45, 126, 306
:* 354
:/ 354
:: 42-43, 45-46, 306, 352
; 38, 93
< <= 76
<< 119, 182, 184, 188, 194, 232-233
<= 38, 61
== 170
and Is Missing 78
>> 412
>? 412
[... ] 34
[Action Choice] 185
[Action] 121, 185
[Boolean] 185
[Column Names] 185
[Enum] 185
[New Entity] 185
[Scripting Only] 121
[Subtable] 121, 185
\! " 34
\!\ 34
\!0 34
\!b 34
\!f 34
\!N 34, 87
\!n 34
\!r 34
\!t 34
__ 408
: 356
{ } 35
| 76
definition 76
glossary 637
|/ 355, 372
|/= 355
|| 355, 372, 412
||= 355

A

Action 186
Add From Row States 162
Add Multiple Columns 124
Add Rows 136
Add To Row States 162
Add Watch (debugger) 53
Alarm Script 218
alarm scripts 218
ANALYSIS PLATFORM 77, 359
analysis platform
by group 180
scripting 190
And 76-77
missing values 77
AndMZ 77, 98
ANOVA 371-373
Any 77, 359
Append 16, 242
Arc 291
ArcBall
definition 319
ArcBall 316–317, 319
Arg 418
argument 190
  glossary 637
ARIMA 206
  seasonal 206
arithmetic
  matrix 353
arithmetic operators 61
Arrow 189
As Column 44–45
As Global 45
As Row State 162, 169, 523
As Table 135, 370
Assign 38, 61–62
assignment operators 61
associative array 35
  adding keys and values 108
  assign values 107
  constructors 107
  create 107
  data type 107
  default value 107
  deleting keys and values 108
  graph theory 110
N Items 108
  traversing 110
associative arrays
  messages 511
  sets 110
attenuating light source 337
audit trail 7
Auto-complete parentheses and braces, in script editor 50
autoexec script 133
automatic recalc 188
auto-submit 6, 37
axis
  property parameters 153
Axis Box 239
  axis boxes
    messages 545
Begin Data Update 135
Bézier curves 342
Big Class.jmp 67, 89, 187
Bivariate 13, 185, 194, 228, 231, 235, 285
blank 34
BlendFunc 340
  bool
  glossary 637
boolean 179
  glossary 637
Border Box 251
  braces
    script editor, auto-complete 50
  bracket see [ ], matrix, Subscript
  break point 33
Bring Window to Front 189
Bubble Plot 215
Bullet Point 244
Button Box 228, 239, 244–245, 248, 302, 307
By 180–181, 190, 194
C
Call List 317, 331
CallDLL 406
CallList 319
Capability 220
capability analysis 259
Caption 86, 88
carriage return 34, 117
Categorical 207
Cell Plot 216
Char 13, 71, 145, 420–421
  character functions 67
  character values
    missing 78
Char 213
Check Box 248, 269
child object 184, 187–188, 192, 194, 196
Choice 207
Cholesky 351, 366
Choose 98
Circle 293
clear 318
Clear Column Selection 148
Clear Globals 102
Clear Select 139
Close 120–121, 232
Close Window 189
classifier
  Hierarchical 209
K-Means 209
coding (DOE) 147, 186
col
property parameters 153
col

glossary 637
Col List Box 251
Get Items 251
Col Maximum 170
Col Mean 170–171
Col Minimum 170
Col N Missing 170
Col Number 170
Col Quantile 170
Col Standardize 170
Col Std Dev 170–171
Col Sum 170

colons 42–46, 126, 306, 352
Color 312–313, 331
Color By Column 139
Color Of 159, 161–162, 165, 168, 523
Color State 161–162, 165, 168, 524
Colors 298
Colors 138
colors 298–299
Column 37, 44, 147
column 186
Column Dialog 266, 271–272
Column Name 145
column names, special characters 147
column properties
delete (JSL) 152
get (JSL) 152
set (JSL) 152
col mntry 152
column references 126
column vector 351
Columns 273
columns
messages 527
columnwise functions 171
Combine States 160–162, 165–166, 291, 324
Combo Box 248, 269
comma 6, 32, 95
command 190
glossary 637
vs. message 121
comments 37
comparison operator
matrix 359
comparison operators 74
examples 76
missing values 77
Concat 71, 355, 372
Concat Items 71
Concatenate 143
conditional function 97
confusion alert
| (pipe symbol) 29
Col Mean vs. Mean, et al. 170
column names 147
current row in data table 160
If vs. IfMZ, et al. 77
introduction 1
missing values for logical operators 77, 98
path variables 36
platforms vs. reports 185, 231
pre-evaluated statistics vs. Summarize
arguments 170
row state operators, which are which? 162
same word, different contexts 150
semicolons and commas in JSL vs. C 95
Set and Get messages 148
subscripting column references 126
Week of Year 80
connections
SAS metadata server objects 613
construct display tree 240
Contains 71
current context 40, 150
Contingency 196
Continuous Distribution 193
Contour 297
Contour Function 283
Contour Plot 215
Contour Profiler 222
Contour Value 221
Control Chart 18, 217
control limits
property parameters 153
Convert File Path 36
Copy From Row States 162
Copy Picture 539
Copy To Row States 162
creating JMP windows 225
currency 85
current context data table 44
Current Data Table 13, 37, 44, 119, 125, 133

