

**MATRIX<sub>x</sub>**<sup>®</sup>

**7.0**

**XMATH USER'S GUIDE**



---

Copyright © 2000 Wind River Systems, Inc.

ALL RIGHTS RESERVED. No part of this publication may be copied in any form, by photocopy, microfilm, retrieval system, or by any other means now known or hereafter invented without the prior written permission of Wind River Systems, Inc.

AutoCode, Embedded Internet, ES*p*, Fast*J*, IxWorks, MATRIX<sub>X</sub>, pRISM, pRISM+, pSOS, RouterWare, Tornado, VxWorks, *wind*, WindNavigator, Wind River Systems, WinRouter, and Xmath are registered trademarks or service marks of Wind River Systems, Inc.

BetterState, Doctor Design, Embedded Desktop, Envoy, How Smart Things Think, HTMLWorks, MotorWorks, OSEKWorks, PersonalJWorks, pSOS+, pSOSim, pSOSystem, SingleStep, SNiFF+, VxDCOM, VxFusion, VxMP, VxSim, VxVMI, Wind Foundation Classes, WindC++, WindNet, Wind River, WindSurf, and WindView are trademarks or service marks of Wind River Systems, Inc. This is a partial list. For a complete list of Wind River trademarks and service marks, see the following URL:

**<http://www.windriver.com/corporate/html/trademark.html>**

Use of the above marks without the express written permission of Wind River Systems, Inc. is prohibited. All other trademarks mentioned herein are the property of their respective owners.

#### **Corporate Headquarters**

Wind River Systems, Inc.  
500 Wind River Way  
Alameda, CA 94501-1153  
U.S.A.

toll free (U.S.): 800/545-WIND  
telephone: 510/748-4100  
facsimile: 510/749-2010

For additional contact information, please visit the Wind River URL:

**<http://www.windriver.com>**

For information on how to contact Customer Support, please visit the following URL:

**<http://www.windriver.com/support>**

# Contents

<b>1</b>	<b>Introduction .....</b>	<b>21</b>
1.1	Using This Manual .....	22
1.1.1	Document Organization .....	22
1.1.2	Commonly-Used Nomenclature .....	23
1.1.3	Conventions .....	24
1.1.4	Related Publications .....	24
1.1.5	Online Help .....	24
1.2	Environment Variables .....	25
1.2.1	ISIHOMES .....	25
1.2.2	XMATH .....	25
1.2.3	XMATH_STARTUP .....	25
1.2.4	XMATH_PRINT .....	26
1.2.5	PRINTER .....	26
1.3	Starting and Stopping Xmath .....	27
1.3.1	Starting Xmath .....	27
	Starting Xmath on UNIX Systems .....	27
	Starting Xmath on Windows Systems .....	29
1.3.2	Interrupting or Terminating Xmath .....	30

1.3.3	Exiting Xmath .....	30
1.3.4	Stopping and Restarting Xmath .....	31
1.4	Licensing .....	32
1.5	Using Xmath Windows .....	33
1.5.1	Mouse Conventions .....	33
1.5.2	Scroll Bars .....	34
1.5.3	Resizing Xmath Windows .....	34
1.5.4	Menus .....	34
1.5.5	Meta Key .....	35
1.6	Xmath Commands Window .....	35
1.6.1	Menus .....	37
1.6.2	Log Area .....	37
1.6.3	Command Area .....	38
	Specifying Directory Pathnames and Filenames .....	38
	Entering Multiple Lines of Information .....	39
	Editing Text by Selecting, Copying, and Pasting .....	40
	Key Bindings Used in Editing Text .....	41
	Recalling Previous Commands .....	43
1.6.4	Message Area .....	44
1.7	Help Window .....	45
<b>2</b>	<b>JumpStart: A Tutorial .....</b>	<b>47</b>
2.1	Starting Xmath for the Tutorial .....	48
2.2	Basic Data-Handling .....	48
2.2.1	Creating Variables .....	49
2.2.2	Variables and Partitions .....	51
2.2.3	Viewing Data .....	53
2.2.4	Saving Data .....	53

	Save Command .....	54
	Print Command .....	55
2.2.5	Loading Data .....	55
	Load Command .....	55
	Read Command .....	56
2.2.6	Cleanup .....	56
2.3	Functions and Commands .....	57
2.3.1	Function Syntax .....	57
2.3.2	Command Syntax .....	58
2.4	Graphics .....	58
2.4.1	Plot() .....	59
	Keywords .....	59
	Graph Objects .....	59
2.4.2	Working in the Xmath Graphics Window .....	60
2.4.3	Using Plot and Graph Objects .....	61
	Using 2D Plotting Capabilities .....	61
	Using 3D Plotting Capabilities .....	63
2.4.4	Using Different Plot Types .....	64
	Strip Plots .....	65
	Polar Plots .....	66
	Bar Plots .....	66
	Contour Plots .....	67
2.4.5	Displaying Multiple Plots at Once .....	68
2.4.6	Animating Plots .....	69
2.4.7	Finishing the Graphics Tutorial .....	70
2.5	Objects .....	70
2.5.1	Strings .....	71
2.5.2	Matrices and Vectors .....	72
	Creating Matrices and Vectors .....	72
	Matrix Index Operations .....	74

	Using Matrix Functions .....	75
2.5.3	Polynomials .....	76
2.5.4	Dynamic Systems .....	78
	Transfer Functions .....	78
	State-Space Systems .....	79
	Analyzing Dynamic Systems .....	80
2.5.5	Parameter Dependent Matrices .....	81
2.5.6	Lists .....	84
2.6	MathScript .....	85
2.6.1	MathScript Features .....	85
2.6.2	Debugger Window (on UNIX Systems) .....	86
2.7	GUI Tools .....	88
2.8	Conclusion .....	88
<b>3</b>	<b>MathScript Basics .....</b>	<b>89</b>
3.1	MathScript Statements .....	89
3.1.1	Assignments .....	89
3.1.2	Rules for Names .....	90
3.1.3	Expressions .....	90
	Logical Expressions .....	91
	Logical Expressions with Matrices .....	92
3.1.4	Operators .....	92
	Operator Precedence .....	94
3.2	Partitions .....	95
3.2.1	Listing Defined Variables .....	96
	Wildcards .....	97
3.2.2	Variable and Partition Comments .....	97
3.2.3	Permanent Variables .....	98

3.2.4	ans .....	99
3.2.5	Xmath Variables Window .....	99
	Fields .....	100
	Menus .....	101
3.3	Punctuation .....	102
3.4	Iterative Conditional Statements .....	104
3.5	Using Predefined Functions and Commands .....	105
3.5.1	Command and Function Calling Syntax .....	105
	Aliases .....	106
	Input Arguments .....	106
	Keywords .....	106
	Single and Multiple Output Arguments .....	107
3.6	Operating System Interface .....	108
3.6.1	Manipulate and Show Current Directory .....	108
3.7	Saving and Loading Data .....	108
3.7.1	ASCII Versus Binary Considerations .....	110
3.7.2	Saving Data in Non-Xmath Formats .....	111
	print .....	111
	fprintf() .....	111
3.7.3	Reading Non-Xmath Data Files into Xmath .....	112
3.8	MathScript Environment .....	112
3.8.1	Changing Environment Settings .....	112
3.8.2	Expanding Pathnames in MathScript Files .....	115
3.8.3	Abbreviating Command Names (alias and unalias) .....	116
3.9	MathScript Batch Files .....	117
3.9.1	Executing a Batch File .....	117
3.9.2	Echoing an Executable File .....	118

3.9.3	startup.ms (on UNIX systems) .....	118
3.9.4	startup.ms (on Windows Systems) .....	119
3.9.5	I/O Redirection .....	120
3.10	Recording an Xmath Session (Diaries) .....	121
3.10.1	Recording Inputs (Command Diary) .....	121
3.10.2	Recording Inputs and Outputs (Session Diary) .....	122
<b>4</b>	<b>Graphics .....</b>	<b>123</b>
4.1	Xmath Plotting Functions and Commands .....	123
4.1.1	General Purpose Plotting Functions .....	123
	plot() .....	124
	uiPlot() .....	124
	plot2d() .....	125
4.1.2	Comparative Analysis: plot() versus plot2d() .....	125
4.1.3	Plotting Commands and Special Purpose Functions: .....	126
	colorind .....	126
	ERASE .....	127
	HARDCOPY .....	127
	pdmplot .....	127
	qplot .....	127
	uiPlotArea .....	127
	uiPlotGet .....	127
4.2	Using the plot() Function .....	128
4.2.1	Plot One Input .....	129
4.2.2	Plot Two Inputs .....	129
4.2.3	Plot Three Inputs .....	129
4.2.4	Color as a Fourth Dimension .....	130
4.2.5	Creating and Displaying a Graph Object .....	130
4.3	Using Keywords with plot .....	131
4.3.1	Labels and Legend .....	135



4.3.2	Colors .....	137
4.3.3	Line and Marker Specifications for Data .....	140
4.3.4	Multiple Graphs and Graph Positioning .....	143
4.3.5	Adding New Data to Existing Plots (keep, copy) .....	145
4.3.6	Axis and Zero Lines .....	148
4.3.7	Tics and Grids .....	149
4.3.8	Free Text and Global Text Settings .....	151
4.3.9	Axis Limits and Logarithmic Scaling .....	153
4.3.10	Animate .....	155
4.3.11	Placement, Scaling, and Rotation .....	156
4.3.12	Background, Edge, and Face Settings .....	158
4.3.13	Lighting Source Settings .....	160
4.3.14	Reusing plot Attributes .....	161
	Hold Keyword .....	161
	Using an Alias in the Keyword String .....	165
4.3.15	Strip Plots .....	165
4.3.16	Bar Plots .....	167
4.3.17	Contour Plots .....	168
4.3.18	Polar Plots .....	169
4.3.19	Clearing the Xmath Graphics Window .....	171
4.4	Interactive Xmath Graphics Window .....	171
4.4.1	Working Interactively .....	174
4.4.2	Toolbar .....	175
	Selection Arrow .....	175
	Text Tool .....	175
	Drawing Tools .....	176
	Zoom In/Zoom Out .....	177
	Rotation Tools .....	177
4.4.3	Menus .....	179

	File .....	179
	Edit .....	181
	View .....	181
	Options .....	181
	Font (UNIX Only) .....	182
	Point (UNIX Only) .....	182
	Tools (Windows Only) .....	182
	Windows .....	183
4.4.4	Xmath Palette .....	183
<b>5</b>	<b>Data Objects and Operators .....</b>	<b>187</b>
5.1	Data Hierarchy .....	187
5.1.1	Data Object Descriptions .....	189
5.2	Matrix .....	190
5.2.1	Matrix Concatenation .....	191
5.2.2	Matrix Operators .....	191
5.2.3	Matrix Indexing .....	193
	Indexing with the Colon Operator (:) .....	194
5.2.4	Vector .....	195
	Regular Vector .....	196
	Logspaced Vector .....	197
5.2.5	Square Matrix .....	197
	Symmetric .....	198
	Diagonal() .....	199
	Identity .....	199
	Toeplitz .....	200
	Hessenberg() .....	200
	Triangular .....	201
	Scalar .....	201
5.3	Polynomial() .....	205
5.3.1	Polynomial Operators .....	206
5.4	Parameter-Dependent Matrix (PDM) .....	207

5.4.1	PDM Organization .....	209
5.4.2	Creating PDMs .....	210
5.4.3	Default PDM Behavior .....	211
5.4.4	PDM Channels .....	214
5.4.5	Indexing to Extract Portions of a PDM .....	215
	PDM Dimensions .....	215
	Dependent Matrices .....	215
	Domain and Name Information .....	218
5.4.6	Modifying PDMs .....	220
	Substitution .....	220
	Concatenation .....	221
	Converting PDMs to Matrices .....	222
5.4.7	Using PDMs with Operators .....	223
5.4.8	Using Functions with PDMs .....	225
5.5	Dynamic System .....	227
5.5.1	State-Space Systems .....	227
5.5.2	Transfer Functions .....	228
5.5.3	Creating Systems .....	229
	Using Operators with Dynamic Systems .....	230
	Creating Subsystems by Indexing into Dynamic Systems .....	231
5.5.4	Functions for Manipulating Dynamic System Objects .....	233
5.5.5	Time Response .....	234
5.6	Strings .....	235
5.6.1	Converting Strings and Numbers .....	236
5.6.2	Special Characters in Strings .....	236
5.6.3	Manipulating Substrings .....	237
5.7	Lists .....	238
5.8	Index Lists .....	239

<b>6</b>	<b>MathScript Programming .....</b>	<b>243</b>
6.1	Overview .....	243
6.1.1	Creating a Sample MSF .....	243
6.1.2	Creating a Sample MSC .....	244
6.1.3	General Rules for MathScript Programs .....	246
6.1.4	MathScript File Formats .....	246
6.1.5	MathScript Programming .....	248
	Assigning Default Values .....	248
	Output Keywords .....	248
	Calling Void Functions .....	249
	Variable Scoping .....	249
6.1.6	Creating Online Help for User-Defined MSFs and MSCs .....	249
6.1.7	Using User-Defined MSFs and MSCs .....	250
	Search Paths .....	251
	Manipulating Search Paths .....	251
	DEFINE .....	252
	MathScript Program Compilation and Execution (.xf, .xc) .....	253
6.2	Examples .....	254
6.3	Programming .....	256
6.3.1	Iterative and Conditional Looping Statements .....	256
	For .....	256
	While .....	256
	If .....	257
	Goto and Labels .....	258
6.3.2	Object Query Functions .....	258
	exist() .....	258
	check() .....	259
	is() .....	260
6.3.3	User Interface Functions .....	260
	getline() .....	261
	getchoice() .....	261

	pause() .....	262
	error() .....	263
	beep() .....	263
6.3.4	Indexing Functions .....	264
	index() .....	264
	find() .....	264
6.4	Using the Xmath Debugger .....	265
6.4.1	Debug .....	266
6.4.2	Debug Mode .....	266
6.4.3	Setting, Showing, and Removing Breakpoints .....	268
6.4.4	Setting and Removing Watchpoints .....	269
6.4.5	Debugger Window Interface .....	270
6.5	Advanced Topics .....	272
6.5.1	Variable Arguments .....	272
	argn() .....	272
	argv() .....	272
	Using argn and argv .....	273
6.5.2	Executing a Function at a Specific Directory .....	276
6.5.3	Partition and Variable Directory Functions .....	276
6.5.4	MathScript Command Output and Error Capture .....	276
6.5.5	Programming for Platform Independence .....	278
<b>7</b>	<b>MathScript Objects .....</b>	<b>279</b>
7.1	MSO Overview .....	279
7.1.1	Object Instantiation .....	280
7.1.2	MSO File Format .....	280
7.1.3	Using MSOs in Xmath .....	281
7.2	Initializer Function .....	282
7.2.1	Class Variables .....	282

7.2.2	Nested Objects .....	283
7.2.3	Type Declaration .....	284
7.3	Operator Overloading .....	285
7.4	Member Functions .....	289
7.4.1	Sample MSO .....	289
7.4.2	Limitations .....	292
<b>8</b>	<b>External Program Interface .....</b>	<b>293</b>
8.1	Overview .....	293
8.1.1	LNx .....	294
8.1.2	UCI Programs .....	296
8.1.3	Compatibility .....	297
8.2	externType Data Types .....	297
8.2.1	Matrix Data Type .....	297
8.2.2	String Data Type .....	298
8.2.3	PDM Data Type .....	299
8.2.4	List Data Type .....	302
8.2.5	Null Data Type .....	303
8.3	LNx and UCI Functions .....	303
8.3.1	XmathMain( ) (for LNx only) .....	304
8.3.2	XmathCommand( ) .....	306
8.3.3	XmathDisplay( ) .....	307
8.3.4	XmathError( ) .....	307
8.3.5	XmathExecute( ) .....	308
8.3.6	XmathGet( ) and XmathPut( ) .....	308
	XmathGet( ) .....	309
	XmathPut( ) .....	309

8.3.7	Example Using XmathGet(), XmathPut(), and XmathExecute()	310
8.3.8	XmathSave() and XmathLoad()	311
	XmathSave()	311
	XmathLoad()	312
	Standard Library Linkage	312
	Example of XmathSave and XmathLoad	312
8.3.9	XmathStart() and XmathStop()	314
	XmathStart()	314
	XmathStop()	314
8.3.10	Sample LNX Demonstrating Most Functions (myfun)	314
8.4	Building and Calling LNX and UCI	316
8.4.1	Building on a UNIX System	316
8.4.2	Sample makefile (UNIX)	317
8.4.3	Building on a Windows System	319
8.4.4	Undefining an LNX	320
8.4.5	Using the User-Callable Interface	320
8.4.6	Building and Calling a UCI	321
8.4.7	LNX Example	321
8.4.8	UCI Examples	322
8.4.9	Calling an LNX in Background Mode	324
8.4.10	Removing an LNX Job	327
8.4.11	Building an LNX to Link a FORTRAN Routine	327
	Calling FORTRAN from C LNX Files	328
	Creating FORTRAN LNX Files	328
8.5	Debugging	330
8.5.1	Debugging an LNX with dbx (on UNIX Systems)	330
8.5.2	Debugging LNXs (on Windows systems)	332
8.5.3	Debugging UCIs (on UNIX systems)	333
8.5.4	Debugging UCIs (on Windows systems)	333

8.6	Advanced Topics .....	334
8.6.1	Handling an Aborted LNX .....	334
8.6.2	Advanced Features and Notes .....	335
8.6.3	Advanced Background LNX Function (IPCWC) .....	335
<b>9</b>	<b>Graphical User Interface .....</b>	<b>341</b>
9.1	Finding Out About the GUI .....	341
9.1.1	GUI Tool Users .....	342
9.1.2	GUI Developers .....	342
9.1.3	Running the GUI Demos .....	342
9.2	Interacting with a GUI Application .....	344
9.2.1	Creating an Example Dialog .....	344
9.2.2	Controlling GUI Objects .....	345
9.3	GUI Programming Overview .....	348
9.4	Concepts and Terminology .....	348
9.4.1	Conceptual Example .....	349
9.4.2	Anatomy of a GUI Tool .....	350
9.4.3	MSC File .....	351
9.4.4	Help File .....	352
9.5	Xmath GUI Functions .....	353
9.6	Tutorial .....	354
9.6.1	Pushbutton .....	354
9.6.2	Calculator .....	357
9.7	Translating Version 5.X GUI Files to Version 6.X PGUI Files .....	361
9.7.1	Overview .....	361
9.7.2	Execution .....	361



9.7.3	Details .....	362
9.7.4	Limitations .....	362
<b>A</b>	<b>X Windows and Motif .....</b>	<b>365</b>
A.1	X Window System .....	365
A.1.1	Starting X .....	366
A.1.2	X Terminology .....	366
A.2	Motif Window Manager .....	367
A.2.1	Motif Frame Components .....	368
	Default Window Menu .....	368
	Frame Buttons .....	369
	Window Operations .....	369
A.2.2	Mouse Focus and the Pointer .....	370
A.2.3	Copying and Pasting with Motif .....	371
A.2.4	Using Menus Without the Mouse .....	371
A.2.5	Using a Motif File Selection Dialog .....	372
A.3	Changing Resource Parameters .....	373
A.3.1	Remapping Your Keyboard .....	375
A.3.2	Sizing and Placing Windows .....	377
<b>B</b>	<b>Xmath HP-GL Driver .....</b>	<b>379</b>
B.1	Supported Devices .....	379
B.2	Setting the Aspect Ratio .....	380
B.3	Color Pen Specifications .....	380
<b>C</b>	<b>Xmath for MATLAB Users .....</b>	<b>381</b>
C.1	Syntactic Differences .....	382
C.1.1	Continuation .....	382

C.1.2	Output Display .....	382
C.1.3	Matrix Punctuation .....	383
C.1.4	String Punctuation .....	383
C.1.5	Logical Not .....	384
C.1.6	Comments .....	384
C.1.7	Function Names .....	385
C.1.8	RAND, ONES, ZEROS, and EYE .....	385
C.1.9	IF, FOR, and WHILE .....	386
C.1.10	Pure Imaginary Number .....	386
C.2	Object Differences .....	386
C.2.1	Strings .....	387
C.2.2	Polynomials .....	387
C.2.3	Dynamic Systems .....	387
C.3	Interpretation Differences .....	388
C.3.1	Environment Commands .....	388
	Creating Diaries .....	389
	Random Seeds and Distribution .....	389
	Number Formatting .....	389
C.3.2	User-Defined Functions and Commands .....	390
C.3.3	plot() .....	391
C.3.4	Transpose Operators .....	391
C.3.5	Convolve .....	392
C.3.6	Series and Parallel .....	392
C.3.7	Simulation .....	392
C.3.8	Eval (Executable Strings) .....	393
C.3.9	Executable Files .....	393
C.3.10	Finding Files .....	394
C.3.11	Debugging Files (on UNIX systems) .....	395

C.3.12	Save and Load .....	395
	Loading In External Data (read) .....	396
	Writing Data to an External File (print, fprintf) .....	397
C.3.13	Useful Aliases .....	397
C.4	Comparison of Frequently Used Commands .....	398
<b>D</b>	<b>Xmath to Mathematica Interface .....</b>	<b>403</b>
D.1	Overview .....	403
D.2	Setup .....	404
	D.2.1 Setting Up the Xmath to Mathematica Interface for All Users ...	405
	D.2.2 Creating a Local LNX (Single User) .....	405
D.3	Syntax .....	406
	D.3.1 Passing Xmath Data to Mathematica .....	407
	D.3.2 Passing Mathematica Data to Xmath .....	407
D.4	Examples .....	407
<b>Index</b>	<b>.....</b>	<b>411</b>



# 1

## *Introduction*

Xmath<sup>®</sup> is a mathematical analysis, visualization, and scripting package that is one of the five main products of the MATRIX<sub>X</sub><sup>®</sup> product family. Complementing SystemBuild<sup>™</sup>, another member of the MATRIX<sub>X</sub> product family, Xmath serves not only as an analytical tool, but also as a working environment and visualization tool for simulation data. Xmath and SystemBuild run concurrently, which allows you to simultaneously edit SystemBuild models, perform Xmath analysis or SystemBuild simulations, and display 2D and 3D graphics in presentation quality.

MathScript, the Xmath programming language, provides unique object-oriented capabilities that facilitate design analysis. Xmath also offers an interactive debugger, a programmable graphical user interface (GUI) layer, and an extensive library of mathematical, system modeling, and analysis functions.

This chapter begins with an outline of the *Xmath User's Guide*, and some use notes. It continues with topics for helping you to get started in Xmath. These basic tasks are divided into the following topics:

*Environment Variables*

*Starting and Stopping Xmath*

*Licensing*

*Using Xmath Windows*

*Xmath Commands Window*

*Help Window*

To complete the exercises in this chapter, Xmath must be properly installed according to the *System Administrator's Guide* for your operating system and platform. For details about X Windows and the Motif window manager, see Appendix A. *X Windows and Motif*. For more information about Windows operating systems, see the appropriate *System Administrator's Guide, Windows Version*.

## 1.1 Using This Manual

This manual discusses Xmath structure and concepts. Chapter 2 is a tutorial. Chapters 3, 4, and 5 cover basic features for general Xmath use. Chapters 6 through 9 describe more advanced aspects of Xmath's structure and its programming abilities. Appendices A–D contain material that is only of interest to specific categories of users. A glossary, which includes some general terms as well as Xmath terms, follows the appendices.

### 1.1.1 Document Organization

This manual includes the following chapters and appendices:

- *1. Introduction* starts with an outline of the *Xmath User's Guide*. It continues with a discussion of useful environmental variables and licensing issues. It also tells how to start and exit Xmath, and introduces the Xmath Commands window and the Xmath Help window.
- *2. JumpStart: A Tutorial* covers Xmath's basic and intermediate capabilities and introduces some of Xmath's more advanced features and concepts.
- *3. MathScript Basics* introduces Xmath's object-oriented language, MathScript, and data management in Xmath.
- *4. Graphics* details the `plot` function, providing a complete listing of all keywords and many examples. This chapter also describes how to change a plot's appearance interactively.
- *5. Data Objects and Operators* discusses the nature and definition of each of Xmath's object classes. It gives examples of how to build and use each object.

- [6. MathScript Programming](#) discusses how to create different types of MathScript files, MathScript Functions (MSFs), and MathScript Commands (MSCs). This chapter also includes brief descriptions of Xmath-supplied functions and commands designed to help you program in MathScript.
- [7. MathScript Objects](#) describes how to define MathScript objects.
- [8. External Program Interface](#) explains the LNX and User-Callable Interface (UCI) features. The LNX facility makes it possible to link C, C++, or FORTRAN subroutines into Xmath. The UCI allows your external programs to use Xmath for graphics and computation.
- [9. Graphical User Interface](#) describes Xmath's programmable graphical user interface (PGUI).
- [A. X Windows and Motif](#) is included for users who are unfamiliar with the workstation environment but want to start using Xmath quickly. This appendix provides a summary of the X and Motif actions used most frequently in Xmath.
- [B. Xmath HP-GL Driver](#) discusses Xmath's HP-GL driver and the devices it supports.
- [C. Xmath for MATLAB Users](#) is designed to help MATLAB users transition to Xmath. Differences in syntax, behavior, and functionality are discussed.
- [D. Xmath to Mathematica Interface](#) describes how to set up and use the Xmath to Mathematica Interface.

### 1.1.2 Commonly-Used Nomenclature

This manual uses the following general nomenclature:

- Matrix variables are generally denoted with capital letters; vectors are represented in lowercase.
- $G(s)$  is used to denote a transfer function of a system where  $s$  is the Laplace variable.  $G(q)$  is used when both continuous and discrete systems are allowed.
- $H(s)$  is used to denote the frequency response, over some range of frequencies of a system where  $s$  is the Laplace variable.  $H(q)$  is used to indicate that the system can be continuous or discrete.
- A single apostrophe following a matrix variable, for example,  $\mathbf{x}'$ , denotes the transpose of that variable. An asterisk following a matrix variable (for

example,  $A^*$ ) indicates the complex conjugate, or Hermitian, transpose of that variable.

### 1.1.3 Conventions

This publication makes use of the following types of conventions: font, format, symbol, mouse, and note. These conventions are detailed in Chapter 2 of the *MATRIX<sub>X</sub> Getting Started Guide*.

### 1.1.4 Related Publications

For a complete list of MATRIX<sub>X</sub> publications, see Chapter 2 of the *MATRIX<sub>X</sub> Getting Started Guide*. The following documents are particularly useful for topics covered in this manual:

- *MATRIX<sub>X</sub> Getting Started Guide*
- *Xmath User's Guide*
- *Control Design Module*
- *Interactive Control Design Module*
- *Interactive System Identification Module, Part 1*
- *Interactive System Identification Module, Part 2*
- *Model Reduction Module*
- *Optimization Module*
- *Robust Control Module*
- *X $\mu$  Module*

### 1.1.5 Online Help

Xmath function reference information is available in the MATRIX<sub>X</sub> online Help. The online Help includes all Xmath functions. Each topic explains a function's inputs, outputs, and keywords in detail. See Chapter 2 of the *MATRIX<sub>X</sub> Getting Started Guide* for complete instructions on using the Help feature.



## 1.2 Environment Variables

This section defines several important environment variables.



---

**NOTE:** The following conventions are used in this manual when referring to environment variables:

- When an environment variable appears in a pathname with its appropriate system dependent environment variable designator ( $\$NAME$  for UNIX and  $\%NAME\%$  for Windows), then you can use the environment variable as shown.
  - When  $NAME$  appears without the environment variable designator, then you must substitute the pathname (value of the variable) in the command.
- 

Xmath defines the **ISIHOME** and **XMATH** environment variables. It also recognizes the other environment variables discussed below. You can define them in your `.cshrc` file (UNIX), in your `autoexec.bat` file (Windows), or in your system properties (environment) (Windows NT). Alternatively, you can define them in each session in an Terminal or Command Prompt window.

### 1.2.1 ISIHOME

**ISIHOME** is an environment variable representing the installation directory for `MATRIXx`. This variable is used in pathnames.

### 1.2.2 XMATH

**XMATH** is an environment variable representing the directory in which Xmath is installed. **XMATH** is used in pathnames.

### 1.2.3 XMATH\_STARTUP

**XMATH\_STARTUP** is an environment variable you can use to specify a directory in which the startup MathScript file (**startup.ms**) is located. When you launch Xmath, the startup MathScript file (**startup.ms**) in the specified directory is executed.

## 1.2.4 XMATH\_PRINT

**XMATH\_PRINT** is an environment variable that lets you set up a default printer. When you run Xmath and use the **HARDCOPY** command, Xmath uses the value of **XMATH\_PRINT** to send the graphics to the printer.

To define **XMATH\_PRINT** for a SunOS system using the print command **lpr** and a printer named **hp0**, define **XMATH\_PRINT**:

```
setenv XMATH_PRINT "lpr -Php0"
```

If you are on an SGI or HP system, set **XMATH\_PRINT** with an entry similar to the following:

```
setenv XMATH_PRINT "lp -dhp0 -c"
```

If you are on a Windows operating system, set **XMATH\_PRINT** with an entry similar to the following:

```
set XMATH_PRINT=ISIHOME\xmath\bin\xmprint your_printer
```

where *your\_printer* is the name of your selected printer.

You can place this command in the **autoexec.bat** file in the root directory of your C drive. On Windows NT, you have the alternative of using the System Properties, Environment tab under the Control Panel to specify the environment variable.



---

**NOTE:** If you specify the **XMATH\_PRINT** environment variable, you do not need to set the **PRINTER** environment variable. (Xmath ignores it.)

---

## 1.2.5 PRINTER

**PRINTER** is an environment variable that lets you specify a default printer (if **XMATH\_PRINT**) is not defined.

For example, to define **PRINTER** on a SunOS system, for a printer named **hp0**, define the **PRINTER** environment variable in your **.cshrc** file with the following:

```
setenv PRINTER "hp0"
```

The next time you run Xmath and use the **HARDCOPY** command, Xmath will use the value of **PRINTER** to send the graphics to the printer.



---

**NOTE:** Wind River recommends the `XMATH_PRINT` environment variable because it allows for platform-specific parameters. `PRINTER` may fail to work on some systems.

---

## 1.3 Starting and Stopping Xmath

This section covers starting and stopping Xmath, as well as terminating Xmath (abnormally) and quitting and restarting Xmath at the same point in your process. Major topics include:

- [Starting Xmath](#)
- [Interrupting or Terminating Xmath](#)
- [Exiting Xmath](#)
- [Stopping and Restarting Xmath](#)

### 1.3.1 Starting Xmath

Starting Xmath is a little different on UNIX and Windows machines, and the options available are also different. Therefore, we have included sections for each operating system.

#### **Starting Xmath on UNIX Systems**

You can start Xmath from any directory in any Terminal window, either in the foreground or the background.

#### **Starting Xmath Locally**

To start Xmath:

1. Bring up a Terminal window.

A Terminal window allows you to input at the operating system prompt.

2. Enter the following command:

```
% xmath
```

Unless all licenses are in use (see 1.4, p.32), the Xmath Commands window appears after a few seconds.

Internal messages and warnings from Xmath may be written to the Terminal window.

### **Starting Xmath on a Remote X Host**

If you want to run Xmath on a remote UNIX host, you can start it from your local machine or from the remote host itself.

To start Xmath from your local computer, type:

```
% xmath -host remoteHostName
```

The remote host must accept a remote shell (rsh). Be aware that when the operating system stores the name of the current working directory, the name may not be equivalent to that of the same directory on the remote host. (For example, **/home/user** on the local machine versus **/net/machine/home/user** on the remote machine.) When there is no verbatim match, Xmath will start in your home directory on the remote machine.

To confirm your location, go to the Xmath Commands window command area and type **show directory**. If necessary, use **set directory** to change the working directory from within Xmath.

To start Xmath from the remote host, type:

```
% xmath -d localHostName:0.0
```

This command displays the Xmath session on your local machine; you need to make sure your local machine accepts the display from a remote host. Consult the documentation on the UNIX operating system command **xhost**.

### **Command-Line Options Available on UNIX**

Table 1-1 contains a partial list of options; some options might not be available on your platform.

Table 1-1 Commonly Used Startup Options for UNIX

Switch	Action
<b>-tty</b>	Start the tty (non-windowing) version. This version is suitable for command-line calculations. It can also be used to submit a list of instructions in batch mode (see <a href="#">3.9 MathScript Batch Files</a> ). The tty version has no online Help or graphics capabilities.
<b>-call <i>name args</i></b>	Runs a user-callable interface (UCI) executable, where <i>name</i> is the image name and <i>args</i> can be any command line arguments required by the UCI.
<b>-clean</b>	If a UCI has terminated abnormally you can run Xmath with this switch to clean up orphaned processes. No other switches are accepted when <b>-clean</b> is specified.

To get help on the **xmath** command in a Terminal window, type:

```
xmath -h
or
xmath -help
```

### Starting Xmath on Windows Systems

To start Xmath on a PC, use one of the following methods:

- Select Start→Programs→MATRIXx xx.x→Xmath
- Enter the following command from the Command Prompt window:

```
SIHOME\bin\xmath
```

where *SIHOME* represents the installation directory for MATRIX<sub>x</sub>.

[Table 1-2](#) contains a partial list of options; some options might not be available on your platform.

Table 1-2 **Commonly Used Startup Options for Windows**

Switch	Action
<b>-call</b> <i>name</i> <b>args</b>	Runs a user-callable interface (UCI) executable, where <i>name</i> is the image name and <i>args</i> can be any command line arguments required by the UCI.
<b>-clean</b>	If a UCI has terminated abnormally you can run Xmath with this switch to clean up orphaned processes. No other switches are accepted when <b>-clean</b> is specified.

Internal messages and warnings from Xmath may be written to the Command Prompt window.

To get help on the **xmath** command in the Command Prompt window, type:

```
SIHOME\bin\xmath -h
```

or

```
SIHOME\bin\xmath -help
```

where you provide the path for the root installation directory of MATRIX<sub>X</sub> (*SIHOME*).

### 1.3.2 Interrupting or Terminating Xmath

To interrupt interactive execution of an Xmath function or command, press Ctrl-C (on UNIX systems) or Ctrl-Break (on Windows systems) from any Xmath window.



**NOTE:** Intrinsic commands (for example, **save** or **load**; see [3.5 Using Predefined Functions and Commands](#)) are noninterruptible. The same is true for window, dialog, or plot creation.

On UNIX systems, if either the windowing version or the tty version is not responding, terminate your Xmath session by pressing Ctrl-\ (hold down the Control key and press the backslash key [\]). This key sequence terminates Xmath properly in unusual circumstances.

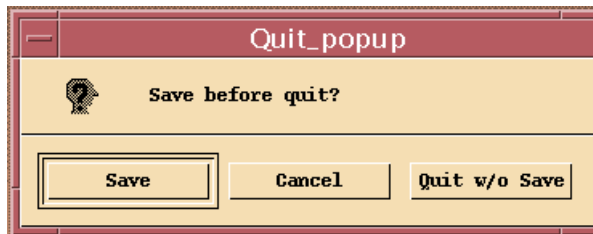
### 1.3.3 Exiting Xmath

From a windowing version of Xmath, use any one of the following methods to exit Xmath:

- Type **quit** in the Xmath Commands window command area (the only part of the Xmath Commands window that accepts input).
- Choose File→Quit from the menu bar.
- With the cursor over the Xmath Commands window, press Ctrl-q.
- On UNIX systems, select Close from the X Windows Default Menu in the Xmath Commands window.
- On Windows systems, click the X (Close) button in the upper right corner of the Xmath Commands window, or click the Xmath icon in the upper left corner of the Xmath Commands window and select Close from the system menu, or use its keyboard equivalent of ALT F4.

In all cases above, the Quit\_popup dialog may appear.

Figure 1-1 **Quit Confirmation Dialog**



You are given the opportunity to save before exiting. Selecting Save here saves all current variables to a file named **save.xmd** in the current working directory. The session terminates after the file is saved.

If you are using the tty version, type **quit**. You may see the following warning:

```
Modified variables that have not been saved exist; quit anyway? (y/n)
```

Type y (yes) or n (no) as desired. For more information, see [3.7 Saving and Loading Data](#).

### 1.3.4 Stopping and Restarting Xmath

You can quit Xmath at any time. To resume at the same point, type **save** in the Xmath Commands window command area before quitting, or select Save in the Quit dialog. This saves all existing data to a file called **save.xmd** in the current working directory.



---

**NOTE:** The **Save** command overwrites any previous **save.xmd** file in the current working directory.

---

To resume a session:

1. Restart Xmath from the same directory
2. Type **load** in the command area.  
The default save file **save.xmd** is loaded.

## 1.4 Licensing

When Xmath starts, it checks out the Xmath Core license. The license for each module is checked out when that module is started; for example, the Control Design Module is checked out when that module is started. If your site has a floating license or counted node-locked license, you may be unable to check out a particular module.

If a Core license is available, the Xmath Commands window appears after a few seconds (see [Figure 1-2](#), p.36 for the UNIX version).

To get license information for your current version:

- Select Help→On Version from the Help menu on any Xmath window.  
A pop-up appears that tells you the version, date, and platform.
- In the Xmath Commands window command area, type:

```
licenseinfo
```

A list of modules for which your site is licensed and their expiration dates appear in the log area.

For additional information about your Xmath license, see the *System Administrator's Guide* for your operating system.



## 1.5 Using Xmath Windows

The major Xmath windows are listed in the table below, along with sections in which you can find information about them. You get to these windows through the Windows menu on each of the other windows.

Table 1-3 **Major Xmath Windows**

Xmath Window	Section
Commands	<a href="#">1.6</a>
Graphics	<a href="#">4.4</a>
Variables	<a href="#">3.2.5</a>
Palette	<a href="#">4.4.4</a>
Debugger (UNIX only)	<a href="#">2.6.2</a>

This section contains general information that applies to all Xmath windows.

### 1.5.1 Mouse Conventions

This document assumes you have a 2- or 3-button mouse. From left to right, the buttons are referred to as MB1, MB2, and MB3. All instructions assume MB1 unless otherwise noted. [Table 1-4](#) lists common mouse instructions.

Table 1-4 **Common Mouse Instructions**

Instruction	Action
click	Press then quickly release MB1.
double-click	Rapidly click MB1 twice.
drag	Hold down MB1 while moving the mouse; release the button when the desired result is obtained.

The following mouse-click combinations are useful for selecting text:

- To select a word, point anywhere within the desired word and double-click.
- To select an entire line, point anywhere on the line and triple-click.

- To select all text in an Xmath window area, move the cursor into the area and quadruple-click.

## 1.5.2 Scroll Bars

Most Xmath windows have horizontal and vertical scroll bars so you can look at data that extends beyond your window border. As you can see in [Figure 1-2](#), p.36, scroll bars have a small arrow on each end and a center area with a rectangular slider.

The size of the slider depends on the amount of data out of view. In [Figure 1-2](#), the horizontal slider fills the whole area because all data is visible. The slider becomes smaller as data accumulates. To move the slider, place the mouse pointer over the slide bar and use MB1 or MB2 to drag in the desired direction. If you click MB1 or MB2 in the scroll bar and off the slider itself, the slider moves toward the point you clicked.

## 1.5.3 Resizing Xmath Windows

Most Xmath windows are divided into several areas. If you make a window shorter, you may notice that some areas get too small to be useful, or even seem to disappear. When this happens, vertically resize these subwindows.

On the right side of a window on UNIX systems, you can see a small square straddling the border between two areas. (See [Figure 1-2](#) for an example.) This is called a *grip* or a *sash*. When you place the pointer over it, the cursor changes to a cross-hairs symbol (+). Drag the grip vertically in the direction you want the area to grow or shrink. Experiment with a combination of resizing the frame and resizing the areas.

On Windows versions, you can resize the windows using standard windows techniques.

## 1.5.4 Menus

The menu bar features pulldown menus that appear on most Xmath windows, although not all menus are active in all windows. You can open menus by clicking on the menu name or dragging down from the menu name.

### 1.5.5 Meta Key

You need to know where the equivalent of the Meta key is on your keyboard if you plan to use Xmath's accelerators. Whenever the documentation or one of the menus refers to Meta, you will need to press the key appropriate to your machine, as shown in [Table 1-5](#).

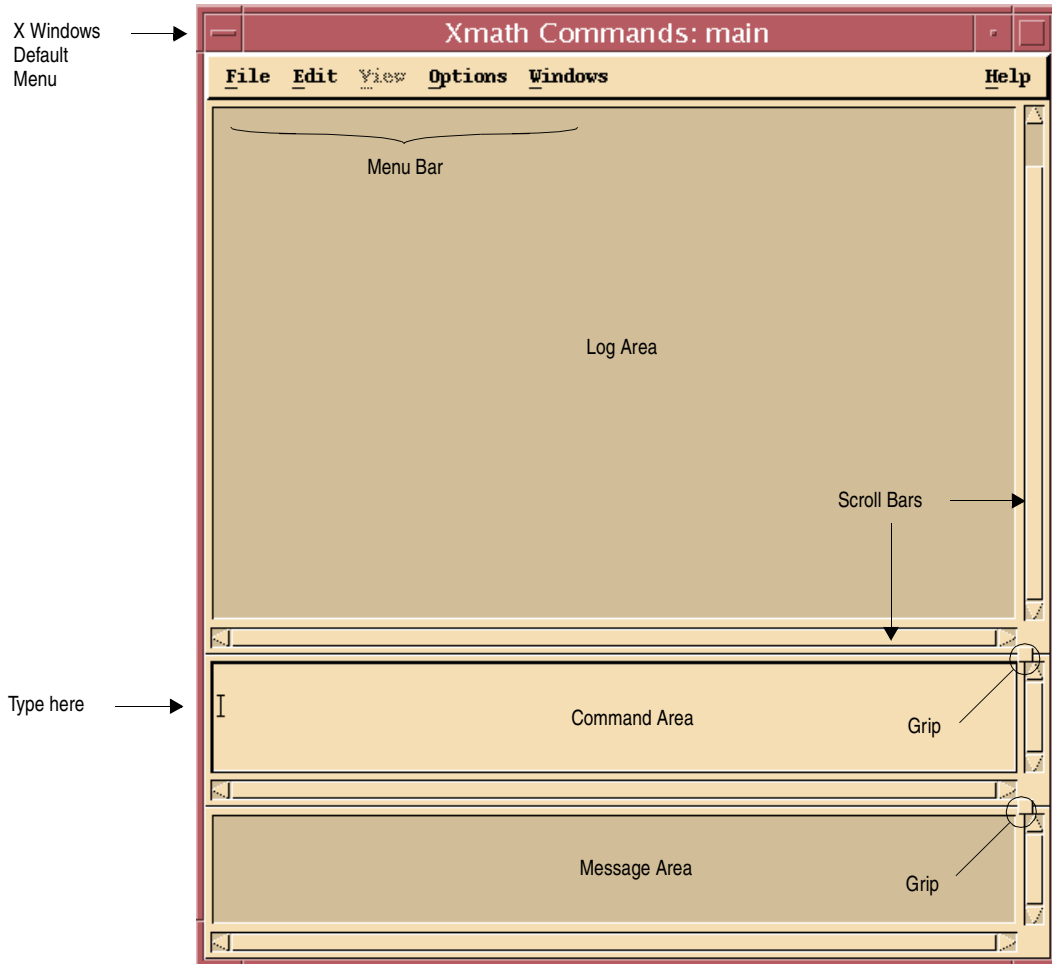
Table 1-5 **Meta Key**

Platform	Key or Key Sequence
Sun	Key with a diamond symbol (on either side of the space bar)
HP	Extend/Char key (to the left of Shift)
IBM	Alt
Windows	Alt
SGI	Alt

## 1.6 Xmath Commands Window

The Xmath Commands window appears when you start Xmath (see [Figure 1-2](#)). This is your primary interface to Xmath. On UNIX systems, the Xmath Commands window contains three primary areas: the log area, the command area, and the message area. Windows systems have only two primary areas: the log area and the command area; the information that goes to the message area in UNIX goes to the log area on Windows systems.

Figure 1-2 Xmath Commands Window (UNIX view)



You can interact with Xmath with both keyboard and mouse. While the keyboard is used for input, mouse position dictates the active input area; the mouse is also used for menu selection, text manipulation, and for displaying shortcut menus (right-click). (See 1.5.1, p.33 for mouse conventions.)

## 1.6.1 Menus

The pulldown menus shown in [Table 1-6](#) are active in the Xmath Commands window.

Table 1-6 **Xmath Menus**

Menu	Description
File	Allows you to execute files, set partitions and directories, load files, save all variables, and exit Xmath.
Edit	Allows you to clear the log area, message area, and command area, as well as send a command and insert a new line in the command area.
Options	Allows you to set the output display precision.
Windows	Quickly finds other Xmath windows and brings them to the foreground.
Help	Invokes the online Help and provides version information.

## 1.6.2 Log Area

The log area keeps a record of your interactions with Xmath. Both inputs and outputs are displayed in the log area. Certain actions in the user interface also cause Xmath to write to this area.

To control the number of lines written to the log area, type

```
set logarea N
```

where  $N$  is the number of lines;  $N$  is also limited by the buffer size, which is machine dependent. Using this command truncates the current contents to that number of lines.

To set the lines to the maximum, type

```
set logarea max
```

This limit is dependent upon hardware and the operating system resources available.

To turn writing to the log area off:

```
set logarea off
```

Current contents of the log area are discarded. While logging is turned off, the data is not being buffered, and it is lost. When you are running batch and

simulation jobs in SystemBuild, setting logging off speeds up their execution slightly.

To turn writing to the log area on:

```
set logarea on
```

All subsequent log data is displayed up to the limit; the limit is what you set previously or the default (maximum).

The command

```
show logarea
```

displays both the number of lines (or ALL) and the state of logging: On or Off.

To erase the log area, select Edit→Clear Log Area, or type

```
erase {logarea}
```

This action is not reversible, although you still have access to command recall to retrieve previous entries.

If a file is executed, the file contents are not written to the log area unless **set echo on** is specified (the default is off).

### 1.6.3 Command Area

The command area is the only part of the Xmath Commands window (or any of the major Xmath windows) that accepts text input, so you can focus anywhere on the window and type. (If you are not familiar with the term *focus*, see [A.2.2 Mouse Focus and the Pointer](#), p.350.)

Pressing Return or Enter causes Xmath to execute everything in the command area.

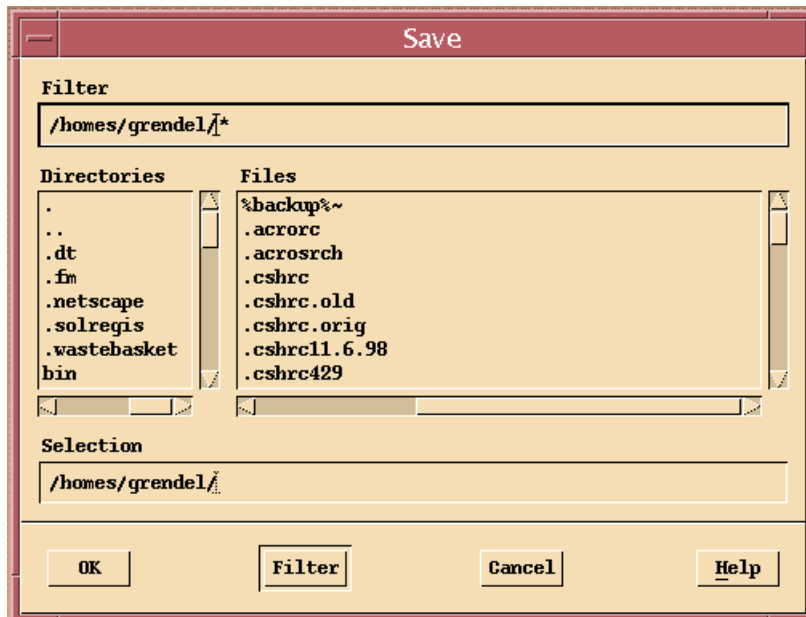
#### Specifying Directory Pathnames and Filenames

Within the command area, you often need to specify directory pathnames and filenames. To do so, you must use valid names. In general, Xmath does not recognize directory pathnames and filenames that contain spaces. Although such names are valid in Windows operating systems, Xmath does not recognize them from the command line; however, if you can select the directory and/or filename from a Browser or File Selection dialog, Xmath does accept them.

Scripts, saving and loading data, printing a file, and changing a directory or path can each be accomplished via a file selection dialog. On UNIX platforms, Xmath uses the Motif file-selection dialog for interactive directory and file specification.

Figure 1-3 shows a typical file selection dialog on UNIX. Most dialogs have the same fields, but some actions may not require all fields.

Figure 1-3 Save Dialog (UNIX version)



If you know the full pathname of the directory or file you want, type it in the Selection field at the *bottom*, and then press Return or click OK.



**NOTE:** All of the file interactions described above can also be accomplished from the command line, provided that the directory pathnames and filenames do not contain spaces, which are not generally recognized in the Xmath command area.

### Entering Multiple Lines of Information

Entering multiple lines of text works differently on UNIX and Windows systems. See the following sections for examples.

## On UNIX Systems

To enter multiple lines of text, press the Line Feed key or Shift-Return to start a new line. When you are finished typing, press Return to send all the lines to Xmath. Whenever the documentation refers to *linefeed*, you need to press the key(s) appropriate to your machine, as shown in [Table 1-7](#).

Table 1-7 **Linefeed Key**

Platform	Key or Key Sequence
Sun	Line Feed or Shift-Return
HP	Insert Line
IBM	Shift-Return
SGI	Shift-Return

You can achieve the same result by selecting Edit→Insert New Line from the menu bar. The accelerator for your machine appears to the right of the Insert New Line menu item.

## On Windows Systems

To send a set of multiple lines on Windows:

1. Enter the multiline mode by pressing Shift-Enter.  
You can press Shift-Enter before or after entering the first line.
2. Enter your lines of text, pressing Enter after each.
3. Leave the multiline mode by pressing Shift-Enter.
4. Send all lines to Xmath by pressing Enter.

The Edit menu provides the Send Command that you can use instead of the Enter key.

## Editing Text by Selecting, Copying, and Pasting

The command area is in insert mode. You can use mouse clicks or keyboard sequences to move the cursor within a line of text.



Your operating system's standard selection, copy, and paste methods are all valid.

The following selection sequences are defined:

- To select a character (forward or back), hold down the Shift key and press the right or left arrow key (command area only for UNIX).
- To select a word, point anywhere in the word and double-click.
- To select a line, point anywhere on the line and triple-click.
- To select all text in the window area, click four times (UNIX only).

You can paste text from any Xmath window or other ASCII source into the Xmath command area.

In UNIX, you can select a previous command from the log area, paste it into the command area, and re-execute it. The following copy and paste method is standard, although it may vary slightly with different window managers:

1. Point to the desired text and drag (holding down MB1) until everything you want appears in reverse video (is highlighted). Avoid highlighting extra characters.
2. Point to the destination and click MB2.

### **Key Bindings Used in Editing Text**

Key strokes help you perform editing functions for Xmath. Key bindings vary somewhat depending upon your type of operating system. You can change the key bindings for UNIX; for Windows, you cannot.

### **UNIX Default Bindings**

The UNIX default bindings are emacs-style, as shown in [Table 1-8](#). On UNIX systems keyboard types vary, so the default mappings for your particular keyboard might be slightly different. For example, arrows may map to editing keys or keypad arrows according to the keyboard.

Table 1-8 **UNIX Default Key Bindings**

<b>Keystrokes</b>	<b>Action</b>
Ctrl-a	Beginning of line
Ctrl-b, ←	Back one character

Table 1-8 **UNIX Default Key Bindings** (Continued)

<b>Keystrokes</b>	<b>Action</b>
Ctrl-d	Delete next character
Ctrl-e	End of line
Ctrl-f, →	Forward one character
Ctrl-j	New line
Ctrl-k	Kill to end of the line
Ctrl-l	Redraw display
Ctrl-n	Next line
Ctrl-o	Put remainder of line on a new line
Ctrl-p	Previous line
Ctrl-u	Delete to the beginning of the line
Ctrl-w	Wipe (delete) selected text
Ctrl-y	Yank back a single line of killed text (unkill)
Ctrl-↑	Move up through recorded inputs (command area recall is discussed on <a href="#">p.43</a> )
Ctrl-↓	Move down through recorded inputs
Backspace	Delete previous character
Delete	Delete previous character
Home	Move cursor to first character of text area
End	Move cursor to last character of text area
PgUp	Move up one page
PgDn	Move down one page
Ins	Insert a new line (linefeed)

### **Windows Bindings**

On Windows, the set of key bindings is more limited but still exists (see [Table 1-9](#)).

Table 1-9 **Windows Key Bindings**

Keystrokes	Action
Ctrl-a	Beginning of line
Ctrl-d	Delete next character
Ctrl-j	New line
Ctrl-k	Kill to end of the line
Ctrl-↑	Move up through recorded inputs (command area recall is discussed on <a href="#">p.43</a> )
Ctrl-↓	Move down through recorded inputs
Delete	Delete next character
Home	Move cursor to first character of text area
End	Move cursor to last character of text area

### ***Changing the Key Bindings (on UNIX systems)***

You may prefer UNIX-style or EDT-style bindings. These, along with many other defaults, are implemented through the file `$XMATH/etc/Xmath`.

To customize your key bindings, use a text editor to create a file called `Xmath` in your home directory. Into this file, copy the desired key binding set from `$XMATH/etc/Xmath`. Your `Xmath` file should contain only those changes that differ from the defaults. Close and save your file.

The new key bindings become effective the next time you invoke `Xmath`. (For more information, see [A.3 Changing Resource Parameters](#).)

### ***Recalling Previous Commands***

`Xmath` has a command area recall feature based on keystrokes, as shown in [Table 1-10](#).

Table 1-10 **Command Area Recall Keystrokes**

<b>Keystrokes</b>	<b>Action</b>
Ctrl-↑	Moving backwards, print recorded inputs in the command area.
Ctrl-↓	Moving forward, print recorded inputs in the command area.
@@	Execute the last command.
@@:p	Print the last input in the command area.
@str	Execute the last input starting with <b>str</b> .
@str:p	Print the last input starting with <b>str</b> .
@n	Execute the <i>n</i> th input.
@:l	List all inputs in the log area.
@str:l	List all inputs starting with <b>str</b> in the log area.
@	List the last 10 inputs. If @ is issued again (without an intervening Xmath command) 10 inputs back from that point will be listed.

- Only syntactically correct inputs are recorded.
- @ commands are not recorded as inputs.
- Multiline inputs are recorded and recalled as one line.
- One hundred inputs are recorded; the oldest are automatically discarded to make room for new inputs.
- An @ command can only be entered in the Xmath Commands window command area on a line by itself. It cannot be issued from a MathScript batch file.

### 1.6.4 Message Area

The message area displays Xmath error messages and warnings. If an error occurred when you were typing in the command area, Xmath highlights the possible source of the error and displays a message in the message area (UNIX) or the log area (Windows). The input is not accepted until you fix the error.

## 1.7 Help Window

Xmath function reference information is available in the  $\text{MATRIX}_X$  online Help. The online Help includes all Xmath functions. Each topic explains a function's inputs, outputs, and keywords in detail. See Chapter 2 of the *MATRIX<sub>X</sub> Getting Started Guide* for complete instructions on using the Help feature.

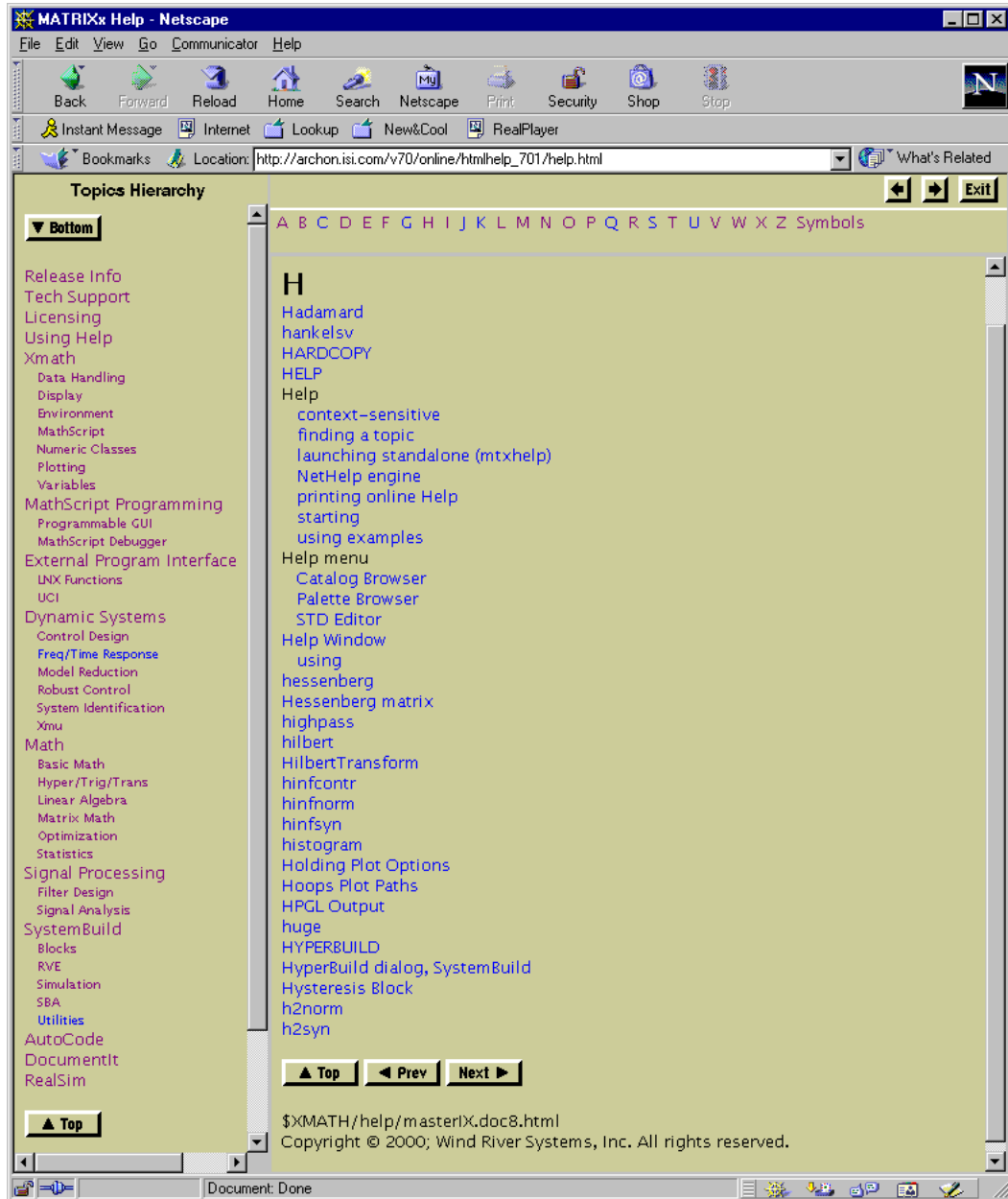
You can invoke the  $\text{MATRIX}_X$  online Help as follows:

- Select Help→Topic from the Xmath Commands window or type **help** in the command area of the Xmath Commands window; a listing of available topics appears in the left pane (see [Figure 1-4](#)). Scroll down to see additional entries.
- Once in the Xmath Help window you can use the Topics Hierarchy (table of contents) in the left pane to locate topics.

For example, to view a linear algebra function topic (for example, the function **hessenberg( )**), click the *Math, Linear Algebra* topic in the left pane, and then click *hessenberg* in the right pane.

- You can also use the Master Index (see [Figure 1-4](#) in the right pane) to locate a topic or function alphabetically. Using the alphabet at the top of the right pane, you can link directly to the topics for any given letter.

Figure 1-4 MATRIX<sub>x</sub> Help Window Topics Hierarchy and Master Index



# 2

## *JumpStart: A Tutorial*

This tutorial introduces basic Xmath features. It highlights some of the ways Xmath is different from other tools. After getting you started, this chapter provides the following major topics; the times shown are estimates of how long it takes to complete each section.

Topic	Page	Time to Complete
<i>Basic Data-Handling</i>	48	15 minutes
<i>Functions and Commands</i>	57	10 minutes
<i>Graphics</i>	58	30 minutes
<i>Objects</i>	70	60 minutes
<i>MathScript</i>	85	15 minutes

To use the JumpStart you must have a properly installed version of Xmath. You should also be familiar with the following:

- Your operating system
- A text editor
- On UNIX platforms, your window manager

If you are new to the workstation environment described in this book, see [Appendix A, X Windows and Motif](#). It will be helpful to new UNIX users because many UNIX-based window managers share common functionality. We assume that workstation users have X Windows and a window manager running before starting this tutorial. The Jumpstart is very basic and you will

be able to complete it even if you are unfamiliar with the workstation environment.

If you find yourself having difficulties with the most basic elements, such as not understanding how to use the Xmath Commands window or how to get online Help, refer back to [Chapter 1](#).

This tutorial contains many cross-references to other parts of the document. It is not necessary to consult the cross references to complete this tutorial. After completing the tutorial, you may want to look into some of the advanced features in [Chapters 6 through 9](#).

## 2.1 Starting Xmath for the Tutorial

In this section, we want you to create a directory called **jumpstart**, make that directory your working directory, and start Xmath. From a Terminal window (UNIX) or the Command Prompt window (Windows), enter the following commands:

```
% mkdir jumpstart  
% cd jumpstart
```

Then start Xmath using one of the methods provided in [1.3.1](#), p.7.

You may stop or interrupt the tutorial at any point. Remember to save your work before you quit and to reload it upon startup again (see [1.3.4](#), p.11 for details.)

## 2.2 Basic Data-Handling

This portion of the tutorial discusses creating and organizing variables, as well as saving, deleting, and retrieving them.



## 2.2.1 Creating Variables

A variable is named information. To create a variable, you must type into the Xmath command area. You can assign a name to data:

```
a=3.14
```

```
a (a scalar) = 3.14
```

and assign the results of expressions or the output of an Xmath function:

```
b=a+expm([1,2;3,4])
```

```
b (a square matrix) =
```

```
  55.109    77.8766
 115.245    167.214
```

Pressing Return or Enter executes everything in the command area. By default, your input is displayed in the log area, followed by the output. To suppress output display, terminate inputs with a semicolon (see [p.93](#) for a way to change display behavior).

```
b;
```

If you input more than one statement on a line, a semicolon or question mark (which forces output) must be used as a separator. Type:

```
c=b^a; d=b/a? c=d-a;
```

```
d (a square matrix)=
```

```
 17.5506  24.8015
 36.7022  53.2528
```

The only output displayed is the value of **d**, but **c** exists.

When entering multiple lines of text in the command area, use the Line Feed key or Shift-Return to start a new line, and press Return when you are finished. If your keyboard doesn't have a Line Feed key select Edit→Insert New Line from the Xmath Commands window, or use the key combination appropriate to your platform (see [p.19](#)).

In the following example, press Line Feed after inputting the numbers 3 and 6, and press Return after the right square bracket:

```
e=[1,2,3
   4,5,6
   7,8,9]
```

```
e (a square matrix) =
```

```
1  2  3  
4  5  6  
7  8  9
```

If you don't assign a variable name to a valid statement, Xmath assigns the value to the temporary variable **ans**. The following expression uses the permanent variable **jay** to create a matrix of complex numbers and assign the matrix to **ans**:

```
e*jay;  
ans?
```

```
ans (a square matrix) =
```

```
  j   2 j   3 j  
4 j   5 j   6 j  
7 j   8 j   9 j
```

**ans** will be changed the next time a statement output is not assigned to a variable.

To comment an existing variable use the **comment** command:

```
comment b "combined an expression and a function"
```

You must enclose the comment string, like all other strings in Xmath, in double quotes:

To retrieve the comment, use the **commentof()** function:

```
commentof(b)
```

```
ans (a string) =   combined an expression and a function
```

Xmath also displays the comment when you view the variable in the Xmath Variable Manager window, which is discussed in [2.2.3](#), p.53.

If you make an error, Xmath attempts to highlight the incorrect input. For example, type:

```
max(E)
```

What you typed remains in the command area with the E in reverse video. The message area displays **E undefined in this scope**.

Go to the command area and replace the capital E with a lowercase e:

```
max(e)
```

```
ans (a scalar) =   9
```

The `max()` function now finds the largest value in the variable `e`. For detailed information on entering and editing text, see [Editing Text by Selecting, Copying, and Pasting](#) on p.20.

### Using Command Recall

To print previous inputs to the command area, hold down the Control key and press the up arrow (Ctrl-↑). For more on command area recall, see [p.23](#).

### Sending Multiple Lines of Data at Once

On UNIX, you press Shift-Return after each line until you are ready to send the entire set of lines to Xmath; then you press Return.

On Windows, pressing Shift-Enter turns on multiline mode. In this mode, pressing the Enter key adds a new line rather than sending the command line to Xmath. Pressing Shift-Enter again turns off this mode. Pressing Enter a final time sends the multiple lines to Xmath for execution.

For example:

	UNIX	Windows
<code>for i=1:10</code>	Shift-Return	Shift-Enter Enter
<code>  i?</code>	Shift-Return	Enter
<code>endfor</code>	Return	Shift-Enter Enter

sends the multiline for-loop to Xmath at one time.

## 2.2.2 Variables and Partitions

Xmath variable names are case-sensitive (for example, `MyVar`, `myvar`, and `MYVAR` are different variables).

A *partition* is a named non-hierarchical directory that contains variables. Partition names are also case-sensitive.

Xmath always starts in the default partition **main**. You can verify this by typing **show partition** in the command area. The full name of a variable includes its partition, so the variable **a**, found in partition **main**, is named **main.a**. However, you don't need to supply a prefix when handling variables in the current partition.

Use the command **new partition** to create partitions. Other commands used for partition handling are **set**, **show**, and **delete**.

1. Create new partitions:

```
new partition data1
new partition data2
```

2. Using variables in the current partition (the default partition **main**), create new variables for the partition **data1**:

```
data1.a=a\b;
data1.b=lyapunov(b,c);
```

3. Go to the new partition **data1** and display a list of the variables in that partition to the log area:

```
set partition data1
who # List variables in the current partition

data1:
      a -- 2x2
      b -- 2x2
```

4. Attach a comment to a partition in the same way you comment variables, except that you must put a period after the partition name to distinguish it from a variable name:

```
comment data1. "vault"
commentof(data1.)

ans (a string) = vault
```

5. Use the same variable name in other partitions:

```
data2.a=random(4,4);
comment data2.a "a random matrix"
who data2.* # List variables in the named partition

data2:
      a -- 4x4
```

6. Look at all the partitions and all existing variables:

```
show partitions      # Shows all partitions
who *.*            # Show all variables in all partitions
```

### 7. Delete a partition (**data1**).

To delete a partition, you must first empty it:

```
delete data1.*      # Delete variables in data1.
```

To delete a partition you are in, change to another partition first:

```
set partition main
delete data1.       # Delete the partition data1
```

## 2.2.3 Viewing Data

The Xmath Variable Manager window lists all variables in the current partition. While it is open, Xmath immediately updates it whenever changes occur in the viewed partition.