glossary 637
current row number 41, 46, 125, 127
glossary 637
custom display 23
display boxes messages 539
display scripting 225
display trees 225
  navigating 227
Distribution 18, 192
Divide 353
DLL 406
DOCUMENTS 36
DOE 190
DOE Bayes Diagonal 192
DOE K Exchange Value 191
DOE Mixture Sum 191
DOE Starts 158
DOE Starts 191
double quote 34
double underscore and variable names 408
Drag Line 306
Drag Marker 306
Drag Polygon 306
Drag Rect 306
Drag Text 306
dt data table reference 230
glossary 637
  referencing a data table 119

E
e 32
eager operator 415
Edge Flag 331
Edit Text 269
Editable 280
Ediv 354
Eigen 351, 365
eigenvalue decomposition 365
elementwise matrix operator 354
e17iding operator 75, 184
glossary 637
EMult 354
Enable 326, 330, 337, 339, 347
cryption
  encrypting a script 407
  hiding variables 408
End 321
End Data Update 335
equality operators 74
escape sequences 34, 87
Eval 186, 415–416, 420–421
eval and Formulas 416
Eval Coord 331
Eval Expr 420–421
Eval Formula 150
Eval Insert 419
Eval List 106, 275, 420–421
Eval Point 331
EvalCoord1 343
EvalCoord2 345
EvalFormula 134
EvalMesh1 343
EvalMesh2 345
EvalPoint2 345
Evaluators
  one-dimensional 342
  two-dimensional 344
exceptions 403–404
Exclude 148, 155
Excluded 161–162, 164, 524
Excluded State 161–162, 165, 524
Exp 32
Expr 274, 415, 417, 420–421, 424, 428
Expr As Picture 265
expression 423, 428
Extra space at bottom of document, in script editor 50

F
Factorial 405
fence matching 32
file path format 36
Files In Directory 268
Fill Color 298
Fill Pattern 301
find a window 232
First 94
Fishbone Diagram 216
Fit Life by X 211
Fit Model 41, 197, 373
Fit Separately 199
Fit Y by X 194
Fitness.jmp 286
Fitted distributions 193
fitting personality 198
flow control 91
Fog 341
Font Color 298
fonts, setting in script editor 50
For 5, 12, 15, 94, 237, 258, 282, 369
For Each Row 16, 46, 125–126, 128, 159–160
Format 81, 84, 149–150
   currency 85
formatting scripts 49
formfeed 34
Formula 124, 172, 416
   formula
      and eval 416
         evaluation 134
        picture 265
Frame Box 238
   frame boxes
      messages 543
freeze all 123
freeze frames 123
freeze frames with scripts 123
freeze pictures 123
Frustum 315
Function 103, 404
   function
      glossary 638
   functions
      local variables 404

G
Gauge Chart 219
Gaussian Process 205
Generalized Linear Model 203
Get 237
Get All Columns As Matrix 135
Get As Matrix 135, 148, 369–370
Get Column Names 145
Get Desirability 221
Get Effect Names 198
Get Effect PValues 198
Get Estimates 198
Get Format 150
Get Formula 148, 150
Get Items
   and Col List Box 251
Get List Check 151
Get Lock 154
Get Name 148
Get Parameter Names 198
Get Picture 265
Get Property 132, 152
Get Random Effect Names 198
Get Range Check 151
Get Rows Where 138, 369
Get Script 133, 154, 188
Get Selected Columns 125, 146
Get Selected Rows 138, 369
Get Std Errors 198
Get Table Variable 132
Get Text 251
Get Values 148
Get Variance Components 198
Get Window Position 189
Get Window Size 189
Get X Matrix 198
Get XPX Inverse 198
Get Y Matrix 198
GIInverse 361
GLM 203
Global Box 228, 239, 244–246, 302
global variable 41–43, 45
   and expr 416
   and functions 405
global variables 4, 31, 33
   and expr 425
   and Global Box 239
   and in-place operators 424
   and matrices 365
   and Slider Box 239
   and Summarize 129
   and Summary 141
   and watch 101
   assignment 61
column names 126
column reference 41
glossary 638
hiding 408
interactive display elements 245
locking and unlocking 102
prefix operator 126
protecting 102
referencing a column 147
referencing a data table 119
referencing an object 182
scoping 45, 352
setting values interactively 14
showing and clearing 102
testing for empty value 85
Glue 38, 93
glue 93
Go To Line 4
Go To Row 137
Gradient Function 285
graph 23
Graph Box 42, 280, 282
Graph Builder 212
Index

graph theory and associative arrays 110
graphic of formula 265
graphics primitives 321

H
Handle 23–24, 26, 244, 302–307
HDirect Product 364
HeadName 419
Hex to Char 429
Hidden 161–162, 164, 524
Hidden State 161–162, 165, 524
Hide 148, 155
hiding global variables 408
Hierarchical Cluster 209
HierBox 228, 239
HLine 288
HList 273
HList Box 18, 227, 240
HOME 36
Host Is 86
Hue State 161–162, 168, 524

I
Icon Box 239
Identity 352
If 97
If vs. IfMZ, et al. 98
IfMZ 77, 98
import 115, 117
In Days 80
In Format 84
In Hours 80
In Minutes 80
In Polygon 296
In Weeks 80
In Years 80
Include 405
   Parse Only 405
indent guides, showing in script editor 50
Index 352, 358
infix operator 35, 44, 126, 352
glossary 638
in-place operators 423, 425
Insert 423, 427
Insert Into 423, 425, 427
InsertInto 423
instructional simulation 25
interactive graph 23, 244, 302–307
Interpolate 99
Invalid Row Number error 126
Inverse 360–361
Inverse Matrices, updating 368
Invert Expr 433
invert matrix 360–365
Invert Row Selection 137
invisible 118, 187–188
invisible data tables 118
Invisible Reports 187
Is Empty 85
Is List 106
Is Matrix 353
Is Missing 78
   and == 78
Ishikawa Diagram 216
Item Analysis 211
iterate 94, 127

J
J 352, 372
JMP Version() 86
JMP Window 431
jmpStart.jsl 133
Join 143
Journal Box 253
Journal Window 189, 265
Journals 265
JSL 29, 54, 115
JSL browser 54
JSL Debugger 32

K
kernel density estimate 23
K-Means Cluster 209

L
Label 148
Labeled 161–162, 164, 525
Labeled State 161–162, 165, 525
Lag 128
leaf 229
legend, adding 286
LELE 151
LELT 151
lexical rules 32
Light 337
light
   attenuating 337
Light Model 338
Line 287, 290
line numbers, in script editor 50
Line Stipple 326
Line Style 301
Line Up Box 251
Line Width 326
LINE_LOOP 323
LINE_STRIP 322
linefeed 34, 117
LIST 322
List 35, 103, 106
list 35, 428
glossary 638
List Box 238, 248, 269
List Check 151
list check
property parameters 152
list checking 151
Load Matrix 335
Load Text File 405
LoadDLL 406
Loc 105, 358
Loc Max 358
Loc Min 358
Loc Sorted 358
Local 42, 102
local
parameters 404
variables 102
local variables
in functions 404
Lock 154
Lock Globals 102
Log 41
logical and, definition 76
logical not, definition 76
logical operator
matrix 359
logical operators 76
logical or, definition 76
Logistic 197
Loglinear Variance 202
Look At 318
LTLE 151
LTLT 151
glossary 638