1. To invoke the Xmath Variable Manager window, select Windows→Variables from the Xmath Commands window.

You should be viewing the current partition (**main**).

2. Click the Partition button in the Xmath Variable Manager window. In the dialog that appears, select **main**, and click OK.

This lists the variables in **main**. Note that you are only *viewing* the partition; *you have not changed your working partition*. (Only the **set partition** command issued from the command area will change the partition. Remember, you can type **show partition** to see the current partition.)

3. Try the selections on the Variable Manager window View menu to change the organization of the variables. Try sort by Name, sort by Size, and sort by Type.
4. To close the Variable Manager window, select File→Close Window.

For additional information on the Variable Manager window, see [3.2.5](#), p.79.

## 2.2.4 Saving Data

The commands and functions in [Table 2-1](#) save data to files.

Table 2-1 **Save Commands and Functions**

---

<b>SAVE</b>	Save variables in Xmath or MATRIX <sub>x</sub> format to a binary or ASCII file. This is the standard way of saving data.
<b>PRINT</b>	Print the values of a list of variables to an ASCII file.
<b>fprintf()</b>	Convert numeric values to a string representation, and then write the string(s) to an ASCII file.

---

You can perform save operations from the command area and from the File menu of most windows.

### Save Command

The easiest save method is to type **SAVE** in the command area. When you do, Xmath saves *all* variables to a file named **save.xmd** in the current working directory. By default, **SAVE** produces a binary file with the variables saved in Xmath format.

You can specify a list of variables, a filename, or a format. For example,

```
save main.* file="main" {ascii}
```

saves all variables in the partition **main** to an ASCII file named **main.xmd** in the current partition. Note that **SAVE** adds the **.xmd** extension for you.

To save all variables to a binary data file via the File menu in either the Xmath Commands window or the Xmath Variable Manager window:

1. From the menu bar, choose File→Save All (UNIX) or File→Save (Windows).

The Save dialog comes into view.

2. Add the filename **data1.xmd** to the path in the Selection field at the bottom of the dialog (UNIX), or select a directory and then specify a filename in the File name field of the Windows' dialog.

For a complete explanation of this dialog, see [Specifying Directory Pathnames and Filenames](#) on p.18.

3. Click OK or Save.

If you look at the log area, you will see that the text equivalent of your save action is echoed there (UNIX only). The current message will be similar to:

```
save file="/YourPath/data1.xmd"
```

## Print Command

The **print** command writes a variable in a text format you can read.

To print a specific variable to a text file:

```
set seed =0;
x=(rand(2,2))*sin([5,1;4,2]);
print x file="x.dat"
```

The function **oscmd()** lets you use an operating system command to display the contents of the file you created to the Xmath Commands window log area:

```
oscmd("cat x.dat")      # UNIX
oscmd("type x.dat")    # Windows

main.x =
-0.77482  0.865292
-0.250204 0.300552

ans (a scalar) = 0
```

## 2.2.5 Loading Data

### Load Command

If you type **load** (with no file specified) in the command area, Xmath looks for the default file **save.xmd** in the working directory and loads it if it exists.

To test this, go to the command area and input the sequence below; these instructions assume you are in partition **main**.

```
a=1; b=2; c=3; d=4;      # Create variables a, b, c, and d

save                    # Save all variables to save.xmd

who *.*                # Verify that the variables are in main

delete *.*             # Delete variables in main

who *.*                # Verify that the variables have been deleted
```

You can then retrieve selected variables or all saved data:

```
load c d "save"        # Load variables c and d from save.xmd

-or-
```

```
load                                # Load all variables in save.xmd
who *.*                             # Verify that the variables have been loaded
```

Xmath supplies the default filename extension **xmd** when you don't supply one. Another way to load saved data is to go to the commands or Variable Manager window and select File→Load from the menu bar.

### Read Command

The **READ** command copies the contents of a file into an Xmath matrix. This function is particularly useful for loading externally generated data into Xmath. The data can be character, integer, or floating-point types, as well as ASCII. Consult the MATRIX<sub>x</sub> online Help *READ* topic. Note that the arguments are a filename, the rows and columns of the data, the type (or format), and the number of bytes in the file you want to skip before reading.

Read in the file you made with the **print** command (see [Print Command](#) on p.55).

- Specify the input filename (**x.dat**), give the row and column dimensions of the data, and specify the input file format (**ascii**).
- Specify an offset of 1; this instructs Xmath to skip the first line (**main.x =**).

We do not have to worry about the last line in the file, (**ans (a scalar) = 0**), because **read** stops after the two rows and columns you specified have been read.

```
xx=read("x.dat",2,2,"ascii",1)

xx (a square matrix) =

-0.77482    0.865292
-0.250204   0.300552
```

### 2.2.6 Cleanup

This concludes the section on basic data-handling. You can delete the variables and partitions you created, as you do not need them later. Do not, however, delete the partition **main**; delete only its contents.



## 2.3 Functions and Commands

If you have been working through the tutorial, you have already used several common commands and functions. In addition to discussing functions and commands, this section includes references to more detailed passages.

### 2.3.1 Function Syntax

Functions operate on a list of input values and return output values. Input arguments are passed by value (a local copy is nested inside the function scope).

Functions are called in the following form:

```
[out1,out2,...,outn] = funName(in1,in2,...inm,{options,keywords})
```

For examples of this syntax, see the `MATRIXX` online Help *Functions* topic.

- Input and output arguments are separated by commas.
- Keywords are enclosed in braces and separated by commas.
- When a string is required, it must be enclosed in double quotes; for example, `line_color="blue"`.
- If a function has multiple outputs, by default only the first output is returned. You must use the brackets if you wish to acquire more than one output.

The following example shows two possible syntaxes for `residue`. Input the following data to see the default output behavior:

```
sys=(makepoly([2:4:6])/makepoly([3,5]));  
Rp=Residue(sys,[5,10,inf],[tol=.5])
```

To see both outputs, use square brackets and assign the outputs to variables:

```
[Rp,C]=Residue(sys,[5,10,inf],[tol=.5])
```

For additional information, see [3.5 Using Predefined Functions and Commands](#), p. 85. Xmath function syntax is detailed for each function in the `MATRIXX` online Help. For a detailed description of how to use MathScript to define your own functions, see [Chapter 6](#).

### 2.3.2 Command Syntax

Like functions, commands operate on inputs. However, command inputs are passed by *reference* and can be changed within the command.

In the MathScript language, command syntax is as follows:

```
command arg1, arg2, ...argN, {keywords}
```

For examples of this syntax, see the MATRIX<sub>X</sub> online Help *Commands* topic.

If you have been working through the tutorial, you might realize that intrinsic commands have a special syntax. Syntaxes we have used are:

```
new partition part_name  
set partition part_name  
delete part_name  
save "filename" var_1 var_2 var_n  
load "filename" var_1 var_2 var_n
```

The most obvious difference is that these commands require spaces rather than commas as separators. The **whatis** command reveals a fundamental difference between these commands and other MathScript commands (MSCs):

```
whatis save  
  
save: intrinsic command
```

Xmath includes many *intrinsic* commands and functions. These commands and functions are part of the Xmath executable.

See the MATRIX<sub>X</sub> online Help for descriptions of Xmath commands. For a detailed description of how to use MathScript to define your own commands, see [Chapter 6](#).

## 2.4 Graphics

The Xmath **plot()** function provides two and three-dimensional graphics that you can manipulate interactively while they are displayed in the Xmath Graphics window. This section introduces **plot()** and several types of plots it can create.

Two additional general purpose plotting functions, `uiPlot()` and `plot2d()`, complement the capabilities of `plot()`. A brief description of these functions can be found in [4.1 . Xmath Plotting Functions and Commands](#)

### 2.4.1 Plot()

The `plot()` function creates a graph object that Xmath displays in the Xmath Graphics window. The most complete syntax for `plot()` is:

```
graphObj = plot(x,y,z,colorindex,{keywords})
```

2D graphs are produced with `y`, or `x, y` as arguments, while 3D graphs require `x, y`, and `z`. For other `plot()` syntaxes see [4.2 Using the plot\(\) Function](#), p.108.

`plot()` behaves like other Xmath functions in the following ways (functions are discussed in [2.3 Functions and Commands](#), p.57.):

- If no output variable name is assigned, Xmath assigns the output (graph object) to the temporary variable `ans`.
- Xmath displays a graph object in the Xmath Graphics window when it is created unless you use a semicolon as a terminator. If you create a graph object within a MathScript, only a `?` terminator causes it to display.
- You can display a graph object with the `?` terminator anytime after creation.
- You can save and load a graph object.

#### Keywords

Keywords define a graph's labeling, layout, and appearance. This tutorial introduces basic keyword use. For a complete keyword listing, see [Table 4-5](#), p.113, or the `MATRIXx` online Help `plot` topic. You can create or change many of the features for which keywords are used interactively via the Xmath Graphics window menus or the Xmath Palette.

#### Graph Objects

`plot()` is the only function that outputs a graph object. Xmath creates a graph object whenever it displays the output of the `plot()` function in the Xmath Graphics window. If you specify an output variable name, Xmath writes the contents of the Xmath Graphics window to the variable; otherwise, Xmath writes the contents to

the default variable **ans**. If you suppress plot with a semicolon, Xmath writes nothing to the Xmath Graphics window. (Other functions may display plots in the Xmath Graphics window, for example, windowing functions such as **firwind()**, but their actual function output is numeric. Only **plot()** allows you to name the contents of the Xmath Graphics window.)

You can copy, save, display, and reload a graph object like any other variable. Additionally, it can be altered or used in a new graph if you use the keywords **keep** or **copy**. We explore the implications of the graph object later in the tutorial.

## 2.4.2 Working in the Xmath Graphics Window

When you use the plot function without suppressing its output, Xmath opens the Xmath Graphics window. The following mouse actions are defined for this window:

- To select an object, click it.  
An object can be a text string, label, grid, data, and so forth.
- Double click an object to select the object and bring up the Xmath Palette.  
The palette title area (center top) gives information on the object you've selected. For example, the title Xmath Palette (tics:axis line) indicates that you've selected an axis line.  
Different menu items and palette locations on the Xmath Palette are enabled based on your selection. For example, if a label is selected, the Font and Point menus are enabled, and the text color can be changed via the palette.
- If you have difficulty selecting an object (for example, you attempt to select a tic mark, but you keep getting the axis), then hold down the Shift key while clicking.  
Xmath cycles through selecting the objects closest to the cursor. A glance at the palette title area reveals the selected object.
- Click and drag to move objects.  
Objects that you can move independently are the legend, date, time, free text, and graphics that you create with the graph tools in the Xmath Graphics window icon bar. You cannot move a graph and its associated plot data, grids, labels, axis information, and so forth interactively, but you can move the entire graph with the **plot()** keyword **position** (see [p.137](#)).

### 2.4.3 Using Plot and Graph Objects

You can plot objects in two- or three-dimensional plots.

#### Using 2D Plotting Capabilities

Before continuing, generate a few waveforms:

```
set seed = 0                                # Set random seed
a = sin(logspace(1,10,15));
b1 = kronecker(a,a);
b2 = b1 + 0.2*random(1,225);
```

Here **graph\_b1** is a graphical object with **b1** plotted versus a time sequence:

```
t=0:0.01:2.24;
graph_b1=plot(t, b1,{title="xy plot",x_lab="time(sec)"})?
```

If you can't see the graph, select Windows→Graphics to bring the Xmath Graphics window to the front.

Plot **b2** with specific labels and titles:

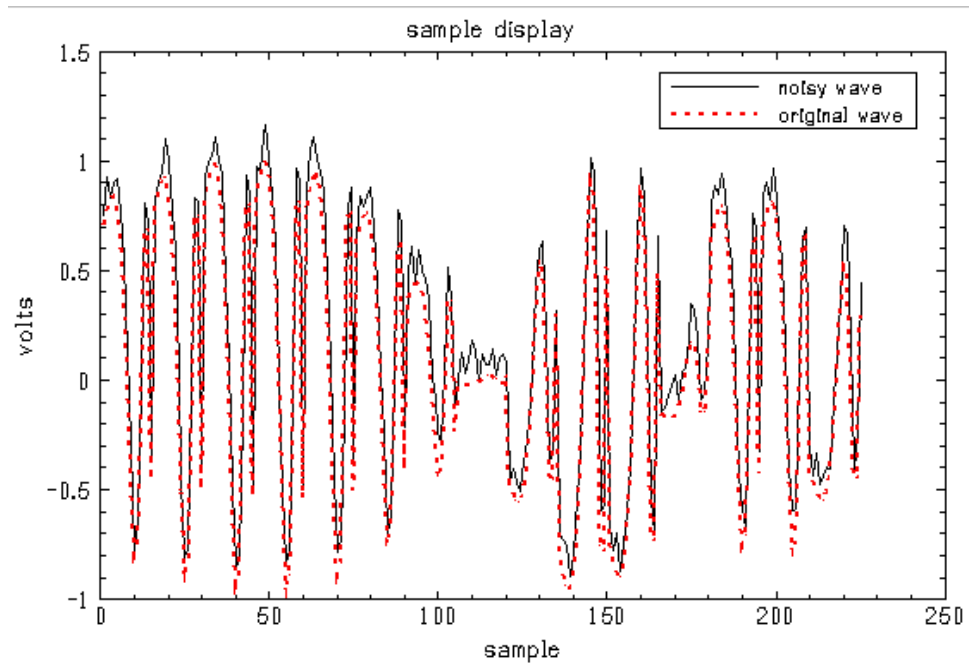
```
graph_b2=plot(b2, {y_lab="volts",x_lab="sample",
title = "sample display",legend = "noisy wave"})?
```

You can plot the original noise-free waveform **b1** over the existing plot by copying the graph object **graph\_b1** into the current graph. In the command area type:

```
both_b=plot(b1,{copy=graph_b2,line_style=3,
line_width=2,legend = "original wave",!grid})?
```

Figure 2-1 shows the result. **b1** is plotted as a thicker dotted line added to **graph\_b2**, a new entry is added to the legend box, the grid is suppressed by the ! negator, and the image is given the name **both\_b**.

Figure 2-1 Overlaid Graph Objects



To see the first plot, type:

```
graph_b1?
```

You do not need to execute the previous plot call to see the graph. **graph\_b1** is unchanged because the keyword **copy** was used and the current contents of the window were given a new name (**both\_b**). If you are adding to a plot and it is not important to retrieve your previous efforts, use **keep** instead of **copy**. **keep** is much faster than **copy**.

When you make interactive changes to a graph object displayed in the Xmath Graphics window, the changes immediately become part of the current graph object. To preserve **graph\_b1** as it is, rename the graph before making changes in one of following ways:

- From the Xmath Graphics window menu bar select File→Bind to variable and save the contents of the Xmath Graphics window to the name **g1**.
- From the Xmath Commands window command line, type:

```
g1=plot()
```

Calling `plot()` with no arguments binds the contents of the Xmath Graphics window to the output variable name.

To illustrate that changes immediately become a part of the current graph object, go to the Options menu and turn on the timestamp and datestamp; then move them to new locations. Double-click a text string, and then change the font and point size using the Xmath Palette. Double-click a curve either in the data or in the legend, and then go to the Xmath Palette and change the marker and line styles.

Display the object `graph_b1` and then the object `g1`:

```
graph_b1?
```

```
g1?
```

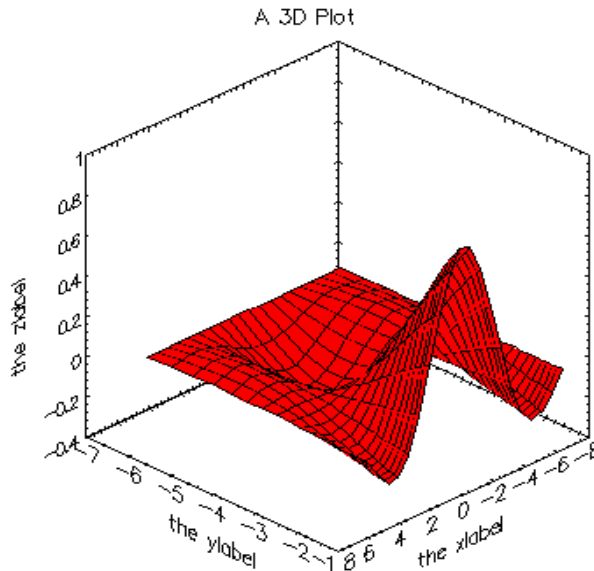
### Using 3D Plotting Capabilities

To demonstrate some of the 3D plotting capabilities, create `x`, `y`, and `z`:

```
x= [-2*pi:.65:2*pi]';  
y= logspace(1,2*pi,20);  
z= sin(x)./x*(sin(y)./y);  
plot(x,-y,z,{title="A 3D Plot",xlab="the xlabel",  
ylab="the ylabel",zlab="the zlabel",!grid})?
```

Figure 2-2 shows these results.

Figure 2-2 **3D Plot with Labels and Title**



You can rotate 3D plots with the rotation tools on the far right of the menu bar in the Xmath Graphics window. The first tool allows you to rotate in all directions (unconstrained); the remaining tools rotate about the three principal axes. Select a rotation tool in the icon bar, and then move to the plotting area. When the tool is active, just the grids are shown; click and drag the cursor until the grid is in the position you want to see, and then release the mouse. Xmath redraws your graph in the new position.

To return to the initial plot position:

Select View→Reset.

To turn off the rotation tool:

Click the arrow (selection tool) on the far left of the menu bar.

#### **2.4.4 Using Different Plot Types**

In this section, we illustrate the use of different kinds of plots: strip, polar, bar, and contour.



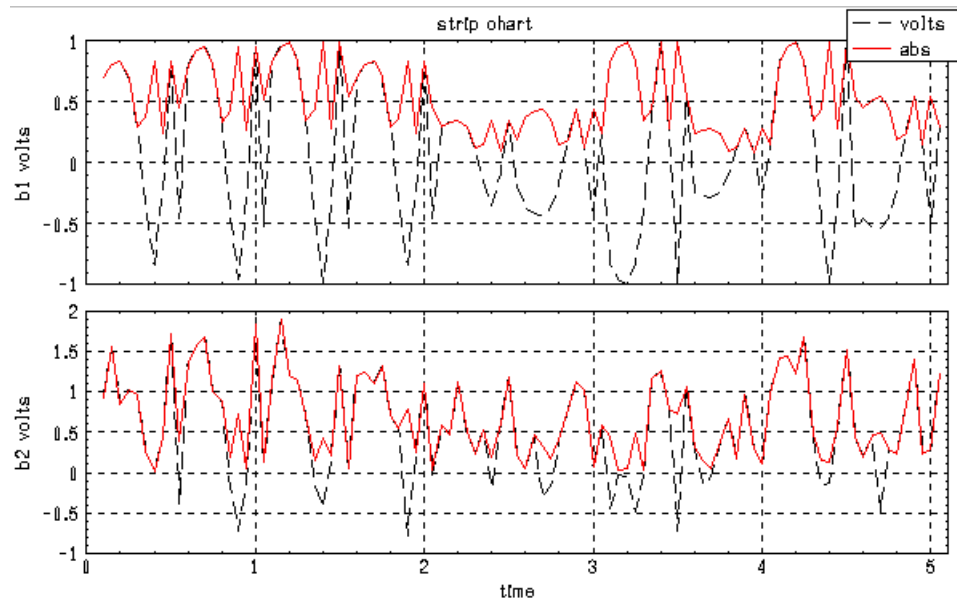
## Strip Plots

The **strip** keyword aligns two or more waveforms in stacked graphs sharing a common x-axis. Optionally, you can specify the number of curves you want in each graph. (Strip plots, like all other multiple graph plots, cannot be rotated or zoomed.) The example below plots four variables; **strip=2** specifies that each graph should contain two curves (see Figure 2-3). We specify an optional **line\_style** vector with **legend** to distinguish the original values of **b** from the absolute values.

```
set seed = 0          # Set random seed
a = sin(logspace(1,10,10));
b1 = kronecker(a,a);
b2 = b1 + random(1,100);
t=.1:0.05:5.05;
plot (t, [b1;b2;abs(b1);abs(b2)]', {strip=2,
    title ="strip chart",line_style=[2,1],
    legend=["volts","abs"],xmax=5.1,
    ylab=["b1 volts","b2 volts"],xlab="time"})?
```

Xmath creates a single legend, and the two plots share the title and **xlab**. Strip chart data is linked; to illustrate this, select a curve in one of the plots; the corresponding curve in the other plot is also highlighted.

Figure 2-3 Strip Plot with Two Curves in Each Strip

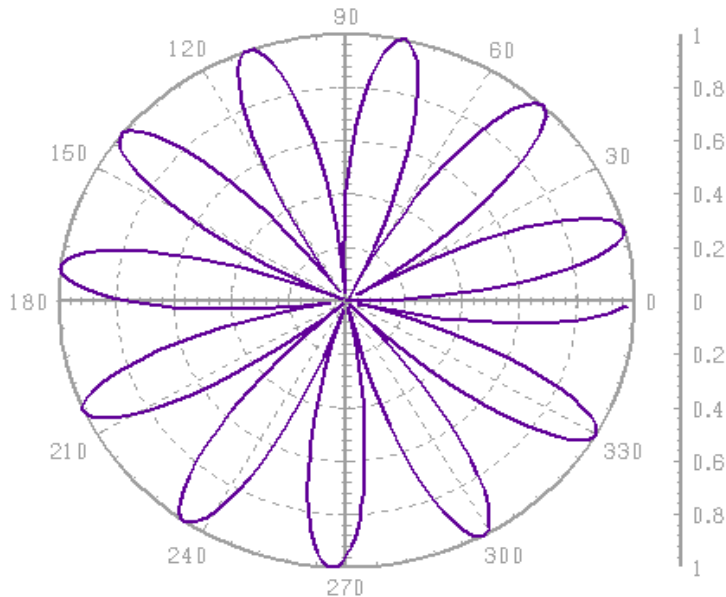


## Polar Plots

Xmath can display data in polar plots (see [Figure 2-4](#)). For example,

```
r = abs(sin(0:.1:35.9));  
theta = 0:1:359;  
plot(theta,r,{polar, fg_color="gray2",  
line_color="royal purple", line_width=2})?
```

Figure 2-4 **Polar Plot**



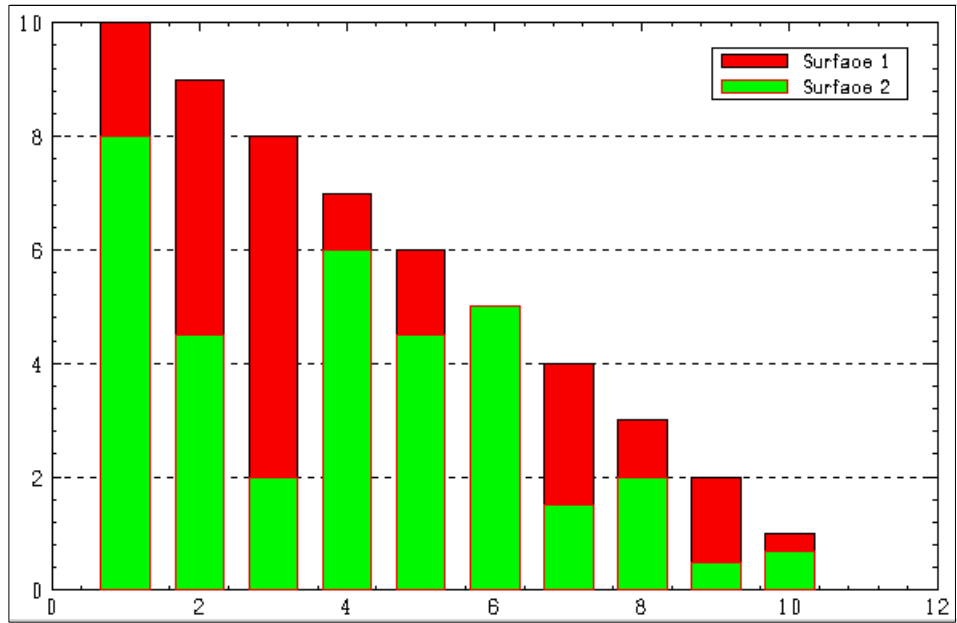
## Bar Plots

Xmath also has bar graph capabilities.

Bar plots can be overlaid using the **keep** keyword. If a variable name is not specified, **keep** adds what you specify to the current contents of the Xmath Graphics window. The results of the example below appear in [Figure 2-5](#).

```
plot(10:-1:1,{bar})?  
plot([8,4.5,2,6,4.5,5,1.5,2,.5,.7],  
{keep,bar,!xgrid,legend})?
```

Figure 2-5 Overlaid Bar Plots



### Contour Plots

```
x = [-2*pi:.6:7]'; y=x;
z = 1.2 + sin(x)./x*(sin(y))';
```

The first graph is a 3D surface plot, with grids suppressed:

```
plot(x,y,z,{'!grid'})?
```

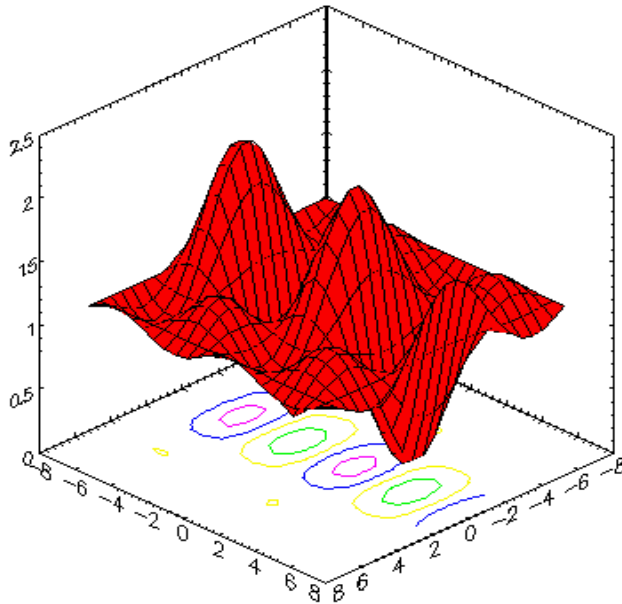
With the **keep** keyword, you can overlay a 2D contour plot of the same surface (see [Figure 2-6](#)):

```
plot(x,y,z, {'keep,contour2d,!face,contour_interval = 0.5'})?
```

Alternatively, you can display a 3D contour plot:

```
plot(x,y,z, {'contour3d'})?
```

Figure 2-6 3D Plot with 2D Contour



### 2.4.5 Displaying Multiple Plots at Once

The **rows** and **columns** keywords allow you to display up to 25 different 2D and 3D graphs at once. The values you assign to **rows** and **columns** determine how the screen is subdivided. Plots are then positioned on the screen with a combination of **row** and **column** numbers or a **graph\_number**. The **rows** and **columns** keywords are *initiators*. This means they remain in effect until a plot call that does not contain a **row** or **column** keyword is issued; at this point the default values **rows=1**, **columns=1** are reset.

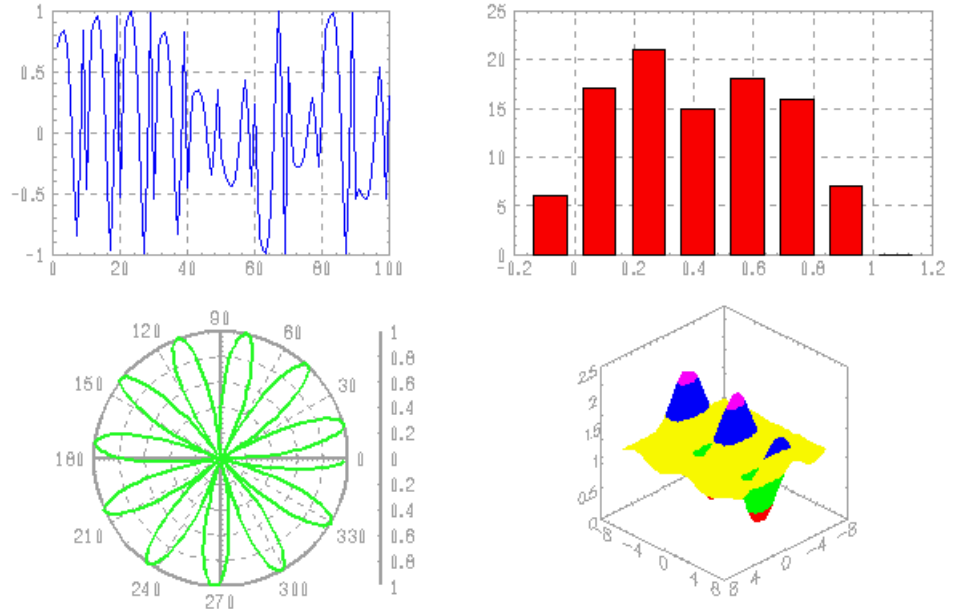
The following example places four plots on the screen in two rows and two columns. Note that you don't need to specify **row=1** or **column=1**; these are default values. The result is shown in [Figure 2-7](#).

```
set seed 0
h=histogram(rand(1:100),{nbins=7,noplot});
plot (b1,{rows=2,columns=2, line_color="blue"})?

plot (theta,r,{polar,row=2, fg_color="gray2",
line_color="royal purple", line_width=2})?
```

```
plot (h, {bar, column=2, !xlabel})?
plot (x,y,z, {contour3d, xinc=4, yinc=4, !grid, graph_number=4})?
```

Figure 2-7 Different Plot Types Positioned with row and column Keywords



### 2.4.6 Animating Plots

Given a series of plots, the **animate** keyword draws each plot as fast as possible so the progression looks like movement. For the following example, create a vector:

```
an1=sin(logspace(1,10,25));
an2=an1(25:-1:1);
an3=kroncker(an1,an2);
```

We will be looking at **an3** using 100 points at a time. First, we plot the entire vector using **animate** and a fixed axis (**axisfix**). By default, axes are adjusted to the current plot range, so, if **animate** is enabled, axes may change while plotting. In this call, **axisfix** holds the axes of the current plot (until they are changed), ensures that the plot background remains the same, and (since the whole vector is plotted) that the plot area is not too small for the plot.

```
plot(an3, {animate, axisfix, xmax=100})?
```

The **animate** keyword stays active until it is disabled explicitly.

Use a loop to plot portions of the data while **animate** is enabled:

```
for i=1:7:524
    plot(an3(1,i:i+100))
endfor
```

To turn off **animate** type:

```
plot({!animate})
```

Alternatively, you can use **plot(reset)** to reset *all* plot defaults.

If you are curious about **axisfix**, repeat the above example without it, and watch the axes.

## 2.4.7 Finishing the Graphics Tutorial

The above examples show only a sampling of the options available for **plot()**.

For more information on **plot()** graphics, first ensure that **animate** is switched off. Then run the graphics demo:

```
plot({!animate})
execute file = "$XMATH/demos/graphics"
```

Also see [4.4 Interactive Xmath Graphics Window](#), p.151, and the MATRIX<sub>X</sub> online Help *Xmath, Plotting* topic.

This ends the graphics portion of the tutorial. Before moving on, you should delete the variables you created in this section:

```
delete *.*      # Delete all variables in all partitions
```

## 2.5 Objects

Unlike most numerical tools, which only deal with matrices, Xmath employs object-oriented programming principles. See [Figure 5-1](#), p.168 for a full description of the Xmath object hierarchy structure. For example, the Toeplitz matrix class is a special kind of square matrix class. It inherits all the properties of

the square matrix class but automatically takes advantage of specific operations which can be performed more efficiently for Toeplitz matrices.

Benefits from Xmath's object-oriented structure include:

- Fewer variables to manage. A single variable can represent several very complex pieces of data. Therefore, you don't need as many variable names, which simplifies variable management.
- Fewer functions. For example, a single function handles continuous and discrete cases.
- Faster calculations. Many objects take advantage of optimized algorithms. This is especially true of all the specialized matrix objects. Xmath recognizes special data properties and automatically uses an optimal method if available.
- More intuitive syntax and ability to overload operators. Overloading means that a single operator can have different meanings when it interacts with different objects.
- More compact user code. Because objects have clearly defined properties, it is simpler for users to check and handle data in their programs.

This section briefly discusses the major Xmath objects. There are examples of how to create each one and, in some cases, examples of special techniques with operators or indexing. The examples create unique data for each object. Therefore, you may quit the tutorial between any of the object discussions and restart when convenient.

## 2.5.1 Strings

A string is a set of characters enclosed in double quotes. To display double quotes within a string you must provide two sets of quotes ("""). You can convert numbers to strings with the `string()` function, while the `char()` function gives the ASCII character for a given integer between 0 and 255.

```
a = "The total score is ";  
b = 301;  
c = a + string(b)?  
  
c (a string) = The total score is 301
```

You can create a matrix of strings using the familiar matrix-constructor syntax.

```
a = ["one", "two"; "three", "four"]
```

```
a (a square matrix of strings) =
```

```
  one      two
  three    four
```

When entering strings in the Xmath Commands window command area, remember that a single string must be complete on a line. If for some reason you must break the string, create separate strings and append them with the + operator:

```
text="Xmath strings cannot be continued " +...
      "across lines, but separate strings can " +...
      "be appended with the + operator."
```

```
text (a string) = Xmath strings cannot be ...
```

## 2.5.2 Matrices and Vectors

This section demonstrates how to create and use matrix and vector objects. It also shows how Xmath's object-oriented structure improves the computational speed of matrix operations.