M
Macintosh 86
macro 369, 404, 416, 423
Mail 86, 89
Make SAS Data Step 387
Make SAS Data Step Window 387
manipulate expressions 423–428
manipulate lists 423–425
MANOVA 201
Map 1 343
Map 2 345
Mapgrid1 343
Mapgrid2 345
Marker 169, 290–291
Marker by Column 139
Marker Of 161–162, 165, 525
Marker Size 290
Marker State 161–162, 165, 525
marker, custom 301
Markers 138
Match 98
Matched Pairs 197
matching parentheses 32
MatchMZ 77, 98
Material 331
Matrix 288, 351–353
matrix
arithmetic 353
arithmetic operators 431
comparisons 359
concatenation 355
constructing 351
constructing with expressions 352
data tables 369–370
decomposition 365
definition 351
deleting rows and columns 353
diagonal 355
glossary 638
introduction 35
logical operators 359
normalization 365
numeric functions 354
range checks 359
ranges of rows or columns 358
ranking 359
solving linear systems 360
sorting 359
special constructors 352
subscripting 356
summarizing columns 360
transposing 356
Matrix Box 239–241
matrix boxes
   messages 548
   Matrix Mult 353
   Max 359
   MaxCol 272
Maximize 434
Maximize Window 189
Maximum 170
Mean 170
message 119
   child object 194
   column 147
   data table 115
   display 232, 257
   glossary 638
   live platform 182
   objects 118
   platform 190
   Report 230
   show properties 121
messages
   associative arrays 511
   axis boxes 545
   columns 527
   data tables 514
   databases 635
   datafeed 630
   display boxes 539
   frame boxes 543
   matrix boxes 548
   nom axis boxes 548
   number col boxes 547
   outline boxes 542
   platforms 529
   SAS results 627
   SAS server objects 615
   scheduling 635
   sockets 632
   stored processes (SAS) 622
   string col boxes 547
   table boxes 544
   text boxes 544
metadata 131
   glossary 638
   table and column properties 13
   metadata server objects (SAS)
      connections 613
Min 359
Minimize 434
Minimize Window 189
Minimum 170
missing values
   and comparison operators 77
   character 78
   matrix 359
mixture (DOE)
   property parameters 153
Mixture Profiler 223
modal dialog boxes 266
   constructing 266, 268
   data column 271
   pick dialogs 270
   using JMP built in dialogs 267
model dialog 199
model dialogs
   closing 199
Month 79
mousedown
   glossary 638
MouseTrap 244, 302, 305–307
mouseup
   glossary 638
Move Rows 138
Move Selected Columns 146
Move Window 189
Mult Matrix 335
Multiply 353
Multivariate 208
Multivariate Control Chart 218
Munger 72
N
N Items 106
name 33
   glossary 638
Name Expr 71, 428
name-binding rules 43
named argument
   glossary 638
NameExpr 416–417, 421, 426
NArg 417
navigate display tree 227
NCol 127, 353
Neural Net 205
New Column 124, 172
New Script 31
New Table 118–119, 121, 125
New Table Property 132–133
New Table Variable 132
New Window 14, 18–19, 42, 177, 240, 280
Next Selected 139
NItems 105
NMissing 170
nom axis boxes messages 548
Nominal Distribution 193
Nominal Logistic 202
Nonlinear Fit 203
Nonpar Density 195
Normal 331
Normal Contour 284–285
Normal Density 26
Normal vectors 339
normalization 365
Not 76
notes (column) property parameters 152
not-in-place operators 424–425
NRow 93, 127, 137, 353
NTable 120
null 34
Num 421
Number 170
Number Col Box 238
number col boxes messages 547
numbers 33
NumDeriv 432–433
NumDeriv2 432–433

O

obj glossary 638
object reference 230
object 54–56, 115, 118, 121, 147, 182, 194, 230, 232, 257
child 192
child object 184, 187–188, 194, 196
glossary 639
ODBC 384
ODBC database glossary 638
On Open 133
OneWay 175, 179, 182, 196
Open 37, 115–117, 119, 121, 125
Open Database 118, 384
open database 384
OpenGL 311
operating system 86
operator 35, 54–433
glossary 639
operator functions 59
operator names
script editor, adding spaces between words 51
operator tips, showing in script editor 50
operators 38
arithmetic 61
assignment 61
order of precedence 35
option 190
Or 76–77
missing values 77
order of precedence 35
Ordinal Logistic 202
OrMZ 77, 98
Ortho 316, 367
Ortho2D 316
orthogonal polynomial 368
Orthographic projections 314
orthonormalize 367
OrthoPoly 368
Outline Box 14, 18, 228, 231, 238, 240–241
outline boxes messages 542
output suppressing 188
Oval 294
Overlay Plot 213

P

Panel Box 252
Parallel Plot 216
Parameters (Enable command) 347
Parametric Survival 203
parentheses 6, 32
script editor, adding spaces inside 51
script editor, auto-complete 50
parenthesis matching 32
Pareto Plot 219
Parse 420–421
Parse XML 409
Partial Disk 331
Partial Least Squares (PLS) 210
Partition 205
Patch Editor.jsl 345
path format 36
path variables 36
patterns
case insensitive 414
pausing script execution 101
Pen Color 298
Pen Size 301
Perspective 312, 315
Perspective projections 314
pick 346
pick dialogs 267
Pick Directory 267
Pick File 267
Picture Box 228, 239
Picture data type 265
picture of formula 265
Pie 291
Pixel Line To 301
Pixel Move To 301
Pixel Origin 301
platform
by group 180
scripting 190
platform scripting
Scatterplot Matrix 216
platform scripting 173
Bivariate 194
Bubble Plot 215
Capability 220
Categorical 207
Cell Plot 216
Chart 213
Choice 207
Contingency 196
Contour Plot 215
Contour Profiler 222
Control Chart 217
Custom Profiler 223
Design of Experiments (DOE) 190
Diagram 216
Discriminant 210
Distribution 192
Fishbone Diagram 216
Fit Model 197
Fit Y by X 194
Gauge Chart 219
Gaussian Process 205
Hierarchical Cluster 209
Ishikawa Diagram 216
Item Analysis 211
K-Means Cluster 209
Life Distribution 211
Logistic 197
Matched Pairs 197
Multivariate 208
Multivariate Control Chart 218
Neural Net 205
Nonlinear Fit 203
Oneway 196
Overlay Plot 213
Parallel Plot 216
Pareto Plot 219
Partial Least Square (PLS) 210
Partition 205
PLS 210
Principal Components 210
Profiler 220
Scatterplot 3D 214
Screening 203
Spinning Plot 215
Surface Plot 223
Survival 212
syntax 179
Tabulate 192
ternary plots 220
Ternary Plot 216
time series 206
tree map 216
Variability Chart 219
writing interactively 175
platforms
messages 529
Plot Col Box 238
PLS 210
Point Size 326
POINTS 322
POLYGON 322
Polygon 295
Polygon Mode 328
Polygon Offset 330
Pop Matrix 319
Popup Box 248
elements 249
Positive Level 197
POSIX 36
glossary 639
postfix operator 35
glossary 639
precedence 35, 94
pre-evaluated statistics 42, 170
glossary 639
preferences 133
preferences for the script editor 50
pre-fill columns 273
prefix colon 44
prefix operator 35, 126, 352
glossary 639
Prepend 243
Preselect Role 150
Previous Selected 139
Principal Components 210
Print 5, 86–87, 415
Print Window 122, 189
procrastination operator 415
Product 96
Profiler 220
programming function 91
projections
  orthographic 314
  perspective 314
property 132, 152
Proportional Hazard 202
protecting global variables 102
punctuation 32, 95
Push Matrix 319, 334
QC Alarm Script 132, 218
QC alarm scripts
  property parameters 153
QC Test Alert 132, 218
QR 368
QUAD_STRIP 323
Quadric Draw Style 332
Quadric Normals 332
Quadric Orientation 332
QUADS 323
Quantile Plot 193
quotation marks 6, 116
quoting operator 415

R
Radio Box 248
Radio Button 269
random functions 66
Random Reset 135, 151
Range Check 151
range check
  no warning 151
  property parameters 152
  syntax 151
range checking 151
Rank 359
rearrange columns 125
Rect 293, 297
Recurrence 212
Recurse 405
Redo Analysis 176, 188
reference 119, 147, 230, 232, 257
  glossary 639
Reformat Script, in script editor 49
reformatting scripts 49
Regression 370
regression 20, 370–371
regular expressions 414
Remove 423, 427
Remove From 423, 425, 427
Remove Watch (debugger) 53
Repeat 72
Report 187, 189, 230, 259
report 178
Resample Freq 67
Reshow 232
response limits (DOE, desirability profiling)
  property parameters 154
Reverse 423, 427
Reverse Into 423, 427
reverse rotation 365
Revert 120
Rotate 313, 316, 328
rotation 365
Row 46, 125, 127, 160
Row Legend 280, 286
Row State 159, 162, 165, 525
row state 154
  glossary 639
row state combination 161
row state scripting
    tutorial 170
row vector 351
Rows menu scripting 136
rowwise functions 171
Run (debugger) 53
Run Formulas 134, 150
Run Script 120
Run Script 31, 100
Run Without Debugger 53
running a script at startup 133