### Creating Matrices and Vectors

You must enclose matrix specifications in square brackets; you separate elements in separate rows by commas and row elements, by semicolons or line feeds:

```
[1,2; 3,4]    # A semicolon or a linefeed
[1,2         # can separate rows
 3,4]
```

```
ans (a square matrix) =
  1  2
  3  4
```

```
ans (a square matrix) =
  1  2
  3  4
```

A vector is a single row or single column matrix. An apostrophe (') transposes a vector or a matrix.

```
i=[1,2,3]
(a row vector) =  1  2  3
i'
```



```
ans (a column vector) =
```

```
 1
 2
 3
```

Regular vectors are row vectors specified as three values in the form **start:step:end**.

```
time=0:0.01:10
```

```
time (a regularly spaced vector) = 0 : 0.01 : 10
```

The **logspace()** function creates logspaced vectors with points evenly spaced on a log scale. Like regular vectors, logspaced vectors are stored as three values.

```
log1=logspace(1,2,5)
```

```
log1 (a log-spaced vector) = 1 : 2 (5 points)
```

Transposing a vector or enclosing it in square brackets expands it:

```
log1'
```

```
ans (a column vector) =
```

```
 1
 1.18921
 1.41421
 1.68179
 2
```

```
[time]
```

```
ans (a row vector) = 0 0.01 0.02 0.03 0.04...
```

To form a vector with descending values, use a negative step:

```
k2=[2:-.25:1]
```

```
k2 (a row vector) = 2 1.75 1.5 1.25 1
```

To reverse a vector, use a negative step value:

```
k3=k2(length(k2):-1:1)
```

```
k3 (a row vector) = 1 1.25 1.5 1.75 2
```

Use vectors in expressions and to define new matrices:

```
g=[1:3;logspace(1,20,3)]
```

g (a rectangular matrix) =

```
1   2   3
1  4.47214  20
```

### Matrix Index Operations

Create the matrix **testm**:

```
testm = [1:3;4:6;7:9]
```

testm (a square matrix) =

```
1   2   3
4   5   6
7   8   9
```

To find any element in **testm**, give the matrix name followed by the row and column index in parentheses:

```
testm(2,3)
```

ans (a scalar) = 6

To find the second row in **testm**, use a colon (:) as a wildcard symbol in place of the column index to denote "second row, all columns":

```
testm(2,:)
```

ans (a row vector) = 4 5 6

To find any column in **testm**, use the wildcard symbol (:) in the rows position:

```
testm(:,1)
```

ans (a column vector) =

```
1
4
7
```

To find submatrices, use vector inputs:

```
testm(1:2,2:3)
```

ans (a square matrix) =

```
2   3
5   6
```

The function `find()` allows you to find indices for matrix elements that meet a certain criterion. `find()` returns each index in `[row, column]` format.

```
find(testm > 7)
ans (an index list) =
     3     2
     3     3
```

The output indicates that the elements found in the third row, second and third columns (3,2) and (3,3) are greater than 7.

You can incorporate `find` results as a special indexing scheme to perform an operation on only the elements meeting the criterion in `find`.

```
testm(find(testm > 7)) = 0
test_matrix (a square matrix) =
     1     2     3
     4     5     6
     7     0     0
```

Xmath changed the elements greater than 7 to zeros.

### Using Matrix Functions

Matrix functions take advantage of the structure of matrix objects. The more specialized a matrix is (that is, the more properties it inherits), the greater the computational speed improvement. For example, consider computing the eigenvalues of a common matrix, a symmetric matrix, and a triangular matrix of the same size (100 × 100).

The `clock()` function monitors elapsed CPU time. It returns the time in seconds since `clock()` was last called. Therefore, you should call it before and after the monitored process.<sup>1</sup>

```
rmat = random(100,100);
clock({cpu}); mm = eig(rmat); clock({cpu})?
```

Xmath automatically uses more efficient algorithms when the matrix fits a given structure. The above example tells how long it takes to find the eigenvalues of a general, random (100 × 100) matrix.

---

1. `clock()` results depend on your machine's configuration.

In the following example, you can see how long it takes with a symmetric matrix of the same size. We use the transpose operator (') to ensure that the matrix is symmetric:

```
smat = rmat * rmat';  
clock({cpu}); mm = eig(smat); clock({cpu})?
```

`eig` takes even more advantage of a triangular matrix:

```
tmat = triu(rmat);  
clock({cpu}); mm = eig(tmat); clock({cpu})?
```

Xmath checks object properties before computations so that it uses the fastest algorithms and performs no unnecessary computations.

### 2.5.3 Polynomials

To create a polynomial, specify its roots with the `polynomial()` function, or specify its coefficients with `makepoly()`:

```
poly1 = polynomial([1,5])
```

```
(x - 1)(x - 5)
```

```
poly2 = makepoly([1:.7:4.5])
```

```
5      4      3      2  
x + 1.7x + 2.4x + 3.1x + 3.8x + 4.5
```

The default variable name is `x`. Both functions have an optional string argument that specifies the variable name. For example:

```
p = polynomial([1+jay,1-jay],"s")
```

```
p (a polynomial) =
```

```
2  
(s - 2s + 2)
```

Several operators and functions are defined differently for polynomials than they are for matrices.

Multiplying two polynomials with the `*` operator returns the polynomial convolution:

```
poly3 = poly1*poly1
```

```
poly3 (a polynomial) =
```

$$(x - 1)^2 (x - 5)^2$$

When adding two polynomials, the corresponding order terms of the two polynomials are added:

**poly1+poly3**

```
ans (a polynomial) =
    4    3    2
x - 12x + 47x - 66x + 30
```

Similarly, when adding a scalar and a polynomial, the scalar is added to the scalar term of the polynomial:

**poly1+1**

```
ans (a polynomial) =
    2
x - 6x + 6
```

When multiplying a polynomial and a scalar, the output format depends on the format of the polynomial:

**poly1\*2**

```
ans (a polynomial) =
    2(x - 1)(x - 5)
```

Use **roots()** to find the roots of a polynomial:

**roots(poly3)**

```
ans (a column vector) =
    1
    1
    5
    5
```

Use **polyval()** to evaluate the polynomial with a scalar value for the variable:

**polyval(poly2,3)**

```
ans (a scalar) = 489.3
```

Indexing into a polynomial is similar to indexing into a matrix. To find and change the coefficient of the third element, type:

**poly2(3)**

```
ans (a scalar) = 2.4

poly2(3) = 9

poly2(a polynomial) =

      5      4      3      2
x + 1.7x + 9x + 3.1x + 3.8x + 4.5
```

## 2.5.4 Dynamic Systems

Xmath represents a dynamic system as either a transfer function or a state-space system. A transfer function consists of two polynomials; a state-space system is represented by four matrices. Transfer functions can only represent single-input single-output (SISO) systems, but state-space systems can represent multiple inputs and output (MIMO) systems. Objects for both types of systems can be either discrete or continuous, depending on the value of the object's sample rate.

### Transfer Functions

A transfer function is built from numerator and denominator polynomials:

```
num = makepoly([1,-163,5.5]);
den = makepoly([1,2.7,5.6,3.5,8.1]);
```

Use **system** to create the transfer function:

```
sysTF = system(num, den, {dt = 1})
```

```
sysTF (a transfer function) =

      2
      x - 163x + 5.5
-----
      4      3      2
x + 2.7x + 5.6x + 3.5x + 8.1

initial delay outputs
0
0
0
0
Input Names
-----
Input 1

Output Names
-----
Output 1
```

System is discrete, sampling at 1 seconds.

If you do not wish to specify a sampling rate, you can use the shorthand form:  
**sys=num/den.**

To extract the numerator or denominator of a transfer function, use **numden**:

```
[n,d]=numden(sysTF)
n (a polynomial) =
      2
     x - 163x + 5.5
d (a polynomial) =
      4      3      2
     x + 2.7x + 5.6x + 3.5x + 8.1
```

### State-Space Systems

To create a state-space system of the form

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

use **system** with four matrices as inputs:

```
ha=[1,0,0,.1; 0,-.2,.1,0; 0,1,0,0;-.2,0,0,1];
hb=[.5,0,0,.3]';
hc=[1,0,1,0];
hSS=system(ha,hb,hc,0)
```

hSS (a state space system) =

```
A
  1      0      0      0.1
  0     -0.2    0.1     0
  0      1      0      0
 -0.2    0      0      1
```

```
B
  0.5
  0
  0
  0.3
```

```
C
1  0  1  0
```

```
D
0
```

```
X0
0
0
0
0
```

System is continuous

Notice that Xmath creates continuous systems by default. To create a discrete system, include the keyword **dt**, which sets the sampling period in seconds:

```
hSSd=system(ha,hb,hc,0, {dt = .1});
```

To extract the state and initial condition matrices from a system, use **abcd**:

```
[ A,B,C,D,X0 ] = abcd(hSSd)
```

The functions **sys2sns()** and **sns2sys()** might interest you:

- **sns2sys()** converts a system from MATRIX<sub>X</sub> to an Xmath object.
- **sys2sns()** converts an Xmath system object to MATRIX<sub>X</sub> format.

## Analyzing Dynamic Systems

You can display the time domain response of a system using the functions in [Table 2-2](#).

Table 2-2 **Time Display Functions**

---

<b>impulse()</b>	Computes the impulse response of a system.
<b>initial()</b>	Computes the unforced response of a system to a given initial condition.
<b>step()</b>	Computes the step response of a system.

---



Table 2-2 **Time Display Functions** (Continued)

---

<b>defTimeRange()</b>	Computes a default time vector for simulations.
<b>sys*u()</b>	Performs a general simulation, where <b>u</b> is a PDM representing system input.

---

These functions return parameter dependent matrices (PDMs), our next topic. For more on these functions, see the MATRIX<sub>X</sub> online Help.

**bode**, **nyquist**, and **nichols** display frequency-domain response in several standard formats. For example, to see the bode plot of the continuous-time system we created earlier, type:

```
bode(hSS)?
```

### 2.5.5 Parameter Dependent Matrices

A parameter-dependent matrix (PDM) is a collection of same-size matrices, with a vector (called the *domain*) attached; each matrix depends upon a corresponding element of the domain vector. A PDM stores matrices as functions of an independent element parameter (the domain). A PDM is often a matrix of a physical parameter, such as time, frequency, or speed.

PDMs are built from string, vector, and matrix objects using the **pdm()** function. For example, the following PDM stores data in a legible compact format:

```
d=[95:99];
AR=[ 60.8; 59.3; 54.4; 50.7; 50.7];
CO=[ 41.2; 41.7; 36.3; 35.7; 35.3];
OR=[ 46.1; 47.5; 47.6; 46.7; 48.7];
WA=[ 45.4; 45.6; 44.0; 43.2; 43.9];
states=["AR","CO","OR","WA"]
eJobs=pdm([AR,CO,OR,WA],d,{domainName="Year",columnNames=states})
```

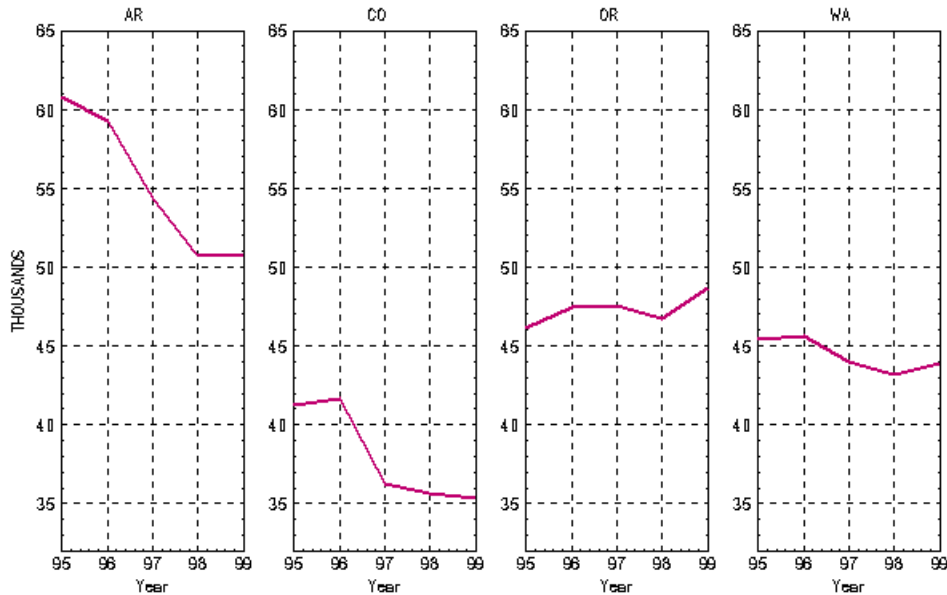
```
eJobs (a pdm) =
```

Year	AR	CO	OR	WA
95	60.8	41.2	46.1	45.4
96	59.3	41.7	47.5	45.6
97	54.4	36.3	47.6	44
98	50.7	35.7	46.7	43.2
99	50.7	35.3	48.7	43.9

The advantage of storing the data, names, and domain together is clearer when we create a plot such as [Figure 2-8](#).

```
g2=plot(eJobs,{strip,ymax=65,ymin=32,ylab="THOUSANDS",  
line_color = "mulberry", line_width = 2})
```

Figure 2-8 PDM Plotted with the **strip** Keyword



PDMs are commonly seen as outputs from functions, such as those listed in [Table 2-2](#). If we calculate the impulse response and step response of **hSSd** (the discrete state-space system created earlier), the responses are formatted as PDMs. The output is too long to show here, but you can view it in the log area:

```
hIm=impulse(hSSd);  
hSt=step(hSSd)?
```

It is convenient to store these related PDMs together in another PDM:

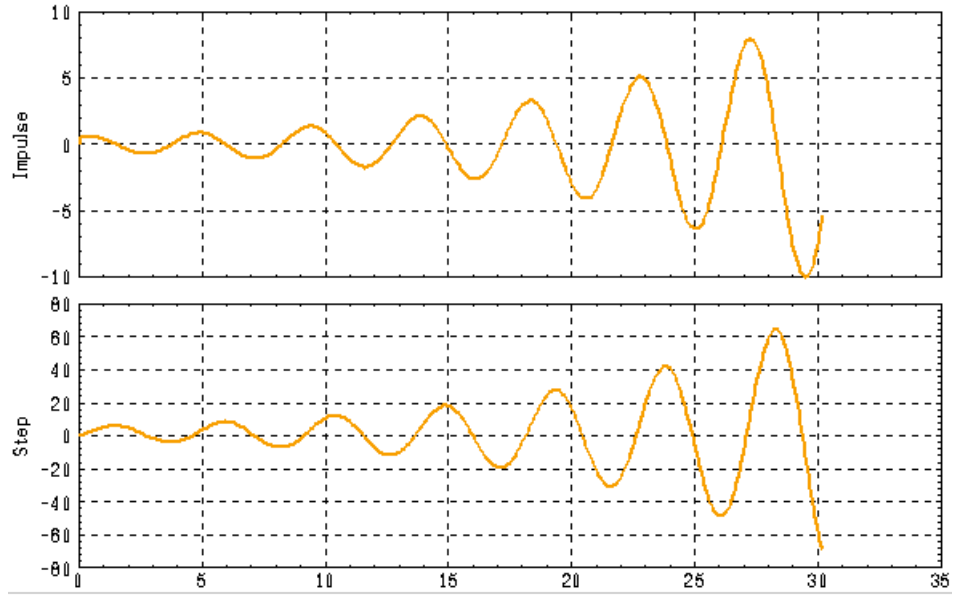
```
hPdm=pdm([hIm;hSt],{rowNames=["Impulse","Step"]})?
```

Plot the responses separately with the **strip** keyword:

```
plot(hPdm,{strip})
```

shows the results.

Figure 2-9 PDM Impulse and Step Responses Plotted Separately



The size of a PDM is given as rows  $\times$  columns  $\times$  length of the domain:

```
size(hPdm)
```

```
ans (a row vector) = 2 1 303
```

Portions of a PDM are accessible with indexing, similar to matrices. Extract the fifth dependent matrix from **hPdm**:

```
hPdm(5)
```

```
ans (a pdm) =
```

```
domain |
-----+-----
 0.4 | Impulse 0.5594
      | Step    2.1394
-----+-----
```

To look at only the impulse responses, type:

```
hPdm(1,1)
```

```
ans (a pdm) =
```

```
domain |
-----+-----
      0 | Impulse 0
     0.1 | Impulse 0.5
     0.2 | Impulse 0.53
     0.3 | Impulse 0.55
     0.4 | Impulse 0.5594
     0.5 | Impulse 0.5578
        :           :           :
```

To perform a general simulation, you can multiply a system by a PDM. Here we use **freq** to create a PDM.

```
u=freq(hSSd,deftimer(hSSd));
Y=hSSd*u;
plot(Y)
```

For more detailed information on PDMs, see [5.4](#), p.187.

## 2.5.6 Lists

A list object is a named collection of elements (objects). A list can contain varied objects (including other lists). It is one-dimensional, storing your specified objects regardless of dimensions or properties. Use the **list()** function to create this object:

```
set seed 0
scalar1 = 1;
string1 = "This is a string object";
poly = makepoly([1,2]);
matrix = random(5,5);
a_list = list(scalar1,string1,poly,matrix)

a_list (a list with 4 elements) =

1:
  1

2:
  This is a string object

3:
  x + 2

4:
  0.211325    0.756044    0.000221135    0.330327    0.665381
  0.628392    0.849745    0.685731    0.878216    0.068374
  0.560849    0.662357    0.726351    0.198514    0.544257
  0.232075    0.231224    0.216463    0.883389    0.652513
  0.307609    0.932962    0.214601    0.312642    0.361636
```

A list containing four objects has a size of 4. To extract an element, specify its order in the list:

```
a_list(3)
ans (a polynomial) =
  x + 2
a_list(1)
ans (a scalar) = 1
```

## 2.6 MathScript

MathScript is the language of Xmath. Every instruction you have typed into the Xmath Commands window so far is a MathScript statement. With a MathScript script file, you can create and define a MathScript function, command, or object as MathScript entities, which are immediately available for use without special linking or compiling. (Chapter 6 describes how to create, define, and debug MathScript entities.)

### 2.6.1 MathScript Features

MathScript provides the following features:

- Familiar programming constructs such as **for** and **while** loops and **if** statements.
- Nested expressions:  

```
x = 20 * log(abs(1 + 2 * jay))
```
- Functions to obtain interactive user input, such as **getline()** and **getchoice()**.  

```
UserIn=getline("Enter the number of states now:")
vote=getchoice("Choose or Lose", ["Repub", "Demo", "Inde"])
```
- Functions to determine whether objects possess certain properties (**check** and **is**). For example:

```
a = [1,0;0,1]
check(a, {identity})

ans (a scalar) = 1
```

These features and more programming issues are discussed in [Chapter 6](#).

## 2.6.2 Debugger Window (on UNIX Systems)

The MathScript Debugger window ([Figure 2-10](#)), referred to as “the debugger,” allows you to interactively debug MathScript. Usually the debugger is activated because a script contains a syntax error or a runtime error (see [6.4](#), p.245). It also opens if you have set up a file to be debugged. You call debug the same way for both functions and commands:

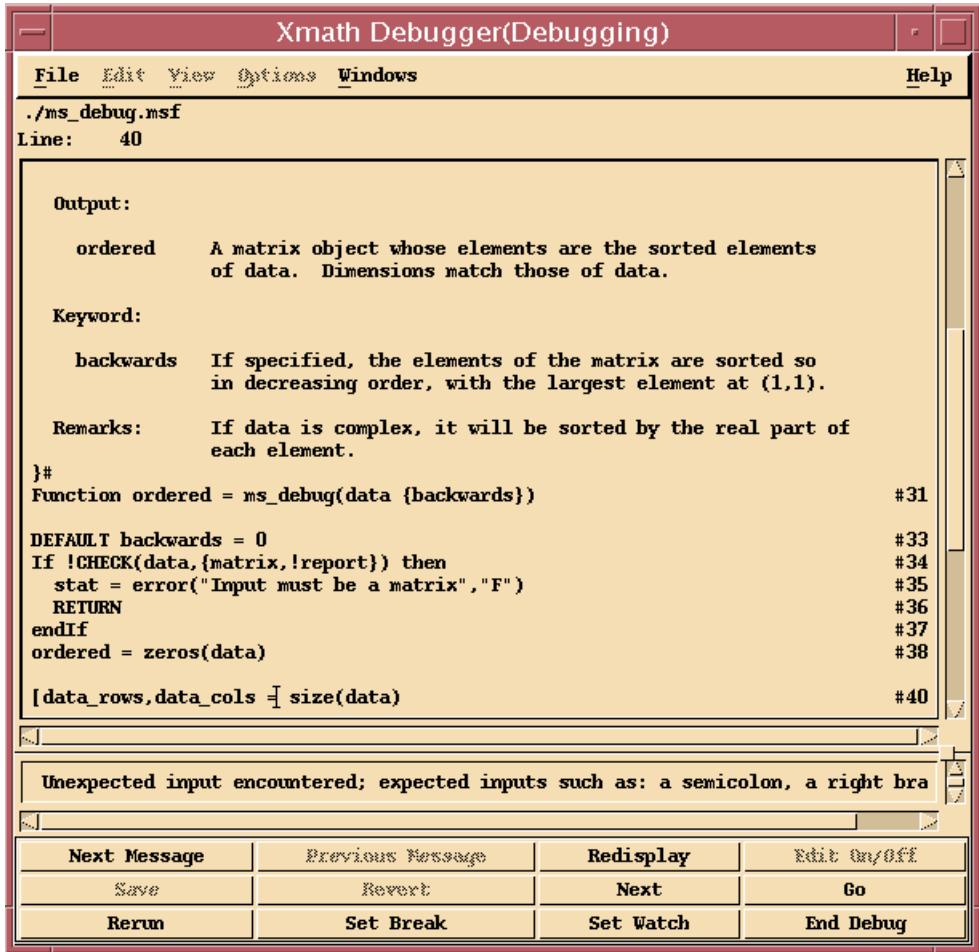
```
debug entity_name
```

The debugger opens whenever the function or command is invoked. To turn off debugging, type:

```
debug entity_name off
```

When the debugger opens, the top field in the window contains the source of the MathScript function or command you are debugging. The filename is displayed below the menu bar. If you don't have write privileges to the source file, the source code may be opened read-only (not editable). The line that is about to be executed is highlighted (unless there are syntax errors in the function, in which case highlighting is used to identify the error). The message area, which displays error messages that occur during execution, is just below the source code area. You can use buttons at the bottom of the window in lieu of debugger commands.

Figure 2-10 Debugger Window (on UNIX Systems)



Run the debugger demo. It instructs you on how to edit an MSF that contains syntax errors. From the command area, type:

```
execute file="$XMATH/demos/debuggingMS1"
```

For more on the debugger window, see [6.4](#), p.245.

## 2.7 GUI Tools

Xmath offers a programmable graphical user interface (PGUI or GUI). For an introduction to the GUI, and instructions on starting and using the GUI demos and tools, see [Chapter 9, Graphical User Interface](#).

To see some examples of GUI tools, type:

```
guidemo
```

To exit Xmath, see [1.3.3 Exiting Xmath](#), p.10.

## 2.8 Conclusion

This concludes the Xmath tutorial.

As you worked through the tutorial, you've become acquainted with the concepts and procedures necessary to use the basic Xmath features (described in Chapters [3](#) through [5](#)). Chapters [6](#) through [9](#) discuss advanced topics:

- [Chapter 6](#) tells how to write your own functions and commands using MathScript.
- [Chapter 7](#) tells how to create your own MathScript object.
- [Chapter 8](#) tells how to link C, C++, or FORTRAN files to Xmath, and also details how to call Xmath from an external program.
- [Chapter 9](#) tells how to program your own graphical user interface.



# 3

## *MathScript Basics*

MathScript is the language of Xmath. MathScript contains many of the facilities common to high-level programming languages, such as logical expressions loops, comments, conditional statements, nested functions and recursion.

### **3.1 MathScript Statements**

A statement is the smallest independent executable instruction. Here are some examples of statements:

```
x = 7
y = ones(3,3)
who
set format long
```

The first two statements are examples of assignments. The last two statements are examples of commands.

#### **3.1.1 Assignments**

The most common MathScript statement is an assignment. An *assignment* is a statement that sets a variable to a specific value defined by the expression on the right-hand side:

```
variable = expression
```

- If an expression output is assigned to a variable, use the question mark (?) terminator to display the result. To suppress the output, use the semicolon (;) terminator.
- A carriage return is also a statement terminator. If **set display** is on, a return displays the result; if **set display** is off, nothing is displayed (see [p.113](#)).
- Variable types do not have to be declared before assignment.
- Objects can be completely or partially modified using assignment statements combined with indexing. For example:

```
y = [100,21:24]
```

```
y (a row vector) = 100 21 22 23 24
```

```
y(1) = 0
```

```
y (a row vector) = 0 21 22 23 24
```

### 3.1.2 Rules for Names

Variable names consist of alphanumeric characters and internal underscores (\_) only.

- Name components must be less than 32 characters in length. For example, variable **b** in partition **a** (**a.b**) could have a total of 31 characters.
- Names should not start with an underscore, because initial-underscore names are reserved for internal use.
- Variable and partition names are case sensitive. The following variables represent two partitions and four different variables:

```
a.b; A.b; a.B; A.B;
```

You can create a variable with the same name as a predefined Xmath function or command; however, you will be unable to access that pre-defined feature until you delete the variable.

### 3.1.3 Expressions

An *expression* is a combination of variable names, functions, and operators that evaluate to a single Xmath object. The Xmath object can then be assigned to a variable name. For example,

```
(1+sin(pi/4))^2 # An expression
```

Expressions can be used as arguments to other functions or operators.

```
cep = abs(fft([1,-4,8,-2]))
```

The functions `exist()` and `check()` are exceptions. These functions require a variable name as an argument.

### Logical Expressions

In MathScript, a nonzero value (with the exception NAN and Inf) is considered TRUE. All logical operators return 0 if FALSE and 1 if TRUE.

```
x = 3; x < (3 * cos(0))
ans (a scalar) = 0
```

Logical operators are “short-circuited.” For example, `exp1 | exp2 | exp3` will return 1 if `exp1` is nonzero without evaluating `exp2` or `exp3`. Therefore, careful ordering of subexpressions in logical expressions may speed up execution.

[Table 3-1](#) lists all MathScript logical operators. For a list of all Xmath operators, see [Table 3-3](#).

Table 3-1 **MathScript Logical Operators**

Operator	Effect
<	Elementwise less than.
>	Elementwise greater than.
<=	Elementwise less than or equal.
>=	Elementwise greater than or equal.
==	Elementwise equal.
<>	Elementwise not equal.
&	Elementwise <b>logical and</b> .
	Elementwise <b>logical or</b> .
!	The logical negator (!) appears directly before an expression. For example, <b>!expr</b> .

### Logical Expressions with Matrices

When used with logical operators, two matrices must be equal in size; the output will be a matrix containing the element-by-element comparison results.

```
a = [1,0;1,1];b = eye(2,2);  
a & b
```

```
ans (a square matrix) =
```

```
1 0  
0 1
```

**ans** is a matrix with 1 in the locations where **a** and **b** are the same.

```
a < b
```

```
ans (a square matrix) =
```

```
0 0  
0 0
```

You can also make logical comparisons with the functions **check()** and **is()**, which return a logical value. The functions **all()**, **any()**, and **none()** can also be used to return a logical value. See [p.240](#) or the MATRIX<sub>X</sub> online Help for more details.

### 3.1.4 Operators

An operator is a nonalphanumeric symbol that operates on its operand(s). Operators with only one operand are called *unary operators*. Operators with two operands are called *binary operators*. [Table 3-2](#) shows how operators are used in expressions.

Table 3-2 Uses of Operators in Expressions

Format	Type	Example
<b>operator</b> operand	Unary (prefix)	<b>-x</b>
operand <b>operator</b>	Unary (suffix)	<b>x'</b>
operand1 <b>operator</b> operand2	Binary	<b>x+y</b>

Table 3-3 lists the operators available in Xmath and their intrinsic functions; overloaded functions are described in other chapters.

Table 3-3 **Xmath Operators**

---

**Xmath Operators**

---

+	addition
-	subtraction (and the unary operator negation)
*	multiplication
/	right division, $A/B$ solves the equation $X*B=A$
\	left division, $B\A$ solves the equation $B*X=A$
'	transpose (unary suffix)
*'	Hermitian (complex conjugate) transpose
.*	element wise multiplication
./	element wise division (left divided by right)
.\	element wise division (right divided by left)
^	raise to a power
or **	
.^	raise elements to a power
or .**	
.*.	Kronecker product
./.	Kronecker right division
.\	Kronecker left division
&	logical AND
	logical OR
!	logical NOT (unary operator)
<	less than
>	greater than

---

Table 3-3 **Xmath Operators** (Continued)

<b>Xmath Operators</b>	
<=	less than or equal
>=	greater than or equal
==	equal
<>	not equal
=	assignment
()	indexing, precedence, and function reference
{ }	keyword delimiters in function references
[ ]	matrix construction and concatenation

Operator behavior depends on the objects involved. Special behaviors are discussed in detail in the object descriptions in [Chapter 5, Data Objects and Operators](#).

### Operator Precedence

You can control operator precedence with parentheses. In [Table 3-4](#), operators are ordered with precedence from highest to lowest (reading from top to bottom).

Table 3-4 **Operator Precedence**

high	non-associative	' *'
	left-associative	** ^ .** .^
↓	left-associative	* / \ .* ./.\ .* ./.\.
	non-associative	! unary + unary -
↓	left-associative	+ -
	left-associative	:
↓	left-associative	> < >= <= == <>
	left-associative	&
low	left-associative	

## 3.2 Partitions

All variables reside in partitions. **main** is the default partition. You do not need to specify the partition name of a variable if it resides in the current partition; just call it by its local name.

Partitions must be created using **new partition** before any variables may be placed in them. To create or use a variable in another partition, you must specify the partition name. (Partition names must meet the naming rules in [3.1.2 Rules for Names](#), p.90.) For example,

```
job1.R = R           # Assign R in the current partition to the
                    # variable R in partition job1.

job2.R = job1.R     # From the current partition, perform
                    # an assignment between two other
                    # partitions.
```

- To show the current partition, use the **show partition** command:

```
show partition

main
```

- To list all defined partitions, type:

```
show partitions
```

Notice the *s* at the end.

Please perform the following steps to get a better understanding of partitions.

1. **main** is the default partition that is created whenever Xmath is started. If you are in **main**, you can create an object in partition **main** by typing:

```
xx = 1
```

This is equivalent to **main.xx = 1**.

2. To create a new partition named **var**, type:

```
new partition var
```

3. You can navigate between partitions with the **set partition** command:

```
set partition var

show partition

var
```

```
xx  
xx not found.
```

4. Because **xx** is defined in partition **main**, its partition name must be included:

```
main.xx?           # variable from another partition  
main.xx (a scalar) = 1  
  
yy = 55?           # create variable in current partition  
yy (a scalar) = 55
```

5. Return to the **main** partition. The original **main.xx** is in local scope, while **yy** exists in the partition **var**.

```
set partition main  
  
xx  
xx (a scalar) = 1  
  
var.yy  
var.yy (a scalar) = 55
```

6. A partition must be empty before it can be deleted. To delete a partition, use the **delete** command. First, delete the partition contents, then the partition itself:

```
delete var.* var.
```

### 3.2.1 Listing Defined Variables

To list all defined variables in the current partition, use the **who** command:

```
who
```

A single wildcard can be used with **who**:

```
who a*             # List variables in the current partition  
                  # that start with a.  
who otherPartition.*1 # List all variables that end in 1 in another  
                  # partition.
```

To list all variables in all partitions, type the following:

```
who *.*
```



## Wildcards

Xmath allows the asterisk (\*) and percent (%) symbols to be used as wildcards for viewing, saving, loading, or deleting variables.

An asterisk denotes “any characters.” Used by itself, an asterisk is a wildcard for all names. Therefore, **delete \*** deletes all variables in the current partition. Used with other characters, an asterisk replaces any number of characters in that position. The percent sign replaces a single character in that position.

```
a3=4; a23=1; b22=144; c23=random(a3,a23);

who*           #Show variables in the current partition.
who a*        #Show variables starting with a.
who *3        #Show variables ending with 3.
who %2%"      #Show 3-character names where 2 is
              #the second character(a23, b22, c23).
```



**NOTE:** You cannot use the wildcard \* twice in a pattern. For example, \*sys\* is not allowed, but \*sys%% is accepted.

### 3.2.2 Variable and Partition Comments

You may attach a comment string to a variable or partition name with the **comment** command.

```
comment main. "this is the default partition"
a=97;
comment a "the first letter of the alphabet"
```

- To retrieve the comment, use **commentof()**:

```
commentof(a)

ans (a string) = the first letter of the alphabet

commentof(main.)

ans (a string) = this is the default partition
```

- You can also view a variable’s comment if you invoke the Xmath Variables window (see [3.2.5 Xmath Variables Window](#), p.99).