S
SAMPLE_DATA 36
SAMPLE_IMPORT_DATA 36
SAMPLE_SCRIPTS 36
SAS file type 122
SAS metadata server 387
SAS metadata server objects
<table>
<thead>
<tr>
<th>Index</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>connections</td>
<td>613</td>
</tr>
<tr>
<td>SAS Name</td>
<td>387</td>
</tr>
<tr>
<td>SAS Open For Var Names</td>
<td>387</td>
</tr>
<tr>
<td>SAS results</td>
<td></td>
</tr>
<tr>
<td>messages</td>
<td>627</td>
</tr>
<tr>
<td>SAS server objects</td>
<td>615</td>
</tr>
<tr>
<td>messages</td>
<td>613</td>
</tr>
<tr>
<td>SAS stored processes</td>
<td></td>
</tr>
<tr>
<td>messages</td>
<td>622</td>
</tr>
<tr>
<td>Save</td>
<td>120</td>
</tr>
<tr>
<td>Save Database</td>
<td>385</td>
</tr>
<tr>
<td>save formula as picture</td>
<td>265</td>
</tr>
<tr>
<td>Save Picture</td>
<td>266</td>
</tr>
<tr>
<td>Save Script</td>
<td>187</td>
</tr>
<tr>
<td>Save Script for All Objects</td>
<td>177</td>
</tr>
<tr>
<td>Save Script to Datatable</td>
<td>133</td>
</tr>
<tr>
<td>Save Script to Report</td>
<td>177</td>
</tr>
<tr>
<td>Save Script to Script Window</td>
<td>177</td>
</tr>
<tr>
<td>Save Script to Window</td>
<td>177</td>
</tr>
<tr>
<td>Save Text File</td>
<td>405</td>
</tr>
<tr>
<td>scalar</td>
<td>351</td>
</tr>
<tr>
<td>glossary</td>
<td>639</td>
</tr>
<tr>
<td>scaling</td>
<td>365</td>
</tr>
<tr>
<td>Scatterplot 3D</td>
<td>214</td>
</tr>
<tr>
<td>Scatterplot Matrix</td>
<td>216</td>
</tr>
<tr>
<td>Scene Box</td>
<td>311</td>
</tr>
<tr>
<td>Schedule</td>
<td>101</td>
</tr>
<tr>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td>messages</td>
<td>635</td>
</tr>
<tr>
<td>scoping operator</td>
<td>42,</td>
</tr>
<tr>
<td>glossary</td>
<td>639</td>
</tr>
<tr>
<td>Screen</td>
<td>203</td>
</tr>
<tr>
<td>Script Box</td>
<td>239</td>
</tr>
<tr>
<td>Script Editor</td>
<td>47</td>
</tr>
<tr>
<td>script editor</td>
<td></td>
</tr>
<tr>
<td>auto-complete parentheses and braces</td>
<td>50</td>
</tr>
<tr>
<td>extra space at bottom of document</td>
<td>50</td>
</tr>
<tr>
<td>indent guides, showing</td>
<td>50</td>
</tr>
<tr>
<td>line numbers, showing</td>
<td>50</td>
</tr>
<tr>
<td>operator tips, showing</td>
<td>50</td>
</tr>
<tr>
<td>preferences</td>
<td>50</td>
</tr>
<tr>
<td>set fonts</td>
<td>50</td>
</tr>
<tr>
<td>spaces in operator names</td>
<td>51</td>
</tr>
<tr>
<td>spaces inside parentheses</td>
<td>51</td>
</tr>
<tr>
<td>tab width</td>
<td>50</td>
</tr>
<tr>
<td>use tabs</td>
<td>50</td>
</tr>
<tr>
<td>variable value tips, showing</td>
<td>51</td>
</tr>
<tr>
<td>script style</td>
<td>49</td>
</tr>
<tr>
<td>scriptable objects</td>
<td>118</td>
</tr>
<tr>
<td>scripting platforms</td>
<td></td>
</tr>
<tr>
<td>Fit Life by X</td>
<td>211</td>
</tr>
<tr>
<td>Graph Builder</td>
<td>212</td>
</tr>
<tr>
<td>Mixture Profiler</td>
<td>223</td>
</tr>
<tr>
<td>scripting vs. programming</td>
<td>40</td>
</tr>
<tr>
<td>Scroll Lock</td>
<td>152</td>
</tr>
<tr>
<td>scroll lock</td>
<td>148</td>
</tr>
<tr>
<td>Scroll Window</td>
<td>189</td>
</tr>
<tr>
<td>seasonal ARIMA</td>
<td>206</td>
</tr>
<tr>
<td>Select</td>
<td>232</td>
</tr>
<tr>
<td>Select All Matching Cells</td>
<td>138</td>
</tr>
<tr>
<td>Select All Rows</td>
<td>137</td>
</tr>
<tr>
<td>Select Matching Cells</td>
<td>138</td>
</tr>
<tr>
<td>Select Rows</td>
<td>125</td>
</tr>
<tr>
<td>Where</td>
<td>138</td>
</tr>
<tr>
<td>Selected</td>
<td>161–</td>
</tr>
<tr>
<td>Selected State</td>
<td>161–</td>
</tr>
<tr>
<td>seasonal ARIMA</td>
<td>206</td>
</tr>
<tr>
<td>selected</td>
<td>161–</td>
</tr>
<tr>
<td>send</td>
<td>194</td>
</tr>
<tr>
<td>Send To Report</td>
<td>232</td>
</tr>
<tr>
<td>Sequence</td>
<td>73</td>
</tr>
<tr>
<td>set and associative arrays</td>
<td>110</td>
</tr>
<tr>
<td>Set Each Value</td>
<td>135</td>
</tr>
<tr>
<td>Set Formula</td>
<td>148</td>
</tr>
<tr>
<td>Set Label Columns</td>
<td>152</td>
</tr>
<tr>
<td>Set Lock</td>