### 3.2.3 Permanent Variables

Permanent variables are values that have special meanings. These variables are defined in all partitions as shown in [Table 3-5](#)

Table 3-5 Permanent Variables

Variable	Definition
<b>Inf</b>	infinity
<b>Jay</b>	sqrt(-1)
<b>NaN</b>	Not a Number
<b>eps</b>	very small number used to initialize outputs to be near zero but not exactly zero
<b>huge</b>	largest finite number less than <b>Inf</b>
<b>null</b>	empty object
<b>pi</b>	famous Greek number
<b>tiny</b>	smallest possible number greater than 0
<b>err</b>	global error status variable (set to NaN)

The name of a permanent variable or predefined function/command can be overridden in the current partition or function/command scope, although it is *not* recommended. When a value that has been given the name of a permanent variable is deleted, the original definition reappears:

```
eps=2
eps (a scalar) = 2
delete eps
eps?
eps (a scalar) = 2.22e-16
sin=1?
sin (a scalar) = 1
sin(pi) # argument out of range
delete sin
```

```
sin(pi)
ans (a scalar) = 1.22465e-16
```

### 3.2.4 *ans*

When a value returned from a function is not assigned to a variable name, it is assigned to the variable **ans**.

```
sin(0.5)
ans (a scalar) = 0.479426
```

The value of **ans** is overwritten anytime the output of a function is not assigned to a variable. Note that the value of **ans** is local to the current partition.

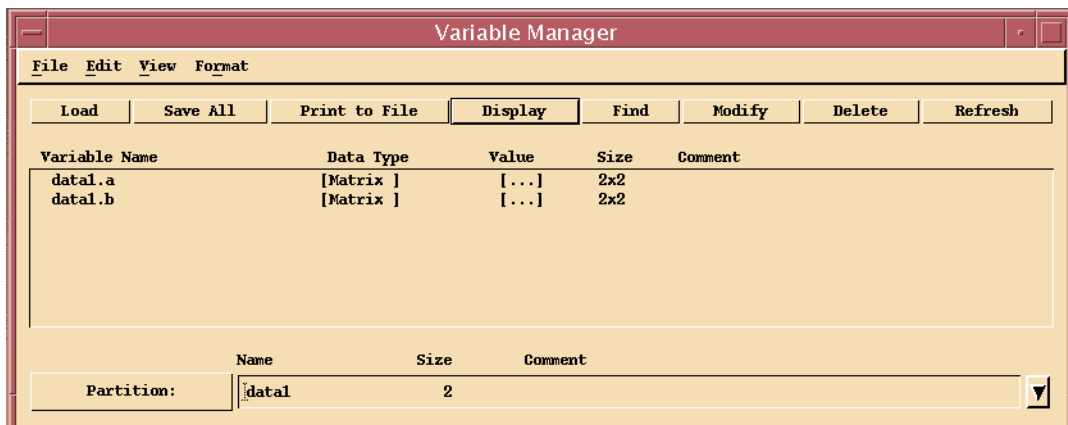
### 3.2.5 *Xmath Variables Window*

The Xmath Variables Manager window (shown in [Figure 3-1](#)) is a graphical interface that simplifies variable management. From this window, you can view variable and partition information, and load and save data.

Select Windows→Variables to invoke the Variables Manager window. The Variables Manager window lists all variables in the current partition. At a glance you can see the variable's type, size, and attached comments (if any). If a variable is locked, an @ **sign** appears on the far left. To display a variable, double-click on it. Numeric and string objects are displayed in the log area of the Xmath Commands window, and graph objects are displayed in the Graphics window.

Because the Variables Manager window is updated each time the value of a variable changes, it is a good idea to minimize or close the window when you don't need it. Leaving it open while executing a lengthy **For** or **While** loop, for example, decreases Xmath's execution speed.

Figure 3-1 Xmath Variables Window (UNIX version)



### Fields

The mid portion of the window is devoted to variable information. Partition information is displayed at the bottom of the window. To view variables in a different partition, click the Partition button, and then select another partition from the subsequent dialog.

**Variable Name** — The name of the variable.

**Data Type** — For variables, displays the major type: matrix, vector, polynomial, PDM, system, string, list, or graph.

**Value** — The value of the variable.

**Size** — For variables, the dimension of the object. For partitions, the number of variables in the partition. See the MATRIX<sub>X</sub> online Help *size* topic for more information.

**Comment** — Displays the comments attached to the partition or the variables. You can scroll to see lengthy comments for variables, but you may need to resize the Variables Manager window to see a lengthy partition comment.

**Name** — The name of the partition.

## Menus

The Variables Manager window provides several pull-down menus with many functions. Some of the most common functions on the menus are also available from buttons immediately below the menu bar.

**File** — Allows you to save and load variables. For an explanation and an example of how to use the file selection dialog, see *Specifying Directory Pathnames and Filenames* on p.18. The **load** command accepts data saved from Xmath or MATRIX<sub>x</sub>.

If you want to load data that has not been created by the above applications, go to the command area in the Commands window and use the **read** command. **read** can place part or all of a data file into an Xmath matrix variable (see *3.7.3 Reading Non-Xmath Data Files into Xmath*, p.112). **save** and **load** operations can also be accomplished from the command area of the Commands window (see *3.7 Saving and Loading Data*, p.108).

The File menu also allows you to print and perform standard window operations.

**Edit** — Lets you perform various editing functions for the partition or variable.

**Copy, Rename, and Modify Data** operate on a selected variable; note that wildcards are not allowed here. **Modify Data** also allows you to lock or unlock a variable.

**Delete** removes the selected variable, and **Undelete** retrieves the last deleted variable.

**View** — Controls the order in which variables are listed. **By Name** displays variables in case-sensitive alphabetical order from top to bottom. **By Date** (the default) displays variables in the order created. The latest variable is shown at the bottom of the list. **By Size** shows variables sorted by dimension from top to bottom. **By Type** shows objects grouped by alphabetized type.

**Format** — Allows you to set the format of the Value field for variables.

**Options** — Provides a Find function along with select and deselect functionality.

**Find** searches the current partition for the specified variable, and lists the result. A single wildcard is allowed in the find pattern. To find the specified variable in all partitions, select **Edit**→**Find** and specify the \* wildcard in place of the partition name in the **Pattern** field. For example, to find the variable **a** in all existing partitions, specify the following pattern: **\*.a**.

### 3.3 Punctuation

MathScript has special uses for the ?, ;, ..., #, and . characters. These are illustrated in [Table 3-6](#).

Table 3-6 Punctuation Mark Usage

---

?	<p>A question mark is a statement terminator. When placed after a numeric or string object, the value is displayed in the log area; when placed after a graph object, the graph is displayed in the Graphics window.</p> <pre>y = eye(3,3)?x=y/2;</pre> <p>y (a square matrix) =</p> <table style="margin-left: 40px;"><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>1</td></tr></table> <p>Interactively, the default display behavior (which can be changed via <b>set display</b>), is to display the output of all assignments and expressions not terminated by a semicolon. If this is the behavior, the question mark is only needed as a separator. If <b>set display</b> is turned off, output is suppressed unless a question mark is used (see <a href="#">p.113</a>).</p>	1	0	0	0	1	0	0	0	1
1	0	0								
0	1	0								
0	0	1								
;	<p>A semicolon (;) disables display to the log area, and acts as a separator or terminator. A semicolon disables display regardless of whether set display is on or off.</p> <pre>x = 1:3:10; x'</pre> <p>ans (a column vector) =</p> <table style="margin-left: 40px;"><tr><td>1</td></tr><tr><td>4</td></tr><tr><td>7</td></tr><tr><td>10</td></tr></table>	1	4	7	10					
1										
4										
7										
10										

---

Table 3-6 Punctuation Mark Usage (Continued)

---

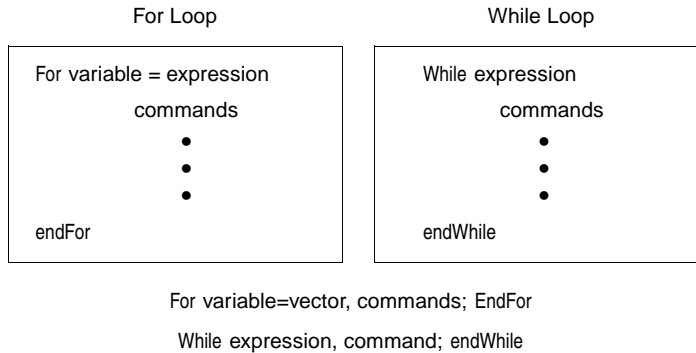
...	<p>An ellipsis (...) is a continuation symbol that allows an Xmath statement to span multiple lines:</p> <pre>plot ... (rand(1,50),{title="Testgraph",line_style=1})</pre> <p>Ellipses are not required if a line ends with a comma, or an operator:</p> <pre>plot (x,y,z,{x_lab="Hello",y_lab="Goodbye", z_lab="Leave town before sundown!"})</pre> <p>However, you cannot continue all commands, even if you use the ellipsis. For example, you cannot split an output assignment; thus, the following multiple line entry results in an error:</p> <pre>[blocknr=selectedblocks,sbname=name, sbin=inputs,sbinname=inputname, sbout=outputs,sboutname=outputsignal] = querysuperblock();</pre> <p>You could split this example input before or after the equal sign (=) but nowhere else.</p>
#	<p>A pound sign (#) comments out everything to the right on a single line. To comment multiple lines of text, surround them with #{ }#.</p> <pre>#Comment a single line #{You can comment multiple lines}#</pre>

---

### 3.4 Iterative Conditional Statements

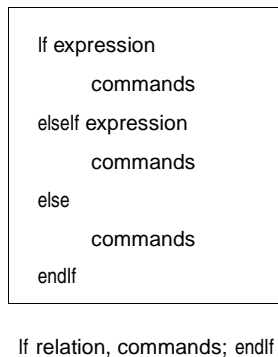
In MathScript, **For** and **While** loops have the syntax shown in [Figure 3-2](#).

Figure 3-2 **For and While Loops**



**If** statements in MathScript have the syntax shown in [Figure 3-3](#).

Figure 3-3 **If Statements**



Note that **end** can be used in place of **endFor**, **endWhile**, or **endIf**.



## 3.5 Using Predefined Functions and Commands

To determine the origin of a function or command use the **whatis** command:

```
whatis freq
```

```
freq : intrinsic function
```

```
whatis bode
```

```
bode : ISI function (path/bode.xf)
```

```
whatis build
```

```
build: intrinsic command
```

- Entities referred to as Wind River functions and commands are written in MathScript. You can view the Wind River function and command MathScript source in the location returned by **whatis**, as shown above.
- Intrinsic functions and commands are written in C++ and built into Xmath by Wind River; you cannot view this source. [Chapter 6](#) describes how to use MathScript to define your own functions (MSFs) and commands (MSCs). The characteristics of Xmath objects are also intrinsic; [Chapter 7](#) describes how to use MathScript to define your own objects (MSOs).

### 3.5.1 Command and Function Calling Syntax

The rules described in this section are general; they apply to both intrinsic functions and commands and MathScript functions and commands.

- The names of functions, commands, and keywords are case-insensitive.
- Function and command names can be abbreviated to minimum of four letters, or the minimum number of characters that uniquely identify the name.

For example:

```
cova([1,2;3,4]);  
t = makep([1,2,3,4]);
```

**covariance()** can be called by specifying the first four characters, while **makepoly()** must be abbreviated to five characters (because it conflicts with **makematrix()**):

- Function inputs, keywords, and outputs are separated by commas.

```
[Ke, ev, P] = estimator(Sys, Qxx, Qyy)
```

## Aliases

Names or strings can be aliased to a shorter string with the **alias** command (see [3.8.3 Abbreviating Command Names \(alias and unalias\)](#), p.116). Then you can refer to the name or string by its alias. For example:

```
alias ef execute file  
alias ts title="TOP SECRET";plot(A, {ts})
```

## Input Arguments

- The syntax for calling intrinsic commands and MathScript commands is slightly different. Inputs for MathScript commands are separated by commas, similar to MathScript functions.<sup>1</sup> For example:

```
xgraph t, {tgraph, average}
```

The majority of commands supplied with the Xmath Core are intrinsic (see [3.5 Using Predefined Functions and Commands](#), p.105), and the arguments are separated by spaces:

```
save "filename" a b c
```

Use the syntax shown in the MATRIX<sub>X</sub> online Help when in doubt.

- Functions and commands cannot be called with fewer than the required number of input arguments, or more than the maximum number of inputs (as specified in the syntax shown in the MATRIX<sub>X</sub> online Help).

## Keywords

- Keywords are optional and case insensitive. Keywords must be placed inside curly braces { }, but the order is not significant.
- A value can be assigned to a keyword. Keywords with no value assigned are given Boolean values.

For example, the following calls give an identical result:

- 
1. On the other hand, SystemBuild SBA commands are all written in MathScript, and use this syntax exclusively.

```
g=plot(x,{legend,!grid})?
g=plot(x,{legend=1,grid=0})?
```

If a keyword is specified but not assigned to an expression, its value is set to 1. This is useful for Boolean keywords, because 1 is interpreted as TRUE. Preceding a keyword with the negation operator (!) causes its value to be set to zero, or logical FALSE. The plot keywords specified above display a legend and no grid lines.

- Expressions can be used as arguments to keywords.

```
t = plot (x, {x_max = (4 * 256), x_lab="time"})
```

### Single and Multiple Output Arguments

- As discussed in [3.2.4 ans](#), p.99, if no output variable is specified, the output is assigned to the default variable `ans`.
- To view and assign multiple function outputs, an output name must be specified in square brackets on the left side of the equation for each output needed.

```
[T,S] = schur(A);
```

- If functions return multiple arguments, the output arguments will be matched left to right. Consider the function `size`:

```
[outputs,inputs,states] = size(aSystem)
```

If a multiple output function is called with a single output name, the output will take the value assigned to the leftmost output according to the function syntax.

```
x = size(aSystem)      # returns outputs
[x,xx] = size(aSystem) # returns outputs, then inputs
```

- You can skip specific output arguments. To do this, use commas as placeholders.

```
[, ,states] = size(aSystem)
```

- Functions cannot be called with fewer than the required number of input arguments or more than the maximum number of outputs (as specified in the syntax shown in the MATRIX<sub>X</sub> online Help).

See also [Variable Arguments](#) on p.252.

## 3.6 Operating System Interface

The `oscmd()` function lets you use operating system commands while in the Xmath environment. The output of the operating system command is displayed in the Commands window log area. For example:

```
oscmd("ls")      # UNIX
oscmd("dir")     # Windows
```

The return value of `oscmd()` is the exit code of the operating system command. For UNIX, if the command passed to `oscmd()` is backgrounded with `&`, the return status is 0, not the execution status of the background command. This behavior is consistent with UNIX calls.

### 3.6.1 Manipulate and Show Current Directory

The Xmath command `set directory` defines the default working directory. Here's how to change this directory:

```
show directory  # Show current working directory.

/home/usr/xmath

set directory  "/home/projX"
save x y z    "3dTest.ms"
```

To set the directory via a dialog, select File→Set Directory.

## 3.7 Saving and Loading Data

Xmath provides commands for reading data files and writing Xmath objects in files. One pair of commands, **SAVE** and **LOAD**, works directly on Xmath objects and files. To increase the flexibility of the interface, the commands **PRINT**, **READ**, and the function `fprintf()` work with a wider variety of file formats.

The **SAVE** command writes Xmath variables to a file if entered without arguments:

```
save
```

All variables in all partitions are written to the binary file **save.xml**, in the current working directory. This is equivalent to selecting File→Save All.

The **LOAD** command without arguments loads the file **save.xml** from the current working directory:

```
load
```

Alternatively, selected objects can be saved and loaded, and you can specify a different filename:

```
a = 1:1:10; b = "this is a test"; c = 55;
```

Save **a** and **b** in file **mysave.xml**:

```
save a b "mysave"; b = 27000;
```

Save **b** and **c** in file **saveagain.xml**:

```
save b c "saveagain"  
delete *  
load b "mysave"  
b
```

```
b (a string) = this is a test
```

```
load b "saveagain"  
b
```

```
b (a scalar) = 27000
```

The extension **xml** is appended to the filename unless you specify a different extension.

Objects with the same names as objects in the loaded file are overwritten. For example:

```
a = 1:1:10;  
aa = "this is a test";  
save  
aa = 55
```

```
aa (a scalar) = 55
```

```
load  
aa
```

```
aa (a string) = this is a test
```

The data is saved in Xmath binary format by default. Alternatively, the data can be saved in an Xmath ASCII, MATRIX<sub>X</sub> binary, or MATRIX<sub>X</sub> ASCII (FSAVE) format.

```
save "mysave" {ascii}           # Xmath ASCII
save "mysave" {MATRIXx}        # MATRIXx binary
save "mysave" {MATRIXx, ascii} # ASCII
```

See the MATRIX<sub>X</sub> online Help topics *SAVE* and *LOAD* for more information. For information on how to save and load files in Xmath format without starting Xmath, see [LNX and UCI Functions](#) on p.283.

### 3.7.1 ASCII Versus Binary Considerations

Format selection (ASCII or binary) is a tradeoff between loading speed and portability.

Compared to the ASCII format, the binary format loads faster in Xmath. The larger the data file, the more noticeable the speed advantage will be. On the other hand, the binary format is typically larger in size and is not portable across different Xmath platforms. For example, a data file created on SunOS will not be usable on Windows NT. Furthermore, a binary data file must be transferred as binary, for example, via the binary mode in FTP.

Before you send a binary data file via email, you must first encode the file with **uuencode** (or an equivalent mail encoder), and the recipient of the email can then use **uudecode** to recover the original binary file.

The ASCII format is fully portable. An ASCII format save file can be transferred to any Xmath platform with NFS, FTP, or email. However, some email gateways have restrictions on the length of lines of the email content. For such systems, the save file, even though it's ASCII, should be treated as a binary file for the purpose of email transmission as mentioned above. Again, this requirement is the same for non-Xmath files that contain long lines.

### 3.7.2 Saving Data in Non-Xmath Formats

#### **print**

The **PRINT** command outputs Xmath data to a file.

```
a = [1.1,2.2,3.3;4.4,5.5,6.6];
print a file="print.tst"
oscmd("more print.tst")           #UNIX

:::::::::::::
print.tst
:::::::::::::
main.a =
1.1    2.2    3.3
4.4    5.5    6.6

ans (a scalar) = 0
```

If a file of the same name exists, it will be overwritten.

#### **fprintf()**

Using the same conventions for formatting as the C language routine **fprintf()**, the **fprintf()** function converts numeric values to a string representation for display, and writes them to an external file. For example:

```
N = 3;
s=fopen("fpr.asc","w");
s=fprintf("fpr.asc","%d Laws of Motion\n",N)
```

where **n** is the newline escape character sequence (see [5.6.2 Special Characters in Strings](#), p.216). If an **fprintf()** call uses a filename that already exists, the output will be appended to the existing file:

```
s=fprintf("fpr.asc","%d Laws of Thermodynamics\n",N)
```




---

**NOTE:** You can use the keyword **reset** to specify that the output file (if it already exists) be truncated.

---

Print out the contents of the newly created file to the log area:

```
oscmd("more fpr.asc")           # UNIX
oscmd("type fpr.asc")           # Windows

:::::::::::::
fpr.asc
```

```
.....  
3 Laws of Motion  
3 Laws of Thermodynamics  
  
ans (a scalar) = 0
```

Refer to the MATRIX<sub>X</sub> online Help for more information on **PRINT** and **fprintf()**.

### 3.7.3 Reading Non-Xmath Data Files into Xmath

The **read()** function reads data files of binary numbers or ASCII text files into an Xmath matrix. The syntax for **read()** is:

```
matrix=read(filename,out_rows,out_cols,type,seek)
```

**read()** can be called with just the filename argument, in which case the entire content of the file is read into an Xmath string value.

See the MATRIX<sub>X</sub> online Help *read* topic for more examples.

## 3.8 MathScript Environment

The **SET**, **SHOW**, **GET**, and **REMOVE** commands allow you to customize the MathScript environment. The **SET** command affects many settings, including data output format, and random distribution. Commands such as **SHOW** and **REMOVE** and the function **get()** support other utilities for displaying current variables and resetting conditions.

### 3.8.1 Changing Environment Settings

Certain aspects of the MathScript programming environment can be modified using the **set** command. For example, **SET format** changes the numerical output format:

```
x = 0.12345678901234567890?  
  
x (a scalar) = 0.123457  
  
set format longe  
x
```



```
x (a scalar) = 0.1234567890123457e-01

set format shorte
x

x (a scalar) = 1.234578e-01

set echo on

show directory

/disk/math/test
```

Table 3-7 is a list of variables that SET controls.

Table 3-7 Environment Variables Controlled with SET

Variable	Effect
<b>autocompile</b>	Sets automatic compilation <b>on/off</b> for user-defined MSFs and MSCs (see <a href="#">MathScript Program Compilation and Execution (.xf, .xc)</a> , p.233). Default is <b>On</b> .
<b>break</b>	Use from within the Xmath debugger ( <a href="#">6.4 Using the Xmath Debugger</a> , p.245) to set a breakpoint at a specified line number.
<b>buffering</b>	Sets text buffering on/off for output to the log area. Default is <b>Off</b> . By default, Xmath sends output to the log area as soon as it is available. If you are looking for maximum possible speed, <b>SET BUFFERING ON</b> .
<b>commanddiary</b>	Records command input in the file you specify ( <a href="#">3.10 Recording an Xmath Session (Diaries)</a> , p.121).
<b>debugonerror</b>	Determines whether or not a script that contains a runtime error will be debugged. Default is <b>On</b> . See <a href="#">6.4 Using the Xmath Debugger</a> , p.245.
<b>directory</b>	Sets the working directory.
<b>display</b>	When in interactive mode, if <b>display</b> is set to <b>On</b> , the result of an assignment is displayed to the log area unless a semicolon (;) is used to suppress the output. If <b>display</b> is set to <b>Off</b> , assignment outputs are not shown unless a question mark (?) is used.  When a MathScript file is executed, the interactive <b>display</b> setting is ignored. Function outputs, including plot output, are not shown unless the question mark (?) terminator is used in the MathScript.  Default is <b>On</b> .

Table 3-7 Environment Variables Controlled with SET (Continued)

Variable	Effect
<b>distribution</b>	Sets the distribution type for the function <b>random( )</b> . Options are <b>uniform</b> and <b>normal</b> . Default is <b>uniform</b> .
<b>echo</b>	<p>Sets <b>on/off</b> echoing of contents of executed MathScript files to the Commands window log area, or the Graphics window, as the case may be (see <a href="#">Echoing an Executable File</a>, p.118).</p> <p>If you want a function output to be displayed upon execution (this includes plot output) a <i>? must</i> be used in the file, and <b>echo</b> must be on when it is executed.</p> <p>Default is <b>Off</b>.</p>
<b>format</b>	Sets numerical display output format. Choices are: <b>compact</b> , <b>engineering</b> , <b>fixed</b> , <b>long</b> , <b>longe</b> , <b>scientific</b> , <b>short</b> , <b>shorte</b> . <b>fixed</b> sets the number of decimal digits in a floating point number to the value defined with <b>set precision</b> (see precision below). Default is <b>compact</b> .
<b>partition</b>	Sets the working partition (see <a href="#">3.2 Partitions</a> , p.95). The default for a new Xmath session is <b>main</b> .
<b>path</b>	Sets a search path for user-defined MSFs and MSCs ( <a href="#">p.231</a> ). Multiple set path commands may be issued.
<b>pause</b>	Sets pause to <b>on/off</b> . If pause is set to <b>off</b> , the <b>pause</b> command is ignored. Default is <b>On</b> .
<b>precision</b>	Specify an integer representing the number of decimal digits. This number affects variable display when <b>set format fixed</b> is specified. Note, most machines cannot display more than 15 or 16 digits.
<b>seed</b>	Specify an integer to be the random seed. The random seed is reset to 0 at the beginning of each Xmath session. To find the current seed, use <b>show seed</b> or <b>get({seed})</b> .
<b>sessiondiary</b>	Record Xmath inputs and outputs in a file ( <a href="#">3.10 Recording an Xmath Session (Diaries)</a> , p.121).
<b>timestamp</b>	Turn <b>on/off</b> variable timestamping whenever a variable is changed or modified. Turning <b>timestamp</b> off can save computational time when variables used in a loop. Default is <b>On</b> .

Table 3-7 Environment Variables Controlled with SET (Continued)

Variable	Effect
<b>uiupdate</b>	Turn <b>on/off</b> variable and partition updating whenever a variable is changed or modified. Turning <b>uiupdate</b> off can save computational time when variables used in a loop. Default is <b>On</b> .
<b>watch</b>	Use within the debugger to set a watchpoint for the named variable.

The **REMOVE** command cancels or deletes environmental settings, such as **path**, **sessiondiary**, or **commanddiary** to cancel or delete the function. **REMOVE** fills this need:

```
remove commanddiary
```

To check the current setting of any **SET** parameter, use the **SHOW** command:

```
show seed
```

```
1.11121e+09
```

The function **get()** can be used to return a current setting that can then be assigned to a variable.

```
working_dir = get({directory});  
working_dir
```

```
current_dir (a string) = /home/xmath/data
```

### 3.8.2 Expanding Pathnames in MathScript Files

Commonly, pathnames are represented by environment variables. You can expand them within a MathScript file in several different ways. For example,

```
set directory = $ENVIR_VAR
```

works because **directory** is a specific option designed for the **SET** command. On the other hand, if you use a general assignment, such as

```
file = "$XMATH/foo"
```

Xmath provides the result

```
$XMATH/foo
```

because this assignment does not contain a command that was specifically designed to expand environment variables.

If you want the expected results from the assignment statement above, you should use the `get` function with the keyword `path`. For example,

```
file = get({path="$XMATH"}) + "/foo"
```

provides the expanded pathname.

You can find additional examples of this type of usage in the following files:

```
$XMATH/modules/basic/hardcopy.msc  
$XMATH/modules/basic/version.msc
```

You can also use the `oscmd()` with the `$ENVIR_VAR` format; in this case, the operating system expands the environment variable.

See the `get()`, `SET`, and `oscmd()` entries in online Help for further information.

### 3.8.3 Abbreviating Command Names (*alias* and *unalias*)

The `ALIAS` command allows you to substitute a name for a text string.

```
alias clear delete *  
alias mkm makematrix
```

To see all current aliases, type:

```
alias
```

An alias defined in any context is local to the defined scope. For example, an alias entered from the command line is not accessible from an `MSF`, `MSC`, or `MSO`. Conversely, an alias defined in an `MSF`, `MSC`, or `MSO` is not accessible from the command line.

Use the `UNALIAS` command to undo any aliases.

```
unalias clear
```

Alias substitution is performed at compilation time. Therefore, a code fragment similar to the following will not have the intended effect:

```
alias sl save  
if do_load  
  alias sl load  
endif  
sl          # Always substituted with load because that was  
           # the last alias command.
```

## 3.9 MathScript Batch Files

MathScript batch files contain sequences of Xmath statements. They are useful for setting up user environments, performing repetitious tasks, and processing programs in batch. MathScript batch files have no declaration statement, and therefore, no inputs or outputs.

Batch files are run using the **execute file** command. A MathScript batch file typically has the suffix **.ms**, but any suffix will do. If the suffix is **.ms**, you can execute the file without specifying the extension (see [3.9.1 Executing a Batch File](#), p.117).

If you do not want a function or command output to be displayed when the file is executed, use the semicolon terminator (see [p.102](#)). If you want the output to be displayed, you must use a question mark as a terminator. This also applies to the output of the **plot()** function.

Executable strings must also be terminated by a semicolon (;) or question mark (?). For example, the following is incomplete:

```
test_string = "show format";  
execute test_string
```

From the above incorrect syntax, you receive the message:

```
Error(s) in executing show format
```

The correct syntax is as follows:

```
test_string = "show format;"  
execute test_string
```

### 3.9.1 Executing a Batch File

You can execute a batch file from either the command area in the Xmath Commands window or the File menu. From the command area, use the **execute file** command. For example, to execute a batch file called **myfile.ms** in the current working directory, type:

```
execute file = "myfile"
```

### 3.9.2 Echoing an Executable File

By default, when you execute a MathScript file, the contents of the file itself are not echoed to the log area. If you specify **SET ECHO ON**, each statement is displayed to the log area as it is being executed. To turn this feature on, type:

```
set echo on
```

You can find out the current echo setting by typing:

```
show echo
```

To turn the echo off, type:

```
set echo off
```

### 3.9.3 *startup.ms* (on UNIX systems)

The environment variable **XMATH\_STARTUP** defines the properties of the Xmath startup icon to execute the **startup.ms** batch file. This batch file contains MathScript statements that execute every time you start a new Xmath session. You can set up your initial working environment for the Xmath session (for example, you can specify a list of directories as a search path).

Xmath looks for and executes **startup.ms** in the following order:

1. The space-separated list of directories specified in the environment variable **XMATH\_STARTUP**
2. The optional **xmath** subdirectory under your home directory (**\$HOME/xmath/startup.ms**)
3. The current directory (**./startup.ms**)

The environment variable **XMATH\_STARTUP** can be set to include multiple directories. For example:

```
setenv XMATH_STARTUP "/home/group /home/user"
```

Xmath will run **startup.ms** in **/home/group** and then **/home/user**. [Example 3-1](#) shows a sample **startup.ms** file.

#### Example 3-1 Sample startup.ms File

```
# set up aliases
alias sp set path =
# set path to several test directories
```

```
sp "/usr/me/tests"
sp "/usr/me/tests/routines"
# set up new partition and go there
new partition projectX
set partition projectX
# output data display format
set format long
```

### 3.9.4 *startup.ms (on Windows Systems)*

The environment variable **XMATH\_STARTUP** defines the properties of the Xmath startup icon to execute the **startup.ms** batch file. This batch file contains MathScript statements that execute every time you start a new Xmath session. You can set up your initial working environment for the Xmath session (for example, you can specify a list of directories as a search path).

The following are sample definitions for **%XMATH\_STARTUP%**.

Windows NT: Set the path to the **startup.ms** batch file by selecting Start→Settings→Control Panel→System. From the System Properties dialog Environment tab, for example, add an entry in the User Variables field (Variable, Value):

```
XMATH_STARTUP %HOME%\user
```

Windows 98 or Windows 95: Set the path to the **startup.ms** batch file by adding the following line to your **AUTOEXEC.BAT** file (or to any other startup batch file):

```
set XMATH_STARTUP=%HOME%\user
```

Xmath looks for and executes **startup.ms** in the following order:

1. The space-separated list of directories specified in the environment variable **XMATH\_STARTUP**
2. The optional **xmath** subdirectory under your home directory (**%HOME%\xmath\startup.ms**)
3. The current directory (**.\startup.ms**)




---

**NOTE:** You must define the **%HOME%** variable yourself.

---

You can set the environment variable **XMATH\_STARTUP** to include multiple directories. For example:

```
set XMATH_STARTUP="%HOME%\group %HOME%\user"
```

Xmath runs **startup.ms** in **%HOME%\group** and then **%HOME%\user**.  
[Example 3-2](#) shows a sample **startup.ms** file.

Example 3-2 **Sample startup.ms File**

```
# set up aliases
alias sp set path =
# set path to several test directories
sp "\\user\me\tests"
sp "\\user\me\tests\routines"
# set up new partition and go there
new partition projectX
set partition projectX
# output data display format
set format long
```

### 3.9.5 I/O Redirection

If you have a lengthy automated process that does not require interactive input, you can run it in background or batch mode using the tty (non-graphical) version of Xmath.

To create a MathScript file suitable for batch execution, start by using an editor to write a script file containing the instructions as you would enter them from the Xmath command line. Alternatively, you can start with a command diary file. Data generated in the batch script file can be written to an external file using the **SAVE** command.

If a file runs to completion and unsaved variables exist, Xmath asks the question:

```
Modified variables that have not been saved exist; quit anyway? (y/n)
```

This presents a problem because you cannot respond while in batch mode. To bypass the situation, you must **SAVE** or **delete** the data at the end of the file. The final entry in a batch file must be **QUIT**. (If **QUIT** does not end the file, Xmath will remain in terminal mode.)

#### I/O Redirection

To run the completed batch file from the UNIX command line, type:

```
% xmath -tty < batchfile.ms > batchfile.output
```



where the MathScript input is contained in **batchfile.ms**, and the output results are redirected to **batchfile.output**. (The output file contains anything that would normally appear in the Commands window log area, so be sure that **echo** is set properly.)

### 3.10 Recording an Xmath Session (Diaries)

Xmath can automatically record commands and responses using command and session diaries. A command diary records user input only, while a session diary records user input and the Xmath responses.

To create a diary, the environmental variable **echo** must be **on**. If it is **off**, a diary file may be opened but nothing will be recorded in it. To determine the **echo** setting, type:

```
show echo
```

If **echo** is **off**, you must type **set echo on** to activate it.

#### 3.10.1 Recording Inputs (Command Diary)

Command diaries record MathScript input. A command diary is by definition an executable file; it contains all valid instructions issued while the command diary was set. However, when the file is executed, you may not see all the outputs you did when you captured the commands; you must either edit the diary to insert the proper terminators, or be sure to use them when you input the commands you are capturing.

To open a command diary, type:

```
set echo on  
set commanddiary "mytest.ms"
```

where **mytest.ms** is the name of the diary file. The file is placed in the current working directory (see [3.6.1 Manipulate and Show Current Directory](#), p. 108 for details on setting the working directory). To see if a diary file is already open, type:

```
show commanddiary
```

If the specified file does not exist, it will be opened for writing. If a diary file of the same name exists, it will be closed and a new file opened.

```
t = 1:0.1:100
s = sin(t)
g=plot (s)?
```

To close a diary file, use the **remove** command:

```
remove commanddiary
```

Since a command diary contains only executable MathScript commands, you can replay the contents using **execute**:

```
execute file = "mytest"
```

Note the output behavior when the file is replayed. When the calls were typed interactively, the outputs of **t** and **s** were written to the log area, but when the file was executed the outputs were omitted. When a value is assigned to a variable, the function outputs will only be displayed if the question mark terminator (?) is used, as was the case for the graph object **g**.

### 3.10.2 Recording Inputs and Outputs (Session Diary)

A session diary records inputs and outputs, that appear in the Commands window log area while the diary is open. This can be useful when the contents of a data object need to be recorded in the file. For example:

```
set echo on

set sessiondiary "session1"

test1 = 0.75;
exist(test1)
sin(test1)

remove sessiondiary
oscmd("more session1")
```

Because session diaries include outputs (which are not MathScript statements), they cannot be executed as command diaries until they are edited.

# 4

## Graphics

This chapter begins with an outline of the plotting functions and commands available in Xmath. The remaining sections show how to graphically display your data with the `plot()` function, and also how to change its appearance interactively with the Xmath Graphics window.

### 4.1 Xmath Plotting Functions and Commands

#### 4.1.1 General Purpose Plotting Functions

Xmath provides a choice of three basic plotting functions:

- The `plot()` function provides an easy to learn syntax for 2d and 3d plotting in an interactive graphics window. For a quick, interactive look at your data, and for 3d plotting, `plot()` is a good choice.
- The `uiPlot()` function provides full featured 2d plotting integrated with an extensive programmable GUI facility. If you want more control over the formatting of your 2d graphics, or the ability to integrate plots with your own interactive Xmath PGUI tools, then `uiPlot()` has the power you need.
- The `plot2d()` function recognizes most `plot()` keyword options and provides quick access to advanced formatting features of the `uiPlot()` function, while avoiding the cost of constructing a programmable GUI tool. Use `plot2d()` to obtain highly-customized 2d graphics without writing a PGUI tool.

## **plot()**

The **plot()** function and its associated Xmath Graphics window provide complete interactive facilities for building, modifying, and viewing 2d and 3d graphics. You can specify graph characteristics as keywords to plot, or you can add or modify them interactively from the Xmath Graphics window menus or the Xmath Palette.

The output of **plot()** is a graphics object. Rather than archiving an executable file that recreates a graph, you can save the images as graph objects. A graph object can be displayed in the Xmath Graphics window, altered with keywords, or combined with another plot to create a new image.

**plot()** keyword options facilitate multiple plots, strip plots, bar plots, polar plots, contour plots, and scatter plots. The animation mode of the Xmath Graphics window provides rapid sequential display of graphics objects.

This chapter provides additional detail on the capabilities of **plot()**. For further information about **plot()** and associated plotting tools, see the MATRIX<sub>X</sub> online Help *Xmath, Plotting* topic.

## **uiPlot()**

The **uiPlot()** function formats and displays 2d plots (including line, scatter, and polygon) in any uiPlotArea widget of a programmable GUI tool. While **plot()** is limited to displaying its objects in a single Xmath Graphics window, **uiPlot()** can generate and display plots in multiple windows. However, this power comes at a considerable cost—the construction of programmable GUI tools and widgets.

**uiPlot()** features include interactive data-viewing, zooming, and curve selection. Animation is achieved through the binding of curves to Xmath variables. Custom callbacks can be programmed in GUI tools, providing application-specific, graphic interaction with the data.

The **uiPlot()** function syntax provides access to the structure of the underlying graphics database. The database hierarchy lets users specify graphics objects much like how one specifies a file path. Properties can then be set using either **uiPlot()** keywords, or generic option strings of the underlying graphics system, resulting in a wide range of custom formatting capabilities.

For more information about the programmable GUI, see [9. Graphical User Interface](#). For further details on using the **uiPlot()** function and associated plotting tools, see the MATRIX<sub>X</sub> online Help *Xmath, Plotting* topic.

**plot2d()**

The **plot2d()** function is based on **uiPlot()**, and is designed to implement most capabilities of both **plot()** and **uiPlot()**, while avoiding the overhead of programmable GUI tools and widgets.

In particular, **plot2d()** provides multiple graphics windows, interactive data-viewing, animation through the binding of curves to Xmath variables, and the power of the **uiPlot** function syntax. Some new features have been implemented such as multiple Y-axes, advanced row/column layout options, and automatic data scaling in one coordinate while constraints are specified in the other.

For those familiar with **plot()** syntax, **plot2d()** supports most of the 2d-related keywords of **plot()**. It is possible to convert most scripts by substituting **plot()** function calls with identical **plot2d()** calls.

The most obvious differences between **plot2d()** and **plot()** are that 3d plotting options and the graphics object are not supported. All **uiPlot()** functionality is available through the **plot2d()** function.

For further details on using the **plot2d()** function and associated plotting tools, see the **MATRIX<sub>X</sub>** online Help *Xmath, Plotting* topic.

### 4.1.2 Comparative Analysis: **plot()** versus **plot2d()**

Table 4-1 **plot()** Advantages

<b>3D Plotting</b>	<b>plot()</b> supports 3d lines and surfaces, and also 2d and 3d contour plots. The <b>plot2d()</b> function does not support 3d or contour plots.
<b>Polar Plots</b>	<b>plot()</b> supports polar coordinate plotting. To display polar plots, <b>plot2d()</b> users must write their own conversion script.
<b>Graph Object</b>	The graphic result of a plot function can be saved to a variable. <b>plot2d()</b> does not support graph objects.

Table 4-2 **plot2d( ) Advantages (these features also available with uiPlot)**

---

<b>Data Viewing</b>	<b>plot2d( )</b> has the capability of displaying (X,Y) data values of curves in a pop-up window interactively activated by the Right Mouse Button (RMB).
<b>Callbacks</b>	<b>plot2d( )</b> callbacks can be attached to curves/sub-areas for click/drag/release with modifiers. The callbacks are implemented using MathScript.
<b>Variable Binding</b>	By binding a variable to a plotted curve, the <b>plot2d( )</b> plots are updated as the variable changes. This is superior to the <b>plot( )</b> method of achieving animated display.
<b>Polygons</b>	<b>plot2d( )</b> has polygon plot capability. <b>plot( )</b> does not support 2d polygon plots.
<b>Hierarchy Selection</b>	By specifying hierarchy paths, selected elements of <b>plot2d( )</b> plots can be addressed for setting/changing attributes.
<b>Plot Inclusion</b>	A <b>plot2d( )</b> sub-area can be made to include a plot from some other window. Useful for displaying a single post-stamp sized element of a large row/column plot. Any changes are updated automatically.
<b>Drivers</b>	Setting generic Hoops options such as driver options, heuristics, visibility and frame are available in <b>plot2d( )</b> .

---

### 4.1.3 Plotting Commands and Special Purpose Functions:

Several additional commands and functions are used with the general purpose Xmath plotting functions. Brief descriptions are given here. Some are discussed in more detail later in this chapter. For more information, see the MATRIX<sub>x</sub> online Help *Xmath, Plotting* topic.

#### **colorind**

The **colorind** function creates a colorindex matrix used as a fourth argument with the plot function to add color emphasis or a fourth dimension to 3d plots.

**ERASE**

The ERASE command can be used to erase the contents of the Xmath Graphics window (plot function display).

**HARDCOPY**

The HARDCOPY command is used to create a hardcopy of the contents of the Xmath Graphics window (plot function display), or a graphics object (plot function display). It can also be used to create hardcopy of plot2d results.

***pdmplot***

The pdmplot function invokes a dialog driven process resulting in plots selected from a specified pdm. It can be used with either the plot or the uiPlot plotting system.

***qplot***

qplot is a simple uiPlot based function. Like plot2d, it provides use of uiPlot features with a pre-programmed GUI. However, qplot does not support plot and other high-level keyword capabilities of plot2d.

***uiPlotArea***

uiPlotArea is a programmable GUI function for creating uiPlotArea widgets.

***uiPlotGet***

uiPlotGet is a programmable GUI function for getting the current cursor position to be used with callback routines.

## 4.2 Using the `plot()` Function

The `plot()` function creates 2D and 3D plots from data; complex components (those containing imaginary elements) are ignored.<sup>1</sup> You can call `plot()` with any one of the following syntaxes:

```
graphObj = plot(y, {keywords})
graphObj = plot(x, y, {keywords})
graphObj = plot(x, y, z, {keywords})
graphObj = plot(x, y, z, colorindex, {keywords})
graphObj = plot()
graphObj = plot({keywords})
graphObj = plot(graphObj, {keywords})
```

In the preceding plot syntaxes, `x` is a vector or matrix; `y` is a vector, matrix, or PDM; and `z` is a vector or matrix. If `z` is a matrix, a color index matrix `colorindex` can be supplied to add color as a fourth dimension. Each syntax is discussed in the following sections.

An existing graph object can be reused as an input in several ways; it can be altered with keywords or combined with another plot to create a new image.

An optional graph object can be included as an input (for one, two, or three input plots). If the data is compatible, the new data is overlaid on `graphObj`, and the modified graph is returned as a graph object from `plot()`. However, a graph object can also be referenced with the `keep` or `copy` keywords. The `keep` keyword is preferable because it is fastest. In either case, you can reference a single graph object. (You can't specify `keep` and the optional graph object input in the same call.)

If you input the data below, you can test each syntax in the sections that follow:

```
# define vectors for plotting

v=[0:.25:30]';
vc=v.*cos(v); vs=v.*sin(v);

# define a PDM

ypdm=pdm([vc,vs]);

# define matrices for plotting
x=[vc,vc]; y=[vs,vs]; z=[1.5*v,1.5*v];
vm=vs*vc';
m=v*v';
```

- 
1. If you need to plot complex data, you can make a real vs. imaginary cartesian graph. Given complex data `z`, call `plot(real(z),imag(z))`.



```
ms=[vs,-vs];
mc=[vc,-vc];
```

### 4.2.1 Plot One Input

For a single argument the syntax is **plot(y)**:

- If **y** is a vector with  $m$  elements, then **y** is plotted versus the vector  $1:m$ .

```
plot(vc)?
```

- If **y** is an  $m \times n$  matrix, then each column of **y** is plotted versus the vector  $1:m$ . The result is  $n$  curves, each with  $m$  points.

```
plot(vm)?
```

- If **y** is an  $m \times n \times d$  PDM where  $m \times n$  is the size of each dependent matrix, and  $d$  is the length of the domain (the independent parameter), then  $m \times n$  curves of  $d$  points are drawn, each versus **domain(y)**. Therefore, each line corresponds to a channel of a PDM (see [5.4.4 PDM Channels](#), p.194):

```
plot(pdm([vc,vs]))?
```

### 4.2.2 Plot Two Inputs

The syntax for two arguments is **plot(x,y)**:

- If **x** and **y** are vectors of the same length, then **y** is plotted against **x**:

```
plot(vs,vc)?
```

- If **x** is an  $m \times 1$  or  $1 \times m$  vector and **y** is an  $m \times n$  matrix, each of the  $n$  columns of **y** is plotted against **x** on a single graph. Each curve has  $m$  points:

```
plot (vs,m(:,1:7:length(vs)))?
```

- If **x** and **y** are both  $m \times n$  matrices, then  $n$  curves are drawn, each consisting of a column of **y** versus the corresponding column of **x**:

```
plot (m,vm)?
```

### 4.2.3 Plot Three Inputs

The syntax for three arguments is **plot(x,y,z)**:

- If  $x$ ,  $y$ , and  $z$  are vectors of the same length, then  $z$  is plotted versus  $x$  and  $y$  as a curve in space:

```
plot (vc,vs,v/3)?
```

- If  $x$  is an  $m \times 1$  or  $1 \times m$  vector, and  $y$  is  $n \times 1$  or  $1 \times n$ , and  $z$  is an  $n \times m$  matrix, then  $z$  is plotted as a surface versus  $x$  and  $y$ :

```
plot (vc(1:50),vs(1:50),vm(61:110,61:110),{!grid})?
```

- If  $x$ ,  $y$ , and  $z$  are matrices of the same dimensions, then  $z$  is plotted versus  $x$  and  $y$  as a surface in space:

```
plot (mc,ms,z,{!grid})?
```

#### 4.2.4 Color as a Fourth Dimension

If inputs  $x$ ,  $y$ , and  $z$  are supplied and  $z$  is a matrix, then you can pass a fourth argument to use color to represent an additional dimension over the data surface. In the following example, the fourth argument is a matrix the same size as  $z$  generated by the `colorind()` function (a colorindex matrix). The values specified with the `face_color` keyword are applied to the data surface at the locations in the colorindex matrix:

```
v=[0:.25:30]';  
x=v.*sin(v);  
Y=x;  
z=vs*-vs';  
z=z(31:60,31:60);  
g1=plot(x(1:30),y(1:30),z,colorind(z),{face_color=9:19})
```

#### 4.2.5 Creating and Displaying a Graph Object

This section discusses common plotting approaches. Keywords mentioned here are discussed in detail later in this chapter.

Graph object output is handled like any other function output, except that it is displayed in the Xmath Graphics window rather than to the log area. When no output is assigned, the graph is written to the default object `ans`.

It is a good practice to use the `?` terminator with `plot()`, regardless of how you call it: interactively, in executable files, or in MathScript entities. This is particularly important when plots are developed in a `.ms` file. (By default, `set echo` is `off` when files are executed so Xmath displays only graphs with the `?` terminator.)

The **keep** keyword (which is also discussed in [4.3.5 Adding New Data to Existing Plots \(keep, copy\)](#), p. 145) combines an existing graph object and any new information. If the plots are compatible, the new information becomes part of the specified graph. For example:

```
v=[0:.05:5];
plot(v.*sin(v),{title="The first curve."})?
plot(-v.*cos(v),{keep,title="The second curve."})?
```

The second curve is plotted over the first; note that **plot()** recognized there was already a title and substituted the newest one. You can still add to the graph, and this time name the output:

```
final=plot({ keep, xlog, xmax=100, title="The Final Graph",
legend=["1st curve","2nd curve"]})?
```

If saving a graph to a variable is an afterthought, you can capture the current image in the Xmath Graphics window by selecting File→Bind to variable from the Xmath Graphics window menu bar or by calling **plot()** with no arguments:

```
g2=plot()           # name current graph object g2
```

Both File→Bind to Variable and **Variable=plot()** do the same thing as **Variable=plot({keep})**.

Once a graph object is assigned to a variable, it can be **saved** to a file and then **loaded** and displayed at a later time. Rather than creating an executable file that recreates the graph, you can archive the images themselves.




---

**NOTE:** If you have an observable delay, when you drag a window across an Xmath Graphics window, try disabling the Show window contents while dragging checkbox on the Control Panel→Display→Plus! tab (Windows NT only).

---

### 4.3 Using Keywords with plot

Every call to **plot()** can have a list of keywords that modify the plot's appearance. Almost everything that can be done using keywords can be done interactively with tools available from the Xmath Graphics window menus and the Xmath Palette. Keywords, however, are very convenient because they provide command-line control of graphics modifications. This implies that plot instructions can be saved

to and retrieved from a diary file or built up independently in a MathScript file. Also, a keyword string may be aliased to a shorter string.

- Plot keywords, as shown in [Table 4-3](#), are used like any other keywords. As a reminder, though, the type of information dictates how the keyword is implemented.

Table 4-3 **Keyword Types**

Keyword Type	Input Samples
Boolean	{!legend}, {!axisfix}
integer	{rows=5}, {line_color=12}
vector	{scale=[.5,.5],line_color=2:24}
string	{title="My Beautiful Graph"}
vector of strings	{legend=["input 1","input2"]}

For Boolean scalars, note that a nonzero value denotes TRUE/on, while 0 denotes FALSE/off. For example:

```
plot({grid,marker}) # grid and marker are on
plot({!grid,!x_lab}) # grid and x_lab are off
```

- If you use the **hold** keyword, the keyword settings remain until you redefine an attribute, until you use **!hold**, or until you call **plot({reset})** (see [Hold Keyword](#) on p.161).
- You can use the negative operator **!** to set a keyword to FALSE or 0. For example, you can use either **!grid** or **grid=0** to turn off *all* grid marks while **grid=1** enables them.

Sections [4.3.1](#) through [4.3.18](#) discuss keywords in functional groups (see [Table 4-4](#)) using examples to illustrate how they work. Each keyword description gives its default setting.

Table 4-4 **Keyword Categories**

Category	Section
Labels and legend	<a href="#">4.3.1</a>
Colors	<a href="#">4.3.2</a>
Line and marker specifications for data	<a href="#">4.3.3</a>

Table 4-4 **Keyword Categories** (Continued)

Category	Section
Multiple graphs and graph positioning	4.3.4
Adding new data to existing plots ( <b>keep</b> , <b>copy</b> )	4.3.5
Axis and zero lines	4.3.6
Tics and grids	4.3.7
Free text and global text settings	4.3.8
Axis limits and logarithmic scaling	4.3.9
Animate	4.3.10
Placement, scaling, and rotation	4.3.11
Background, edge, and face settings	4.3.12
Lighting source settings	4.3.13
Holding graph attributes	4.3.14
Strip plots	4.3.15
Bar plots	4.3.16
Contour plots	4.3.17
Polar plots	4.3.18

An alphabetized list of all keywords and the location of each appears in [Table 4-5](#).

Table 4-5 **Plot Keywords (Alphabetized Listing)**

Keyword	Page	Keyword	Page	Keyword	Page
<b>animate</b>	155	<b>line_width</b>	140	<b>x_axis_fix</b>	148
<b>axis</b>	135	<b>log</b>	154	<b>x_axis_line</b>	148
<b>axis_fix</b>	148	<b>marker</b>	141	<b>x_grid</b>	150
<b>axis_line<sup>a</sup></b>	148	<b>marker_color</b>	141	<b>x_inc</b>	154
<b>bar</b>	167	<b>marker_size</b>	142	<b>x_lab</b>	135

Table 4-5 **Plot Keywords (Alphabetized Listing)** *(Continued)*

<b>Keyword</b>	<b>Page</b>	<b>Keyword</b>	<b>Page</b>	<b>Keyword</b>	<b>Page</b>
bg_color	158	marker_style	142	x_log	154
colormap	138	move	157	x_max	154
column	143	polar	170	x_min	154
columns	143	position	157	x_tic	149
contour	168	projection	156	x_tic_lab	150
contour2d	168	r_inc	170	x_zero_line	148
contour3d	168	r_max	170	y_axis	148
contour_interval	168	reset	161	y_axis_fix	148
copy	146	rotate	156	y_axis_line	148
date	135	row	144	y_grid	150
edge	159	rows	144	y_inc	150
edge_color	159	scale	156	y_lab	135
edge_style	159	strip	165	y_log	154
edge_width	159	text	152	y_max	154
face	158	text_angle	152	y_min	154
face_color	159	text_color	152	y_tic	149
face_style	159	text_font	152	y_tic_lab	150
fg_color	146	text_position	152	y_zero_line	148
graph_number	159	text_style	152	z_axis	148
grid	150	text_size	152	z_axis_fix	148
hold	161	theta_inc	170	z_axis_line	148
keep	146	theta_max	170	z_grid	150
keepssubplot	146	theta_min	170	z_inc	154
legend	136	tic	149	z_lab	135

Table 4-5 Plot Keywords (Alphabetized Listing) (Continued)

Keyword	Page	Keyword	Page	Keyword	Page
light	160	tic_lab	150	z_log	154
light_color	160	tic_maj	149	z_max	154
light_direction	160	tic_min	149	z_min	154
line	140	time	136	z_tic	149
line_color	140	title	135	z_tic_lab	150
line_style	140	x_axis	148	z_zero_line	148

a Underscores are always optional. For example, both `x_axis` and `axis` are acceptable.

### 4.3.1 Labels and Legend

Labels allow you to place a text string in a specific location relative to the plotted data. Labels are therefore bound to the plot and their locations cannot be changed.

The keywords `legend`, `date`, and `time` also place text on the graph, but you *can* move these small text objects with the mouse. (To create “independent” text, use the `text` keywords on [p. 152](#), or create free text interactively.) [Table 4-6](#) summarizes the labels and legends.

Table 4-6 Label and Legend Keywords

Keywords	Description
<code>title</code>	String for the graph title above the plot. Default is an empty string.
<code>x_lab</code>	String for the x-axis label. Default is an empty string.
<code>y_lab</code>	String for the y-axis label. Default is an empty string.
<code>z_lab</code>	String for the z-axis label. Default is an empty string.
<code>date</code>	Places the date in the upper left corner; format is: <code>dayName_month_date_year</code> . Default is an empty string.

Table 4-6 **Label and Legend Keywords** (Continued)

Keywords	Description
<b>time</b>	Places the time in the upper left corner. Format is <i>hour_minutes_seconds</i> on a 24 hour clock. Default is an empty string.
<b>legend</b>	For multi-line or contour plots, you can specify a vector of strings naming each line or contour, for example, <b>legend =["Time","Speed"]</b> . The default labels for 2D plots are the line number followed by the corresponding line style (and color, for color monitors). For 3D plots, the default legend corresponds to differing surface styles.

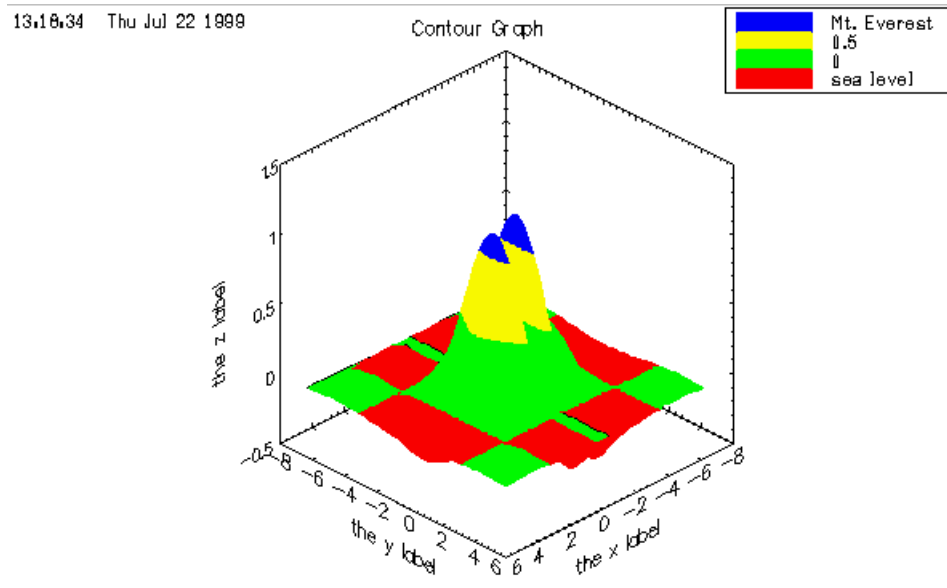
Tic labels (numbers corresponding to major tic marks) are discussed on [p.149](#).

The example that follows creates 3D data and then creates the contour graph shown in [Figure 4-1](#). All axis information is negated so that you can clearly see every label (negating axis information is optional). Note the string of vectors used to label the legend. There are four intervals in this contour, and this vector of strings provides new labels for only the first and last; the default label is displayed for intervals where the null string "" is specified.

```
x=[-2*pi:0.35:2*pi]';
x=[x;x];y=x;
z=sin(round(x))./x*(sin(y)./y)';
legetext=["Mt. Everest","", "", "sea level"];
g=plot(x,y,z,{!grid,contour3d, time, date,
    title="Contour Graph", xlabel="the x label",
    ylabel="the y label", zlab="the z label",
    legend=legetext})?
```



Figure 4-1 Label Locations and Legend



### 4.3.2 Colors

Many keywords take a color as an argument. You can specify colors by number or name, and a vector of color names or numbers is acceptable. You can see the current colormap on the Xmath Palette. On color monitors, up to 64 colors can be allocated.

If a value is specified (an integer between 1 and 64), Xmath indexes into the current colormap.

If a color name is specified, Xmath searches for a match in the following tables in the order shown:

1. The currently installed Xmath colormap.

For black and white systems, the current colormap simply represents black, six shades of gray, and white. On color systems, each row in the colormap is a color; the first column represents red intensity; the second, green intensity; and the third, blue intensity.

2. On UNIX systems, the X11 color name database (often stored in `/usr/lib/X11/rgb.txt`)

This is a very long list.

- The list of supported Xmath color names (Table 4-7).

The first eight colors on this list compose the default sequence for line and marker colors. The first color is black or white (depending on the background color), followed by red, green, yellow, blue, magenta, cyan, and black or white.

If you use strings to specify these colors, spacing must be typed as shown, but case is not important. For example:

```
plot(x, {bg_color="CADET BLUE",fg_color=51})
```

- The list of default (built-in), machine-dependent color names.

As soon as a name match is found in one of the locations above, Xmath looks at the corresponding values, compares them to values in the current colormap, and then implements the closest color available in the current colormap.

To supply your own colormap, construct an  $n \times 3$  matrix with values representing red, green, and blue intensity ranging from 0 to 1. Before installing your color map, it's a good idea to save the default color map:

```
mapDefault=plot({colormap})
```

This saves the colormap to the variable **mapDefault**.

To replace the current colormap with your own **mapMyColors**, type:

```
plot({colormap=mapMyColors})
```

Your colormap now appears in the Xmath Palette as the current colormap.

For more on colormaps, see the MATRIX<sub>X</sub> online Help listings for **Color List**, **Colormaps**, and **Color**.

Table 4-7 **String Color Names for Xmath Supported Colors**

No.	Name	No.	Name	No.	Name
1	"black"	22	"chris cyan"	43	"aliki aqua"
2	"red"	23	"periwinkle"	44	"cyan"
3	"green"	24	"prussian blue"	45	"cerulean"
4	"yellow"	25	"cadet blue"	46	"big blue"

Table 4-7 **String Color Names for Xmath Supported Colors** (Continued)

No.	Name	No.	Name	No.	Name
5	"blue"	26	"kam blue"	47	"lapis"
6	"magenta"	27	"royal purple"	48	"blue"
7	"cyan"	28	"red violet"	49	"marine blue"
8	"white"	29	"mulberry"	50	"violet"
9	"silly putty"	30	"orchid"	51	"mark magenta"
10	"peach"	31	"maroon"	52	"purple"
11	"salmon"	32	"strawberry"	53	"fuchsia"
12	"brick"	33	"fire engine red"	54	"berry"
13	"kin orange"	34	"orange"	55	"raspberry ron"
14	"burnt umber"	35	"pumpkin"	56	"red"
15	"brown"	36	"golden dawn"	57	"black"
16	"coffee"	37	"yellow"	58	"gray5"
17	"mustard"	38	"lemon yellow"	59	"gray4"
18	"neon green"	39	"light green"	60	"gray3"
19	"forest green"	40	"algae"	61	"gray2"
20	"teal"	41	"grant green"	62	"gray1"
21	"ocean green"	42	"new grass"	63	"gray0"
				64	"white"

The following keywords dictate color changes for different plot elements:

- **bg\_color**
- **edge\_color**
- **face\_color**
- **fg\_color**
- **grid\_color**
- **light\_color**

- `line_color`
- `marker_color`
- `text_color`

The meanings of these keywords are discussed elsewhere within the keyword functional groups.

### 4.3.3 Line and Marker Specifications for Data

You can change the color, style, and width for lines (for example, curves) of data as specified in [Table 4-8](#). If you make changes to lines and specify the **legend** keyword, your changes are reflected in the legend.

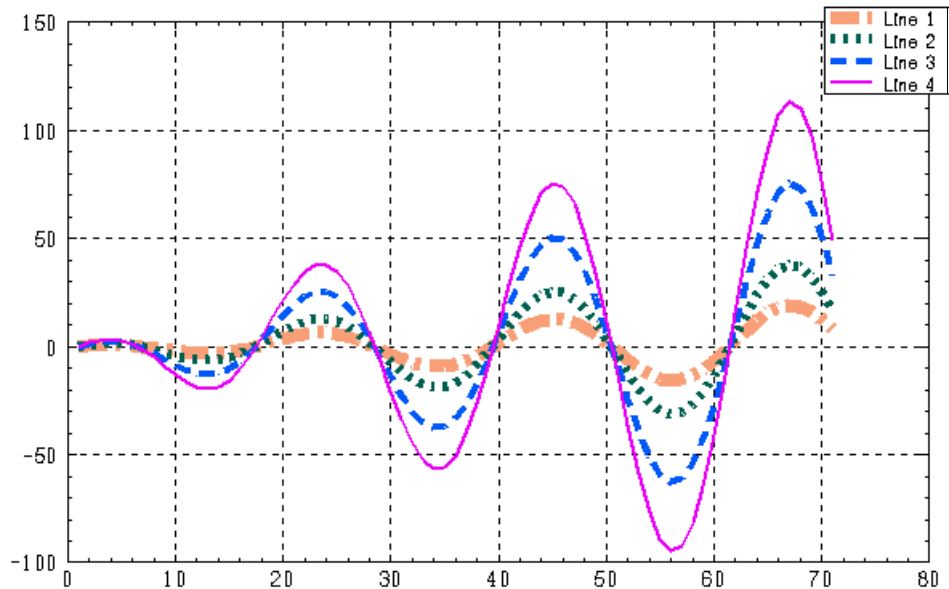
Table 4-8 **Line Specification Keywords**

Keyword	Description																		
<code>line</code>	Boolean that turns line plotting on or off. Default=1.																		
<code>line_color</code>	Integer, string, vector of integers, or vector of strings for specifying data line colors (see <a href="#">4.3.2 Colors</a> , p. 137). If <code>line_color</code> specifies a vector, the given color sequence is cycled through. On color monitors for plots with multiple curves, Xmath automatically assigns each curve a different color.																		
<code>line_width</code>	Any float is accepted. The variety of line widths allowed is machine dependent; if you specify a value the machine can't provide, it supplies the closest thing. The default value of 1 is approximately equal to 1 pixel on your monitor. On a high resolution monitor, the difference between .5 and 1 may be visible; on others the output might be the same.																		
<code>line_style</code>	Integer, vector of integers, string, or vector of strings that specify line styles for each curve on the graph. The <b>line_style</b> mapping is: <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Integer</th> <th>String</th> </tr> </thead> <tbody> <tr><td>0</td><td>" "</td></tr> <tr><td>1</td><td>"_ _ _"</td></tr> <tr><td>2</td><td>"_ - "</td></tr> <tr><td>3</td><td>"_ . ."</td></tr> <tr><td>4</td><td>"_ - ."</td></tr> <tr><td>5</td><td>"_ . ."</td></tr> <tr><td>6</td><td>"_ . ."</td></tr> <tr><td>7</td><td>"_ - _"</td></tr> </tbody> </table>	Integer	String	0	" "	1	"_ _ _"	2	"_ - "	3	"_ . ."	4	"_ - ."	5	"_ . ."	6	"_ . ."	7	"_ - _"
Integer	String																		
0	" "																		
1	"_ _ _"																		
2	"_ - "																		
3	"_ . ."																		
4	"_ - ."																		
5	"_ . ."																		
6	"_ . ."																		
7	"_ - _"																		
	If <code>line_style</code> is set to a vector of integers, strings, or names, Xmath cycles through the specified sequence of styles.																		

The following example generates several line styles and widths; the plot appears in [Figure 4-2](#):

```
v=[0:2/7:20]';vc=v.*cos(v);
x=[vc,vc*2,vc*4,vc*6];
plot(x,{legend, line_width=[8,6,4,2],line_style=[4,3,2,1],
      line_color=["peach","teal","lapis","purple"]})
```

Figure 4-2 **Line Styles and Widths**



Markers, as described in [Table 4-9](#) are symbols plotted at each data point. You can change a marker's size, style, or color using integers, floats, or strings the same as you do with line styles. To see a plot with only markers, use the keywords `{!line,marker}`.

Table 4-9 **Marker Specification Keywords**

Keyword	Description
<b>marker</b>	Boolean that turns on/off plot markers. Default=0.
<b>marker_color</b>	Integer, vector of integers, string, or vector of strings that specifies marker color (see <a href="#">4.3.2 Colors</a> , p.137). You can specify an integer or string for each curve on a graph. If a vector is specified, the color sequence is cycled through.

Table 4-9 **Marker Specification Keywords** (Continued)

Keyword	Description																																	
<b>marker_size</b>	Any nonzero float is accepted. The range of marker sizes allowed is machine-dependent; if you specify a value the machine can't provide it will supply the closest thing. The default value is 0.5.																																	
<b>marker_style</b>	Integer, vector of integers, string, or vector of strings that specifies marker style for each curve on the graph. The marker style mapping is: <table border="1" data-bbox="525 539 1039 864"> <thead> <tr> <th>Integer</th> <th>String</th> <th>Style</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>" "</td> <td>no markers</td> </tr> <tr> <td>1</td> <td>"*"</td> <td>asterisks</td> </tr> <tr> <td>2</td> <td>"x"</td> <td>x's</td> </tr> <tr> <td>3</td> <td>"+"</td> <td>crosses</td> </tr> <tr> <td>4</td> <td>"o"</td> <td>circles</td> </tr> <tr> <td>5</td> <td>"(*)"</td> <td>filled circles</td> </tr> <tr> <td>6</td> <td>"["</td> <td>squares</td> </tr> <tr> <td>7</td> <td>"[*]"</td> <td>filled squares</td> </tr> <tr> <td>8</td> <td>"/\\"</td> <td>triangles</td> </tr> <tr> <td>9</td> <td>"/*\\"</td> <td>filled triangles</td> </tr> </tbody> </table> <p>The default marker style is 1. If a vector of marker styles is specified, they will be cycled through.</p>	Integer	String	Style	0	" "	no markers	1	"*"	asterisks	2	"x"	x's	3	"+"	crosses	4	"o"	circles	5	"(*)"	filled circles	6	"["	squares	7	"[*]"	filled squares	8	"/\\"	triangles	9	"/*\\"	filled triangles
Integer	String	Style																																
0	" "	no markers																																
1	"*"	asterisks																																
2	"x"	x's																																
3	"+"	crosses																																
4	"o"	circles																																
5	"(*)"	filled circles																																
6	"["	squares																																
7	"[*]"	filled squares																																
8	"/\\"	triangles																																
9	"/*\\"	filled triangles																																

You can use a combination of line styles and markers to expand the number of unique lines you can plot. This is especially valuable for those using black-and-white monitors or for complicated plots that will be printed in black and white.