<td>154</td>
</tr>
<tr>
<td>Set Matrix</td>
<td>135</td>
</tr>
<tr>
<td>Set Name</td>
<td>122,</td>
</tr>
<tr>
<td>Set Property</td>
<td>132,</td>
</tr>
<tr>
<td>Set Row States</td>
<td>169</td>
</tr>
<tr>
<td>Set Script</td>
<td>538</td>
</tr>
<tr>
<td>Set Scroll Lock Columns</td>
<td>152</td>
</tr>
<tr>
<td>Set Selected</td>
<td>146</td>
</tr>
<tr>
<td>Set Table Variable</td>
<td>132</td>
</tr>
<tr>
<td>Set Values</td>
<td>148,</td>
</tr>
<tr>
<td>Set Wrap</td>
<td>244</td>
</tr>
<tr>
<td>Shade Model</td>
<td>339</td>
</tr>
<tr>
<td>Shade State</td>
<td>161–</td>
</tr>
<tr>
<td>Shape</td>
<td>353</td>
</tr>
<tr>
<td>Shift</td>
<td>423,</td>
</tr>
<tr>
<td>Shift Into</td>
<td>423,</td>
</tr>
<tr>
<td>short-circuiting operators</td>
<td>78</td>
</tr>
<tr>
<td>Show</td>
<td>229</td>
</tr>
<tr>
<td>Show 4, 21, 86–87, 415</td>
<td>229</td>
</tr>
<tr>
<td>Show Arcball</td>
<td>320</td>
</tr>
<tr>
<td>Show Commands</td>
<td>55</td>
</tr>
<tr>
<td>Show Globals</td>
<td>102</td>
</tr>
<tr>
<td>Show indent guides, in script editor</td>
<td>50</td>
</tr>
<tr>
<td>Show line numbers, in script editor</td>
<td>50</td>
</tr>
<tr>
<td>Show operator tips, in script editor</td>
<td>50</td>
</tr>
<tr>
<td>Show Properties</td>
<td>56,</td>
</tr>
<tr>
<td>Show Properties 203–205, 222, 147, 184–185, 188, 232, 235</td>
<td>229</td>
</tr>
<tr>
<td>Show Tree Structure</td>
<td>229</td>
</tr>
</tbody>
</table>
Show Tree Structure 229
Show variable value tips, in script editor 51
Show Window 189
Sib Append 243
sigma
property parameters 153
Simplify Expr 433
singular value decomposition 367
Size Window 189
slice matrix 357
Slider Box 14, 228, 239, 244–246, 302
socket
commands 382
sockets
datagram 382
example 382
messages 383, 632
stream 382
using 381
Solve 361
solve linear system 360–365
Sort 133, 142
Sort List 423, 427
Sort List Into 423, 427
sort matrix 359
spaces 33
Spaces in operator names, in script editor 51
Spaces inside parentheses, in script editor 51
Speak 86, 88
spec limits
property parameters 153
Sphere 332
Spinning Plot 215
Spline Coef, example 195
Spline Eval, example 195
Spline Smooth, example 195
Split 142
Stack 142
Standard Least Squares 199
startup script 133
StatusMsg 86, 89
Std Dev 170
Step 100
Step (debugger) 53
Stepwise 201
Stop Script 100
stored expression 428
stored processes (SAS)
messages 622
String Col Box 238
string col boxes
messages 547
strings 34
Subscript 231, 257, 356
subscripting column names 126
subscripts 105
with column references 126
subscripts with columns 147
Subset 140–141
Substitute 16, 424, 426, 428
Substitute Into 260, 275, 424, 426, 428
Sum 170
Summarize 93, 127, 129
Summary 129, 140–141
Summation 93, 96
Suppress Formula Eval 134
Suppressing output 188
Surface Plot 223
Survival 212
SVD 351, 367
Sweep 361–362, 372
T
tab 34
Tab Box 239, 249
Example 250
Tab width, in script editor 50
Table Box 228, 238
table variable 132
tables boxes
messages 544
Tables menu scripting 140
tabs
script editor 50
Tabulate 192
TEMP 36
Term Value 221
Ternary Plot 216
Text 297, 313, 332
Text Box 239
Text Box 250
text boxes
messages 544
text command 287
Text Edit Box 239, 250
Text file type 122
Text Wrap, controlling 244
TextEditBox 228
Throw 403
Time Of Day 79
Time Series 206
Transfer Functions 207
Title 190, 271
title 187–188
toggle 139, 183
glossary 639
tokens 32
Trace 356, 366
transfer functions (time series) 207
Translate 313, 316, 328
Transpose 143, 356
transpose 365
Tree Map 216
TRIANGLE_FAN 324
TRIANGLE_STRIP 323
TRIANGLES 322
trigonometric functions 65
troubleshoot 101, 307
    infinite loop 137
truth tables 78
Try 403
t-statistics 25
tutorial ??–373
    custom report 238
    customize analysis 13
data table 11, 126, 128
display box 275
display 18, 23, 245
display tree 241
    instructional simulation 25
    matrix 370–371
platform 13, 234
programming 93
    QC chart 169
    row state 170
Type 85