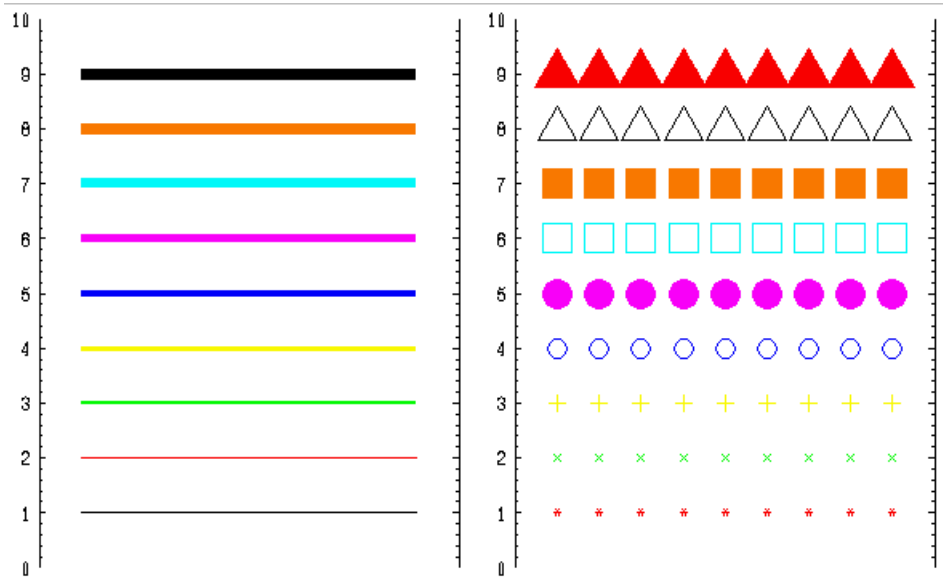
```
a=1:9; b=ones(9,9);for i=1:9; b(i,:)=a;endfor
plot(!grid,!x_axis,y_inc=1,axisfix,hold)

plot(b,{columns=2,
       line_width=[.5, 1, 2, 3.5, 4, 5.5, 6.5, 7, 7.5]}?)

g=plot(b,{keep,column=2,!line,marker_size=[.25, .5,
      .75, 1, 1.5, 2.25, 2.25, 2.5, 2.75]})
plot({reset})
```

The final result, [Figure 4-3](#), shows some of the line and marker styles in a variety of widths and sizes. Normally **hold** and **axisfix** need to be turned off with **!hold** and **!axisfix**, but **plot({reset})**, which resets everything, is used in this example.

Figure 4-3 Line and Marker Styles with Varying Widths and Sizes



### 4.3.4 Multiple Graphs and Graph Positioning

The keywords shown in [Table 4-10](#) allow you to place more than one plot in the Xmath Graphics window. If you are displaying multiple graphs, you can ensure that they are the same size by dividing the window into rows and/or columns and then positioning the graphs with row and column coordinates or graph number. You cannot rotate or zoom plots with multiple graphs interactively.

Table 4-10 Graph Specification Keyword

Keyword	Description
<b>column</b>	Integer specifying the column position of the graph. Default=1.
<b>columns</b>	Integer specifying how many columns the plot window is divided into. Default=1.
<b>row</b>	Integer specifying the row position of the graph. Default=1.

Table 4-10 **Graph Specification Keyword** (Continued)

Keyword	Description
<b>rows</b>	Integer specifying how many rows the plot window will be divided into. Default=1.
<b>graph_number</b>	Integer specifying alternate representation for row and column in a multi-graph plot. For <b>rows=m</b> and <b>columns=n</b> , the “cells” are numbered from 1 to <b>m * n</b> going across the rows and then down the columns. Thus, for <b>rows=2</b> and <b>columns=2</b> , <b>graph_number=3</b> is equivalent to <b>row=2, column=1</b> .

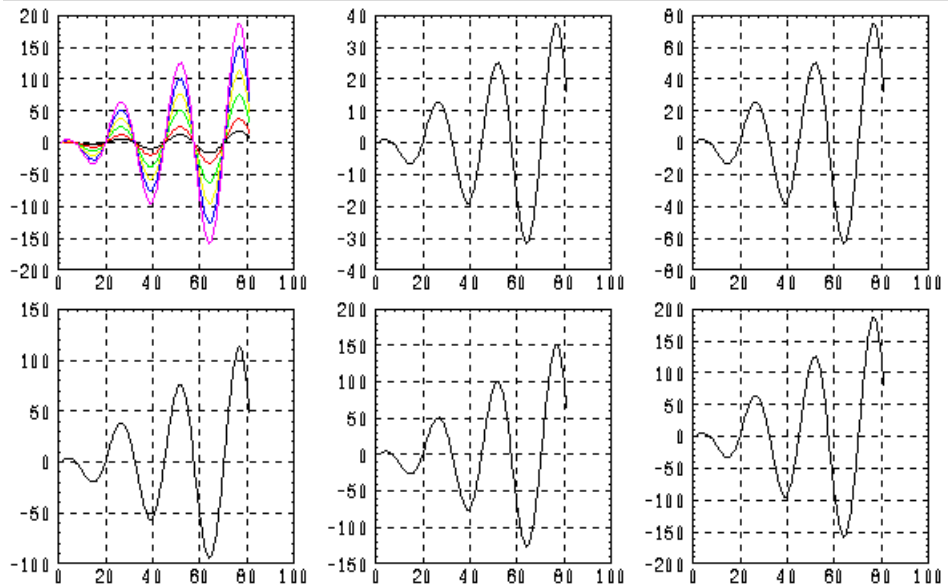
Note that the keywords **row**, **rows**, **column**, and **columns** all default to 1. Therefore, you needn't specify **row=1** or **column=1** because Xmath attempts to place graphs in these locations by default. The keywords **rows** and **columns** are *initiators*. If they are used in a **plot()** call, the row/column setting remains in effect for subsequent plots that use the keywords **row**, **column**, or **graph\_number**. If a plot is called that does not contain **row**, **column**, or **graph\_number**, the default format (**{rows=1,columns=1}**) is reset.

The following example places six graphs in the window; the final plot appears in [Figure 4-4](#).

```
v=[0:.25:20]';
vc=v.*cos(v);
x=[vc,vc*2,vc*4,vc*6,vc*8,vc*10];
g=plot (x,{rows=2,columns=3})           #assume row 1 col 1
g=plot (vc*2,{keep=g,column=2});       #assume column=1
g=plot (vc*4,{keep=g,column=3});       #assume row=1
g=plot (vc*6,{keep=g,graph_number=4});
g=plot (vc*8,{keep=g,graph_number=5});
g=plot (vc*10,{keep=g,graph_number=6})?
```



Figure 4-4 Plots Placed with row, column, and graph\_number Keywords



### 4.3.5 Adding New Data to Existing Plots (*keep*, *copy*)

Xmath has two ways of storing the image in the Xmath Graphics window in a variable. The keywords **keep** and **copy** described in Table 4-11 both use the contents of the Xmath Graphics window, but they may affect previously saved variables differently.

**Keep** combines the attributes and data from your current plot call with the current contents of the Xmath Graphics window and updates the variable. **Keep** is best used when you are building a plot by overlaying data or adding attributes to an existing plot. Because **keep** uses whatever is in the Xmath Graphics window, Xmath keeps changes you make with interactive tools automatically.

If you create a graph object **g1** and later create a graph object **g3** that keeps **g1**, a common incorrect perception is that **g1** has the old view and **g3**, the new. In reality, both variables point to the same graph object. You can test this as follows:

```
v=[1:.25:30]';vs=abs(v.*sin(v));vm=vs*vs';
g1=plot(vs,v)
```

```
g2=plot(vs(1:30),vs(1:30),vm(61:90),61:90)
```

```
g3=plot({keep=g1,log})
```

```
g1
```

where **g1** and **g3** are the same.

As long as the data dimensions allow it, Xmath performs any **keep** you specify. For example, you can combine a 2D and 3D plot. The following example uses the **keep** keyword to specify a 2D plot and provides the 3D information internally:

```
plot(vs(1:30),vs(1:30),vm(61:90,61:90),{!grid,keep=g3,
      ylab="The Y label",xlab="The X label",
      zlab="The Z Label"})
```

If you want to re-use a graph object but you don't want it to be altered, use the keyword **copy** instead of **keep** (see [Table 4-11](#) for these keyword descriptions). For example, **g1** and **g2** remain different in this case:

```
g2=plot({copy=g1,legend})
```

```
g1
```

Copying is computationally expensive, but it means you can save each stage when building a plot.

Table 4-11 **Data Keywords**

Keyword	Description
<b>keep</b>	Specifies that the current plot should be added to the specified graph object. If no graph object is specified, the plot is combined with the current contents of the Xmath Graphics window. If the graphs are incompatible, the new plot overwrites the Xmath Graphics window.
<b>keepsubplot</b>	Boolean. Used with <b>keep</b> when adding or replacing data on a subplot. <b>keepsubplot</b> indicates that new data should be laid over the existing data on the subplot. <b>!keepsubplot</b> indicates that the subplot should contain only the new data. Default is 1.
<b>copy</b>	<b>copy</b> can be specified with a graph object argument <b>{copy = GraphObj}</b> ; if no graph object is specified, the plot is combined with the current contents of the Xmath Graphics window. <b>copy</b> differs from <b>keep</b> in that <b>copy</b> does not alter the original graph object. In this case, the new data and graph keywords are combined with a copy of the existing graph object. The combined graph object is returned, while the copied graph object remains unchanged.

Figure 4-5 shows an example created by combining graph objects through the sequence of inputs below. By default, if graph objects with different data ranges are combined, Xmath rescales the plot to accommodate all the data. As you create each plot below, notice how the axes change to accommodate the new data with each curve addition.

```
v=[0:.25:20]; vc=v.*cos(v);vs=v.*sin(v);
plot({title="You can add to a graph as you work!"})?

plot(vc,{keep})?

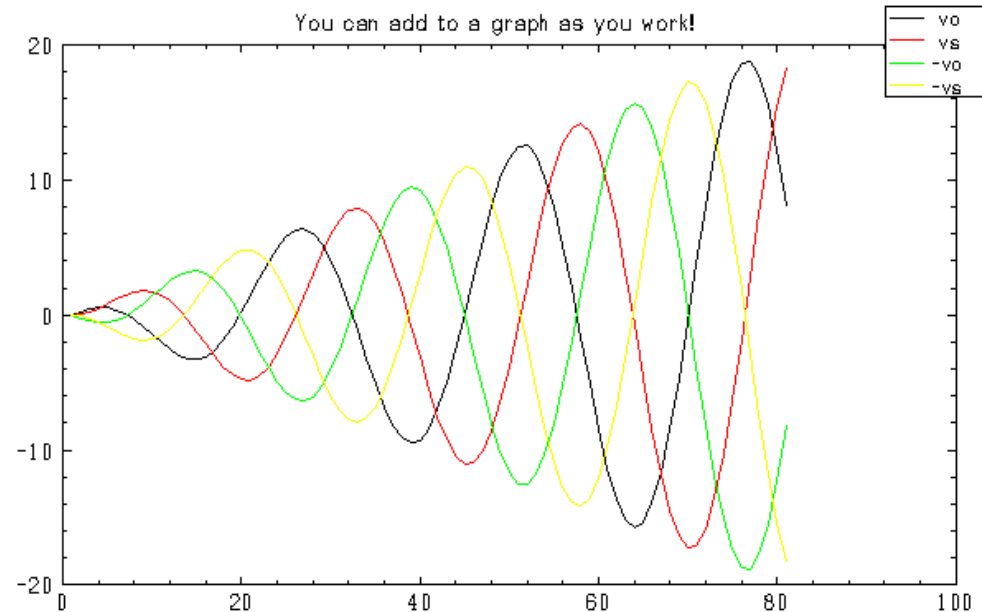
plot(vs,{keep})?

plot(-vc,{keep})?

plot(-vs,{keep})?

g=plot({keep,!grid,legend=[" vc"," vs","-vc","-vs"]})?
```

Figure 4-5 Combination of Graph Objects



If you do not want the plots rescaled, you must specify one of the `axisfix` keywords (see [Table 4-12](#)).

### 4.3.6 Axis and Zero Lines

The keywords described in [Table 4-12](#) control axis and zero-line display.

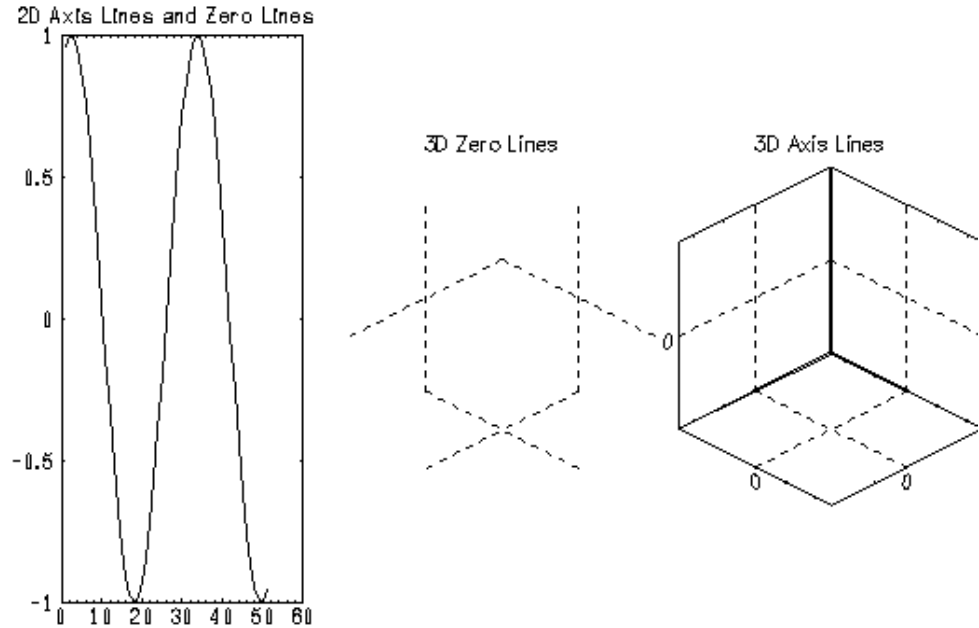
Table 4-12 **Axis and Zero Line Keywords**

Keyword	Description
<b>axis</b>	Boolean that turns on or off <i>all</i> axis graphics on the entire graph. This includes grids, zero lines, tic marks, and tic labels. If an attribute is specified, it is applied to all axis graphics.
<b>x_axis</b> <b>y_axis</b> <b>z_axis</b>	Booleans that toggle <i>all</i> axis graphics on the x, y, or z axis. Axis graphics color, style, and width attributes affect all components on the named axis.
<b>axisfix</b> <b>x_axisfix</b> <b>y_axisfix</b> <b>z_axisfix</b>	Booleans that toggles automatic axis scaling when graph objects are combined. Default=0 (autoscaling on). If <b>axisfix=1</b> , axis limits are those of the kept graph object.
<b>axis_line</b>	Boolean that toggles lines for all axes. Default=1.
<b>x_axis_line</b> <b>y_axis_line</b> <b>z_axis_line</b>	Booleans that toggle axis line for the x, y, or z axis, respectively. Default=1.
<b>zero_line</b>	Boolean that toggles zero lines on all axes. Default=0.
<b>x_zero_line</b> <b>y_zero_line</b> <b>z_zero_line</b>	Booleans that toggle zero lines on the x, y, or z axis, respectively. Default=0.

The following call produces the zero lines and axes for 2D and 3D plots shown in [Figure 4-6](#). This demonstrates axis and zero lines in 2D and 3D plots.

```
plot(sin(-5:.2:5),{columns=3,!grid,
    title="2D Axis Lines and Zero Lines"})
plot(0,0,0,{column=2,!axis,title="3D Zero Lines"})
plot(0,0,0,{column=3,!zero_line,title="3D Axis Lines"})?
```

Figure 4-6 Zero Lines and Axes for 2D and 3D Plots



### 4.3.7 Tics and Grids

Tics and grids appear by default on all plots. You can suppress these features on one or more axes. [Table 4-13](#) describes the keywords `grid`, `tic`, and `tic_lab` which are especially useful because they control all axes.

Table 4-13 Tic and Grid Keywords

Keyword	Description
<code>tic</code>	Boolean that toggles tic marks on all axes. Default=1.
<code>tic_maj</code>	Boolean that toggles major tic marks on all axes. Default=1.
<code>tic_min</code>	Boolean that toggles minor tic marks on all axes. Default=0.
<code>x_tic</code> <code>y_tic</code> <code>z_tic</code>	Booleans that toggle tic marks on the x, y, or z axis, respectively. Default=1.

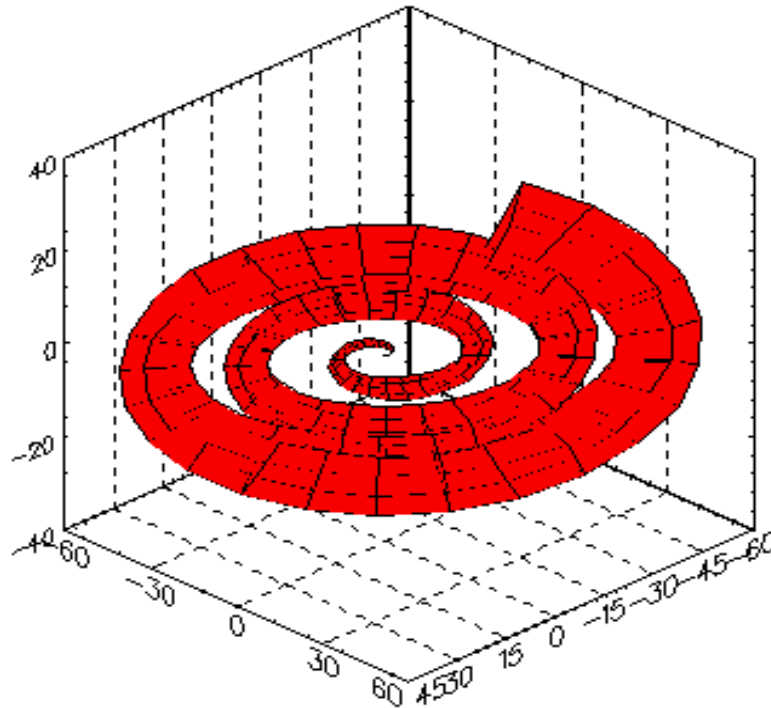
Table 4-13 Tic and Grid Keywords (Continued)

Keyword	Description
<code>x_inc</code> <code>y_inc</code> <code>z_inc</code>	Integers specifying the major tic increment for the x, y, or z axis, respectively.
<code>tic_lab</code>	Boolean that toggles tic mark numbering on all axes. Default=1.
<code>x_tic_lab</code> <code>y_tic_lab</code> <code>z_tic_lab</code>	Booleans that toggle tic mark numbering on the x, y, or z axis, respectively. Default=1.
<code>grid</code>	Boolean that toggles all grids. Default=1.
<code>grid_color</code> <code>grid_style</code> <code>grid_width</code>	Grid color, style, and width attributes can be changed for the entire graph. Colors are specified as described starting on <a href="#">p.137</a> . Line styles and widths are specified as described on <a href="#">p.140</a> .
<code>x_grid</code> <code>y_grid</code> <code>z_grid</code>	Booleans that toggle the x, y, or z grid, respectively. Default=1.

The following instructions produce the changing tic and grid setting shown in [Figure 4-7](#).

```
v=[[1:.15:15],[15:-.15:1]];
vc=[v.*cos(v)];vs=[v.*sin(v)];
vc5=vc.*[ 2;2.5;2.75;2.5;2];
vs5=vs.*[ 1;1.5;2;1.75;1.25];
plot(-vc5,vc5,vs5,{yinc=30,xinc=15,!zgrid})
```

Figure 4-7 Changing Tic and Grid Settings



### 4.3.8 Free Text and Global Text Settings

The `text` keyword places a single string onto the plot. You can alter the angle, color, font, position, size, and style of the string with keywords (see [Table 4-14](#)).

The `text` keyword loosely corresponds to the interactive free text feature. If you want to add more than one text string to a plot or show a variety of text styles, you can work on the plot interactively or combine several plots with the `keep` keyword. Text keywords do not affect text associated with the data, such as labels and titles. You can change these interactively, or, in the case of labels, with keywords.

Table 4-14 Free Text and Global Text Keywords

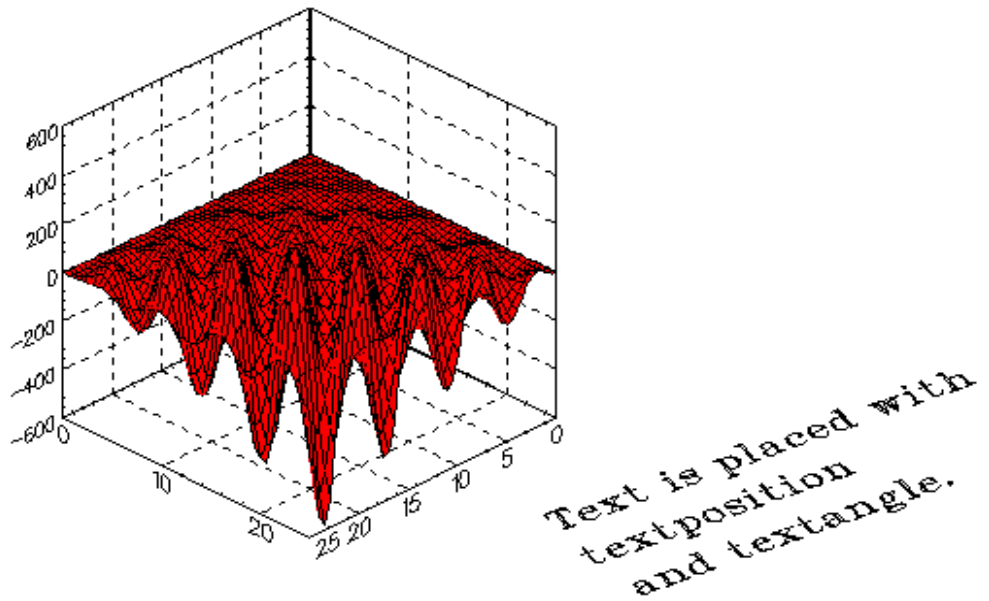
Keyword	Description																				
<b>text</b>	String containing the text to be written on the plot.																				
<b>text_angle</b>	Vector of three float numbers [x,y,z] specifying the angle of the text's clockwise rotation about the axis.																				
<b>text_font</b>	<p>Integer or a string from the following:</p> <table border="1"> <thead> <tr> <th>Integer</th> <th>String</th> </tr> </thead> <tbody> <tr><td>1</td><td>"simplex"</td></tr> <tr><td>2</td><td>"duplex"</td></tr> <tr><td>3</td><td>"triplex"</td></tr> <tr><td>4</td><td>"complex"</td></tr> <tr><td>5</td><td>"script"</td></tr> <tr><td>6</td><td>"greek"</td></tr> <tr><td>7</td><td>"times"</td></tr> <tr><td>8</td><td>"helvetica"</td></tr> <tr><td>9</td><td>"courier"</td></tr> </tbody> </table> <p>Fonts 1 through 6 are Hershey fonts, while fonts 7, 8, and 9 are PostScript fonts. The default font (<b>font=8</b>) is Helvetica.</p> <p>If your platform is not be able to create the font you want in the size you want, it attempts to supply the closest thing.</p>	Integer	String	1	"simplex"	2	"duplex"	3	"triplex"	4	"complex"	5	"script"	6	"greek"	7	"times"	8	"helvetica"	9	"courier"
Integer	String																				
1	"simplex"																				
2	"duplex"																				
3	"triplex"																				
4	"complex"																				
5	"script"																				
6	"greek"																				
7	"times"																				
8	"helvetica"																				
9	"courier"																				
<b>text_color</b>	Integer or string indicating the color name. Specifies the color for <i>all</i> text in the graph. Default=" <b>black</b> ". See <a href="#">p.138</a> .																				
<b>text_position</b>	Vector of two float numbers [x,y] used to place a line of text anywhere in the Xmath Graphics window. The upper left corner of the text line is placed at the specified position. When first drawn, a plot extends from -1 to +1. Note that any float is acceptable, so it is possible to position the text outside the viewport. If you do, it may seem as though the string was not created; you must zoom out to view the text. Default= [0,0].																				
<b>text_size</b>	Floating-point number specifying the size in points. One point is about 1/72 inches.																				
<b>text_style</b>	<p>Integer or string indicating the text style:</p> <table border="1"> <thead> <tr> <th>Integer</th> <th>Font</th> </tr> </thead> <tbody> <tr><td>1</td><td>"plain"</td></tr> <tr><td>2</td><td>"bold"</td></tr> <tr><td>3</td><td>"italic"</td></tr> <tr><td>4</td><td>"bold italic"</td></tr> </tbody> </table> <p>Default is 1 (<b>plain</b>).</p>	Integer	Font	1	"plain"	2	"bold"	3	"italic"	4	"bold italic"										
Integer	Font																				
1	"plain"																				
2	"bold"																				
3	"italic"																				
4	"bold italic"																				



The following example uses `text` keywords; it produces the plot shown in [Figure 4-8](#). Note that `text_position` and `position` work on the same principle.

```
v=[0:.5:25]'; vc=v.*cos(v); vs=v.*sin(v); vm=vs*vc'; plot(v,v,vm)
plot({'keep', scale=[.9,.9],position=[-.4,0],
      x_inc=5,y_inc=10,
      text="Text is placed with \"ntextposition\"+...
      \"n and textangle.\"",
      text_font=3,text_size=14,
      text_angle=[0,0,30],
      text_position=[.1,-.71]})
```

Figure 4-8 Text Changes and Text String Placement



### 4.3.9 Axis Limits and Logarithmic Scaling

You can change the actual scaling of the data (to log scale, for example). You can also specify the minimum and/or maximum range of data you want to see on any dimension (see [Table 4-15](#) for the pertinent keyword descriptions).

Table 4-15 **Axis Limits and Logarithmic Scaling Words**

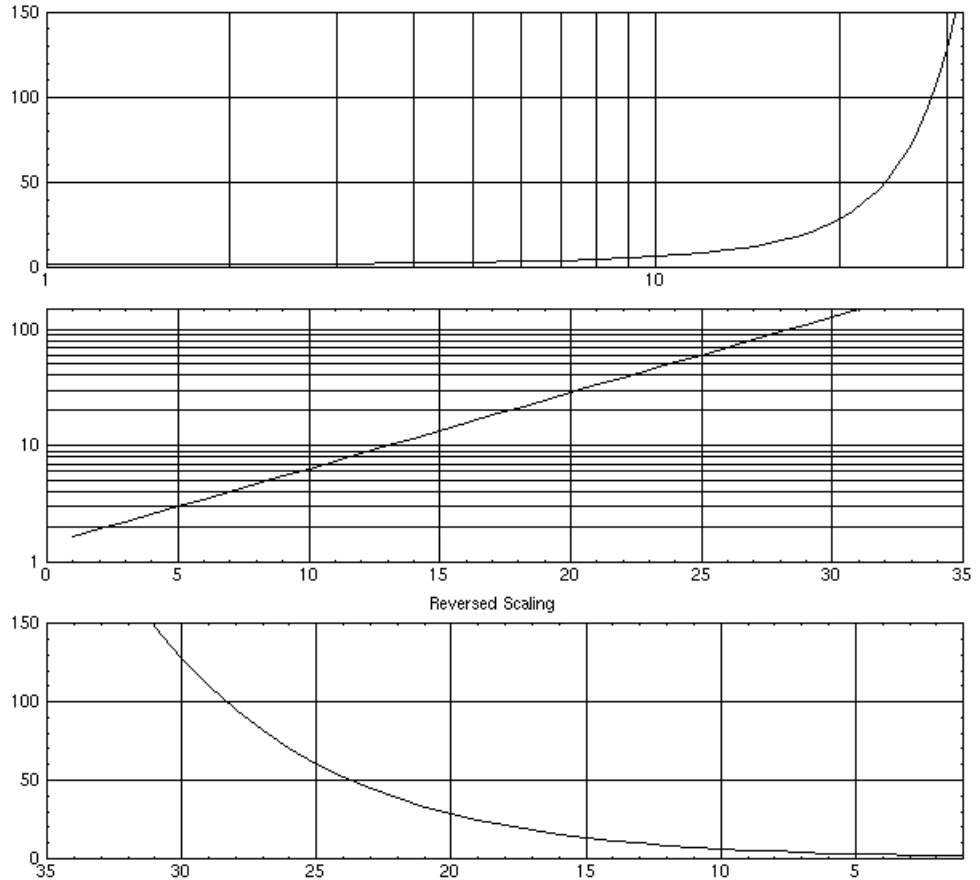
Keyword	Description
<b>log</b>	Turn on/off log scaling for <i>all</i> axes. Default=0.
<b>x_log</b> <b>y_log</b> <b>z_log</b>	Turn on/off log scaling for the specified axis. Default=0.
<b>x_min</b> <b>y_min</b> <b>z_min</b>	Integer indicating the minimum for the x, y, or z axis, respectively.
<b>x_max</b> <b>y_max</b> <b>z_max</b>	Integer indicating the maximum for the x, y, or z axis, respectively.
<b>x_inc</b> <b>y_inc</b> <b>z_inc</b>	Integer indicating the increment for the x, y, or z axis, respectively. For logarithmic axes, this value becomes multiplicative. This means that if the integer is greater than 1, it increases by multiples, and if it is less than 1, it decreases by multiples. So if <b>x_inc=10</b> , tic values are 1,10,100, etc. If <b>x_inc=0.1</b> , values are 1, 0.1, .01, etc.

You can make an axis go backward by making the value of **xmin** greater than **xmax** as illustrated in the following example:

```
x=exp(.5:0.15:5);
plot(x,{x_log,rows=3,xmax=32})
plot(x,{keep,row=2,y_log,ymax=150})
plot(x,{keep,row=3,xmin=35,xmax=1,title="Reversed Scaling"})
```

The results appear in [Figure 4-9](#).

Figure 4-9 Axis Maximums and Minimums



### 4.3.10 Animate

The **animate** keyword allows you to plot new data without redrawing other parts of the plot. It is a Boolean used to show changes in data as quickly as possible for an animation effect between successive plots. Default=0.

The following example plots a series of curves on the same axes. The first plot sets the dimensions of the plot; the second plot holds the dimensions of the first and specifies that only the data will be redrawn each time. The curves are plotted

within a loop and then `animate` is turned off. Alternatively, `plot({reset})` could be used to restore the original settings.

```
a=[0:20/75:20];a=a.*cos(a);
b=[10:-10/75:0];b=b.*sin(b); c=[a,b];
plot ({animate,ymin=-85,ymax=85,xmax=150});
for i=[1:.25:5],[5:-.25:100]];
plot(c*i,{linestyle=1})?
endfor
plot({!animate})
```

### 4.3.11 Placement, Scaling, and Rotation

The **placement**, **scaling**, and **rotation** keywords operate on a graph as a whole (see [Table 4-16](#) for descriptions). This means **scale** changes reduce or enlarge the entire graph, including labels, and so forth. The keywords, **rotate**, **projection**, and **position**, also operate on an entire graph. You can use these keywords when plotting a single graph or multiple graphs ([4.3.4 Multiple Graphs and Graph Positioning](#), p.143).

Table 4-16 **Placement, Scaling, and Rotation Keywords**

Keyword	Description
<b>scale</b>	Vector of two float numbers [x,y]. Each float indicates the amount of compression (float < 1) or expansion (float >1) on the relevant x or y Xmath Graphics window coordinate. Default is [1,1].
<b>rotate</b>	Vector of three float numbers [x,y,z] specifying the angle in degrees of a 3D plot's rotation. Assumes a right-hand coordinate system, based on the Xmath Graphics window axes (not the object coordinates). Rotations are performed counter-clockwise, first about the x-axis, then about the y-axis, then about the z-axis. Because this rotation is based on window coordinates (the current view), it may be simpler to rotate one axis at a time.
<b>projection</b>	String equal to one of the following string values: <b>"stretched"</b> Stretches the graph so that it fills as much of the plotting area as possible. This is the default projection for 2D graphs. <b>"orthographic"</b> Indicates a coscaled setting such that unit distances on all axes are equal. This is the default projection for 3D graphs.

Table 4-16 Placement, Scaling, and Rotation Keywords (Continued)

Keyword	Description
<b>move</b>	Vector of two float numbers [x,y] specifying the distance (in Xmath Graphics window coordinates) to move the object from its current position. [-1,-1] is the lower left corner, [1,1] is the upper right corner, and [0,0] is the center of the window.
<b>position</b>	Vector of two float numbers [x,y] specifying the Xmath Graphics window coordinates of the center of the graph object. Default is [0,0] (the middle of the window). [-1,-1] is the lower left corner, and [1,1] is the upper right corner.