U
underscores and variables names 408
uninitialized variable 85
units of measure
    property parameters 154
unload list 275
UnloadDLL 406
Unlock Globals 102
unquoting operator 415
Update 144, 312
Use tabs, in script editor 50
Use Value Labels 149

V
Value Labels 149
value ordering
    property parameters 154
Values 124
Vari Chart Force REML 219
Variability Chart 219
variable value tooltips, showing in script editor 51
variables
    global, hiding 408
    variables, local
        locking and unlocking 102
VConcat 355, 372
VecDiag 355
VecQuadratic 356
vector 351
    glossary 639
    vectors
        normal 339
Vertex 331
VLine 288
VList 273
VList Box 14, 18, 227, 307

W-Z
Wait 88, 101
Watch 101
Web 431
Week Of Year 79
While 96, 282
wildcard 231
window
    custom 240
    window scripting 225
Windows 86
Wrap point, controlling 244
Write 86–87, 415
X Function 283
XML
    Parse XML 409
    parsing example 409
xmlAttr 409
xmlText 409
xmlAttr 409
xmlText 409
Y Function 282
Y2K date interpretation 80
Year 79
year 2000 date interpretation 80
Zero Or Missing 78
Zoom Window 189, 232
Your Turn

We welcome your feedback.

- If you have comments about this book, please send them to yourturn@sas.com. Include the full title and page numbers (if applicable).

- If you have comments about the software, please send them to suggest@sas.com.
SAS® Publishing gives you the tools to flourish in any environment with SAS!

Whether you are new to the workforce or an experienced professional, you need to distinguish yourself in this rapidly changing and competitive job market. SAS® Publishing provides you with a wide range of resources to help you set yourself apart.

**SAS® Press Series**

Need to learn the basics? Struggling with a programming problem? You’ll find the expert answers that you need in example-rich books from the SAS Press Series. Written by experienced SAS professionals from around the world, these books deliver real-world insights on a broad range of topics for all skill levels.

support.sas.com/saspress

**SAS® Documentation**

To successfully implement applications using SAS software, companies in every industry and on every continent all turn to the one source for accurate, timely, and reliable information—SAS documentation. We currently produce the following types of reference documentation: online help that is built into the software, tutorials that are integrated into the product, reference documentation delivered in HTML and PDF—free on the Web, and hard-copy books.

support.sas.com/publishing

**SAS® Learning Edition 4.1**

Get a workplace advantage, perform analytics in less time, and prepare for the SAS Base Programming exam and SAS Advanced Programming exam with SAS® Learning Edition 4.1. This inexpensive, intuitive personal learning version of SAS includes Base SAS® 9.1.3, SAS/STAT®, SAS/GRAPH®, SAS/QC®, SAS/ETS®, and SAS® Enterprise Guide® 4.1. Whether you are a professor, student, or business professional, this is a great way to learn SAS.

support.sas.com/LE