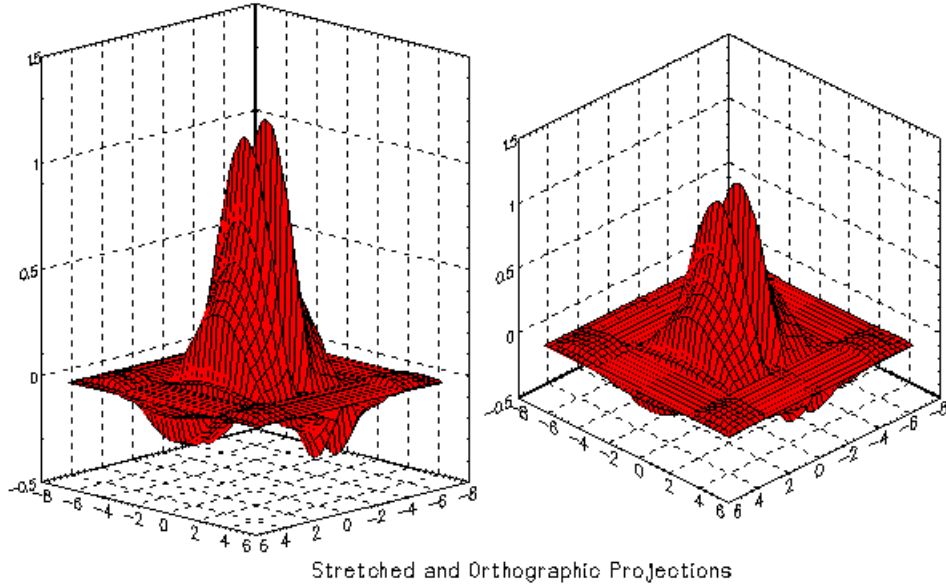
The following example uses scaling, rotation, and projection, and the `text_position` keyword, which works much the same as `position` does. This example creates the projection shown in [Figure 4-10](#).

```
x=[-2*pi:0.35:2*pi]';
x=[x;x];y=(x); z=sin(round(x))./x*(sin(y)./y)';

v=[0:.5:20]';vs=v.*sin(v);vm=vs*vs';
plot (x,y,z,{columns=2,scale=[1,.9],rotate=[-20,0,0],
projection="stretched"})?

g=plot (x,y,z,{keep,column=2,projection="orthographic",
text="Stretched and Orthographic Projections",
text_position=[-.35,-.9]})?
```

Figure 4-10 Stretched and Orthographic Projections



### 4.3.12 Background, Edge, and Face Settings

Table 4-17 Background, Edge, and Face Setting Keywords

Keyword	Description
<b>bg_color</b>	Specifies the Xmath Graphics window background or foreground color. Accepts an integer or a string (see <a href="#">p.138</a> ).
<b>fg_color</b>	
<b>face</b>	Boolean that turns surface filling on 3D surfaces or bar plots on or off. Default=1.

Table 4-17 **Background, Edge, and Face Setting Keywords** (Continued)

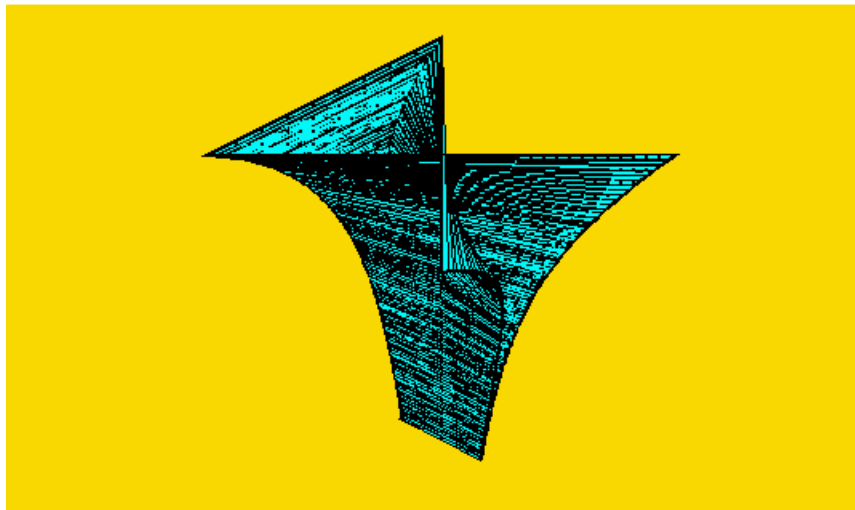
Keyword	Description																						
<b>face_style</b>	<p>Changes the style on all faces. Specify an integer corresponding to the desired style from the following list:</p> <table border="1"> <thead> <tr> <th>Integer</th> <th>Face Style</th> </tr> </thead> <tbody> <tr><td>0</td><td>none (default)</td></tr> <tr><td>1</td><td>solid</td></tr> <tr><td>2</td><td>cross-hatched pattern</td></tr> <tr><td>3</td><td>vertical-line</td></tr> <tr><td>4</td><td>horizontal-line pattern</td></tr> <tr><td>5</td><td>left-slanting diagonal pattern</td></tr> <tr><td>6</td><td>right-slanting diagonal pattern</td></tr> <tr><td>7</td><td>dotted pattern</td></tr> <tr><td>8</td><td>diamond pattern</td></tr> <tr><td>9</td><td>square pattern</td></tr> </tbody> </table> <p><b>NOTE:</b> For black and white monitors, the face styles are only shown if the face color is black or white. If you specify a shade of gray and a face style, you only get gray.</p>	Integer	Face Style	0	none (default)	1	solid	2	cross-hatched pattern	3	vertical-line	4	horizontal-line pattern	5	left-slanting diagonal pattern	6	right-slanting diagonal pattern	7	dotted pattern	8	diamond pattern	9	square pattern
Integer	Face Style																						
0	none (default)																						
1	solid																						
2	cross-hatched pattern																						
3	vertical-line																						
4	horizontal-line pattern																						
5	left-slanting diagonal pattern																						
6	right-slanting diagonal pattern																						
7	dotted pattern																						
8	diamond pattern																						
9	square pattern																						
<b>face_color</b>	<p>Specifies the color of 3D surfaces based on the <i>z</i> data values. Acceptable inputs are an integer, a vector of integers, a string, or a vector of strings (see p.138). If you specify a vector, Xmath cycles through the given sequence.</p>																						
<b>edge</b>	<p>Toggles the display of web lines on 3D surfaces or bar plots. Default=1.</p>																						
<b>edge_color</b>	<p>Specifies the color of the web lines on 3D surfaces or bar plots. Acceptable inputs are an integer, a vector of integers, a string, or a vector of strings (p.138). If you specify a vector, Xmath cycles through the given sequence.</p>																						
<b>edge_style</b>	<p>Sets the style of all web lines. This keyword accepts an integer or a string equivalent indicating the border line type. Default=1 (a solid line). Allowed values are:</p> <table border="1"> <thead> <tr> <th>Integer</th> <th>String</th> </tr> </thead> <tbody> <tr><td>0</td><td>" " "</td></tr> <tr><td>1</td><td>"----"</td></tr> <tr><td>2</td><td>"- -"</td></tr> <tr><td>3</td><td>". . ."</td></tr> <tr><td>4</td><td>"-.-"</td></tr> </tbody> </table>	Integer	String	0	" " "	1	"----"	2	"- -"	3	". . ."	4	"-.-"										
Integer	String																						
0	" " "																						
1	"----"																						
2	"- -"																						
3	". . ."																						
4	"-.-"																						
<b>edge_width</b>	<p>Sets the width of all web lines. This keyword, like <b>line_width</b>, accepts any floating point number.</p>																						

The following example displays a variety of edge and face specifications.

```
x=logspace(1,180,90);y=logspace(90,270,90);  
z=45:134;  
a=[-x;x;-x;x];b=[y;-y;-y;y];c=( [z;z;z;z] );  
  
plot(a,b,c,{edge_width=2,  
        face_style=7,!grid,!axis,bg_color="gold",  
        edge_color="black",face_color="cyan"})
```

The graph appears in [Figure 4-11](#).

Figure 4-11 **Edge and Face Styles**



### 4.3.13 Lighting Source Settings

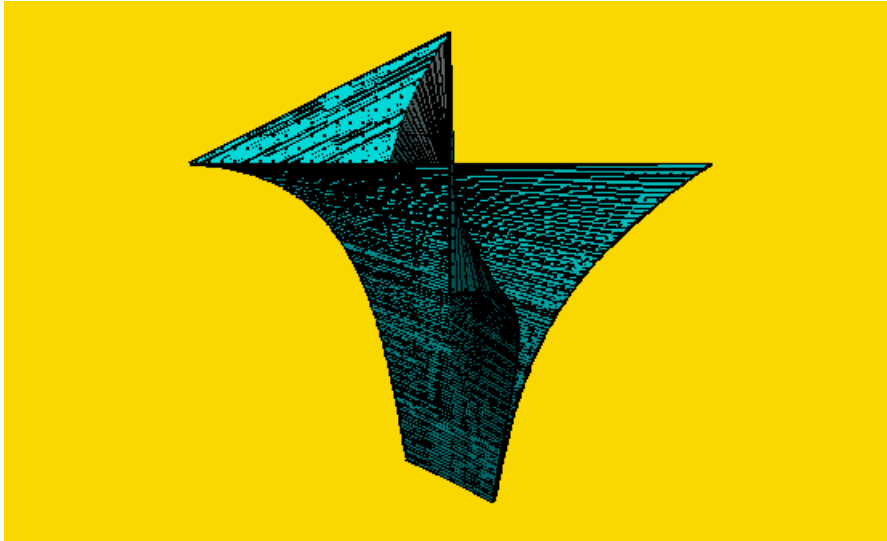
Table 4-18 **Lighting Source Setting Keywords**

Keywords	Description
<b>light</b>	Boolean that turns light source on or off. Default=0.
<b>light_color</b>	Integer (color number) or string (color name) specifying light source color. Default="white".
<b>light_direction</b>	Vector of three real numbers [x,y,z] indicating the direction in which the light travels. Light source location is assumed to be infinitely far away. The default path vector is [1,-1,3].



Setting `light= 1` for the plot shown in Figure 4-11 produces a very different graph (see Figure 4-12)

Figure 4-12 Edge and Face Styles with Light Added



### 4.3.14 Reusing plot Attributes

#### Hold Keyword

Table 4-19 Holding Graph Attributes

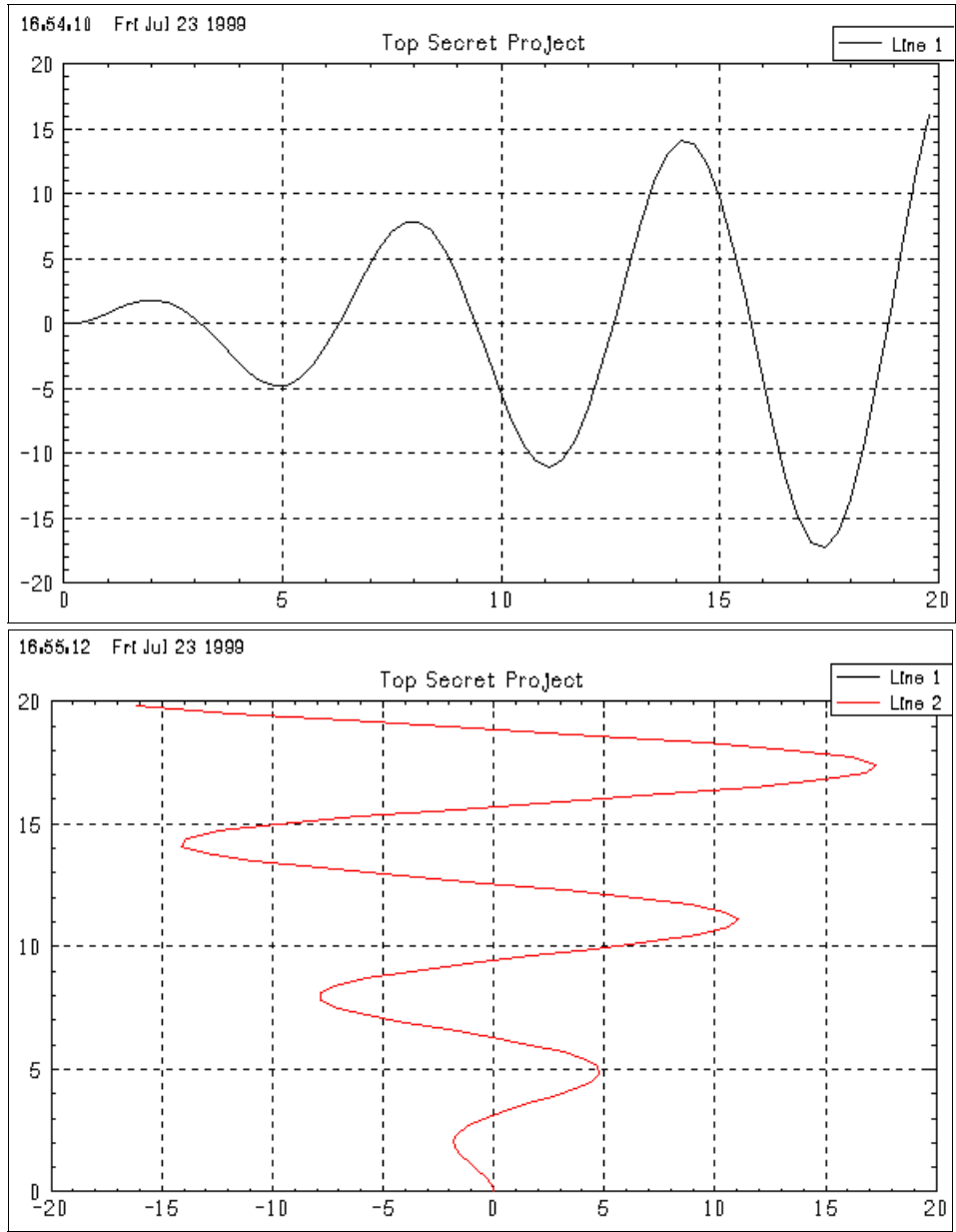
Keyword	Description
<b>hold</b>	Boolean. When <b>{hold}</b> is used with other keywords it makes them 'permanent', applying them to all future graphs until <b>hold</b> is terminated with <b>plot(!hold)</b> . Note that <b>{!hold}</b> removes keywords specified by the most recent hold. When you invoke <b>hold</b> in numerous plot calls, a hold stack is formed. You can specify a <i>negative</i> integer as an argument to <b>hold</b> ( <b>{hold =-n}</b> ) to remove the last <b>n hold</b> invocations from the hold stack. Use <b>plot(reset)</b> to clear the entire hold stack. (Default=0)
<b>reset</b>	Resets plot options to their startup values. Use this keyword alone, that is, <b>plot(reset)</b> .

You can use plot attributes, such as line widths, the legend, and titles, with the **hold** keyword, but you cannot use plot types (strip, bar, contour, and polar) with **hold**. When you use an attribute with the **hold** keyword, it replaces the current default. The following example uses the hold stack:

```
v=[0:.3:20]';vs=v.*sin(v);  
plot(v,vs,{hold, time, date, legend,  
      title="Top Secret Project", scale=[1,.95]})  
  
plot([-vs,-vs],[v,v] ,{hold,scale=[1,.95]})  
  
plot(v+12,vs,{!hold})  
  
plot([-vs,vs] ,{!hold})
```

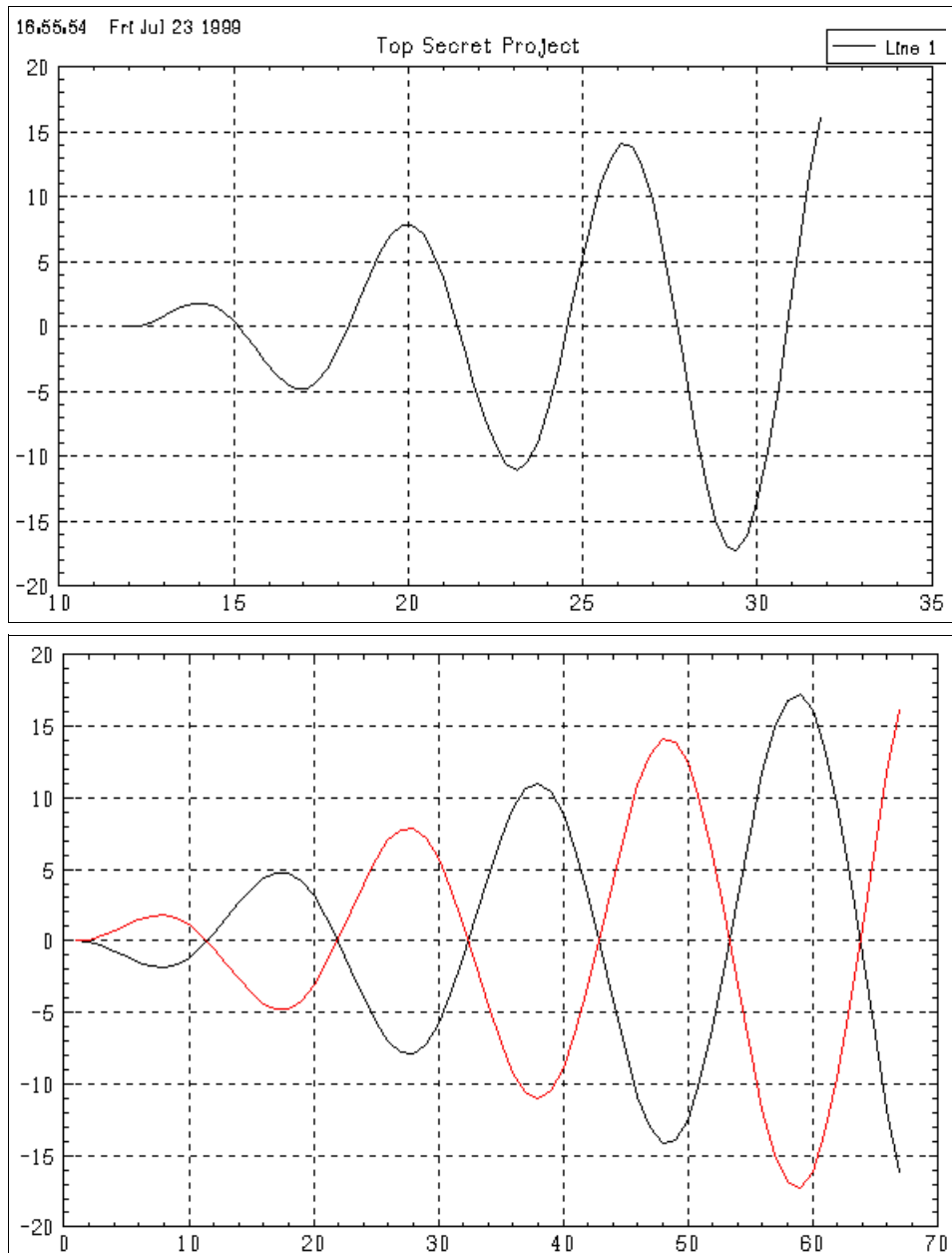
You can see the results of each of these four plots in [Figure 4-13](#) and [Figure 4-14](#).

Figure 4-13 Results of First and Second plot Commands Using hold



4

Figure 4-14 Results of the Third and Fourth plot Commands Using !hold



### Using an Alias in the Keyword String

Another way to reuse plot attributes is to create an alias. You can then use the alias in the keyword string. An advantage of using an alias is that the defaults are not affected. You can store the aliases you use frequently in your **startup.ms** file.

You can obtain the results we achieved with the **hold** keyword with an alias:

```
alias XX time,date, title="Top Secret Project",scale=[1,.95], legend
plot([-vs,vs1},{XX})
```

This example reproduces the fourth **plot** command above (see [Figure 4-14](#)).



**NOTE:** The **rows** and **columns** keywords are a special case and cannot be explicitly used with the **hold** keyword (see [p.144](#)). Because they are initiators, they are automatically held until a plot call is made that does not contain the **row** or **column** keyword.

### 4.3.15 Strip Plots

The **strip** keyword is an integer indicating the number of data lines to be plotted on each strip plot in a given set.

The default is 1; Xmath plots one strip per channel or column of data for up to 10 strips. After 10, **strip** adds data to the existing strips.

If a value  $n$  is specified, Xmath creates strip plots with  $n$  data lines per strip plot. If the number of lines is not evenly divisible by  $n$ , the data lines corresponding to the remainder are lost.

Xmath creates strip charts such that the first data line appears on the first strip chart, the second appears on the second strip chart, and so on until each strip in the first cycle has a data line. All the lines in the first cycle have the same line style. Xmath draws the second cycle of lines with a different line style. You can interactively modify line styles, colors, markers, and so forth (see [Interactive Xmath Graphics Window](#) on [p.171](#)). When you alter a data line, Xmath also changes all lines in that cycle. Note, however, that changes that do not affect data (for example, grid lines) are *not* passed to other strips.

By default, strip plots are laid out as follows:

- **plot(y,{strip=N})** where  $y$  is an  $(m \times n)$  matrix and  $n$  is an integer multiple of  $N$ . Strips are arranged as an  $((n/N) \times 1)$  matrix of plots. Each strip contains  $N$  graphs.

- **plot(y,strip=N)** where **y** is an  $(m \times n \times T)$  PDM and  $m$  is an integer multiple of  $N$ . The results are an  $((m/N) \times n)$  plot matrix. (Think of the PDM as a column vector of blocks.) Each subplot contains  $N$  graphs.
- **plot(y,strip,columns=m,rows=n)** where **y** is a matrix with  $N$  columns and  $N$  is an integer multiple of  $m \times n$ . This syntax creates an  $(m \times n)$  plot matrix that is filled with graphs rowwise. The number of data lines in each subplot is  $N$ . This option is very handy because it precludes having to write a nested loop to fill in a matrix of plots.
- **plot(y,strip,columns=m,rows=n)** where **y** is an  $(m_1 \times n_1 \times T)$  PDM and  $m_1 \times n_1$  is an integer multiple of  $m \times n$ . The result is identical to that obtained by plotting **makematrix(y)** with the same keywords. When used with **columns=1**, this syntax specifies a column of strip plots instead of a matrix of strip plots.

To demonstrate strip plots, load the following file:

```
load "$XMATH/demos/sys.xmd"
```

This file contains **sys**, a lightly damped mechanical system that inputs two forces and outputs two positions. It is discrete, sampling at 1 second. For this example, we use this data to create a system with a sampling rate of 1 second and named inputs and outputs:

```
sysd=system(sys,{dt=1,inputNames=["Force 1","Force 2"],  
outputNames=["Position 1","Position 2"]});
```

Obtain a frequency response of the new system.

```
f = [1:200]/400; gd=freq(sysd,f);
```

Create a continuous version of this system and create a frequency response for it:

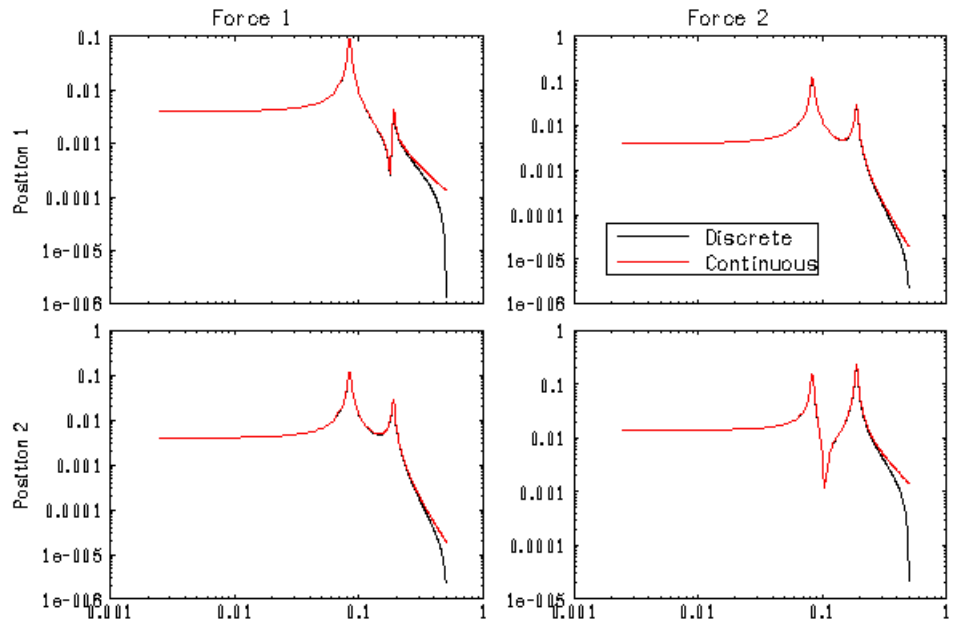
```
sysc=makecontinuous(sysd); gc=freq(sysc,f);
```

Plot the continuous and discrete systems.

```
plot(abs([gd;gc]),  
{xlog,ylog,strip=2,legend=["Discrete","Continuous"],!grid})?
```

The results of this example appear in [Figure 4-15](#).

Figure 4-15 Frequency Responses



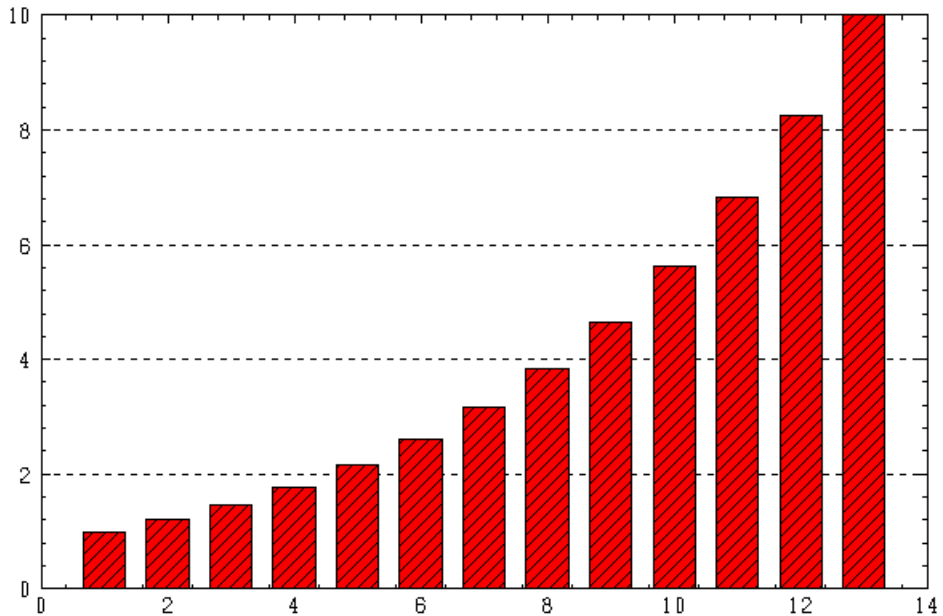
### 4.3.16 Bar Plots

The `bar` keyword is a Boolean used to indicate that the current plot is a bar plot. Each coordinate is plotted as the center of a bar whose height is the y or z coordinate. Default=0.

```
plot(logspace(1,10,13),{bar,face_style=5,!x_grid})
```

This plot appears in [Figure 4-16](#).

Figure 4-16 **Bar Plot**



### 4.3.17 Contour Plots

A contour plot is a 3D plot that shades portions of the plot based on the z data values; the effect is like a topographical map. You can use a legend to show which value ranges correspond to the color or fill pattern shown in the contour plot. If you specify the keyword **face\_color** and supply a vector of colors, those colors will be used to shade the data values.

Table 4-20 **Contour Plots Keywords**

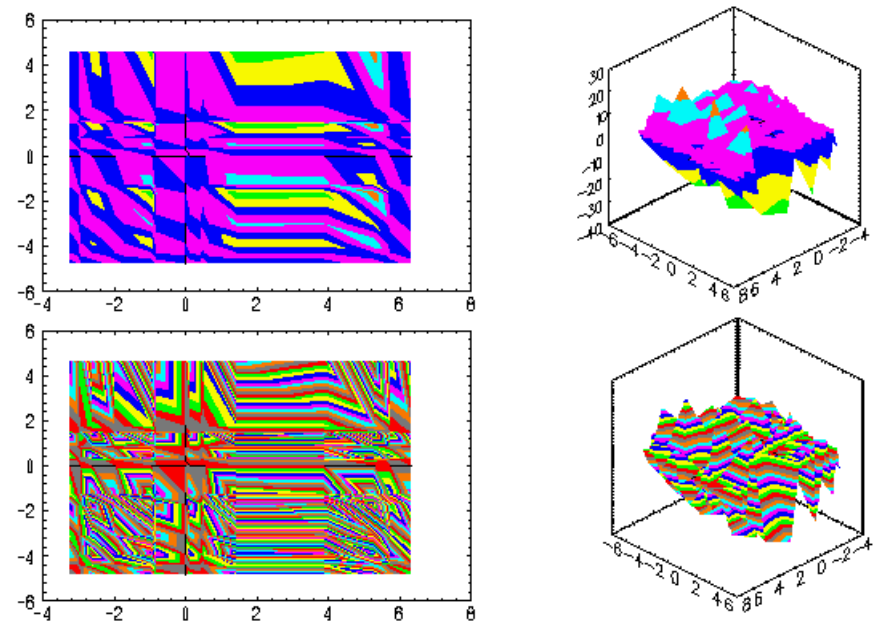
Keywords	Description
<b>contour</b> <b>contour2d</b>	Booleans used to indicate that the current 2D plot is a contour plot. Requires x, y, and z data. Default=0.
<b>contour3d</b>	Boolean used to indicate that the current 3D plot is a contour plot. Requires x, y, and z data. Default=0.
<b>contour_interval</b>	Float value which can be used with <b>contour</b> or <b>contour3d</b> to determine the intervals of the contour plot. Defaults to the internally calculated tic label values of the z data.



The following instructions produce [Figure 4-17](#):

```
v=[0:.5:7]';
vc=v.*cos(v); vs=v.*sin(v); vm=vs*vc';
plot(vc,vs,vm,{rows=2,columns=2,contour2d,!grid})
plot(vc,vs,vm,{keep,row=2,contour2d,!grid,
  contour_interval=1.3})
plot(vc,vs,vm,{keep,column=2, contour3d,!grid})
plot(vc,vs,vm,{keep,row=2,column=2,contour3d,
  !grid,contour_interval=1.3,!z_tic_lab})
```

Figure 4-17 3D Contours with Different Intervals



### 4.3.18 Polar Plots

The polar plot option draws a 2D plot on a polar grid. Polar plots require a radius (magnitude vector) and an angle vector in degrees (theta):

```
plot(radius,theta,{polar})
```

Table 4-21 **Polar Plot Keywords**

Keywords	Descriptions
<b>polar</b>	Boolean used to make the current 2D plot a polar plot. Default=0.
<b>r_inc</b>	Integer specifying the polar radius increment value to be marked on the vertical axis of the polar plot.
<b>r_max</b>	Integer specifying the maximum polar radius to be plotted on a polar plot.
<b>theta_inc</b>	Integer specifying the polar angle increment value to be marked around the circumference of the polar plot.
<b>theta_min</b> <b>theta_max</b>	Integer specifying the minimum or maximum polar angle increment to be marked around the circumference of the polar plot.

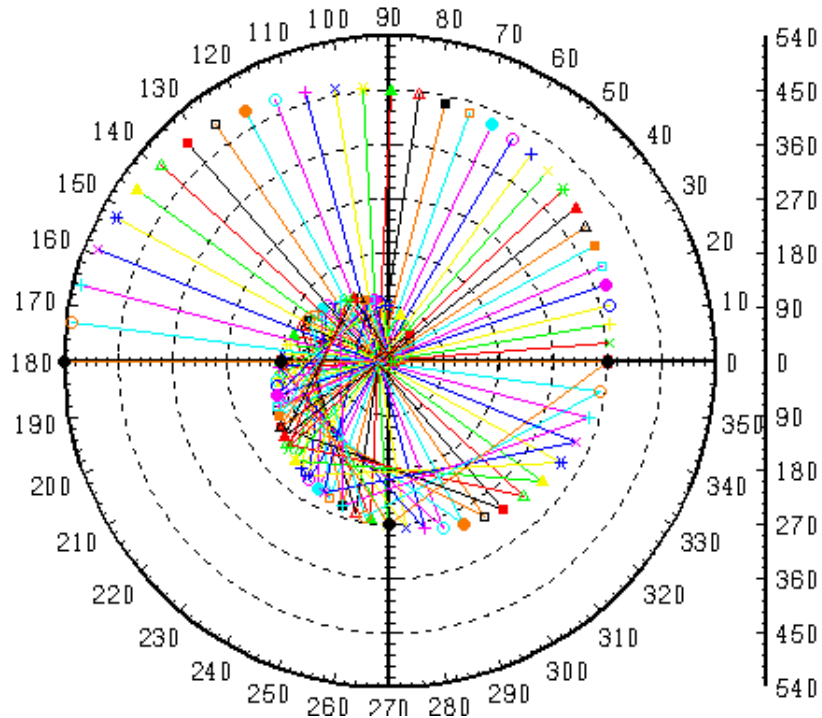
The following instructions produce [Figure 4-18](#).

```
t=[logspace(1,180,32);logspace(90,270,32);  
   logspace(180,360,32);logspace(360,540,32)];  
plot (t,t,{polar,!x_grid,r_inc=90,  
        theta_inc=10,marker})
```

### 4.3.19 Clearing the Xmath Graphics Window

To clear the Xmath Graphics window, type **ERASE** in the Xmath Commands window command area.

Figure 4-18 **Polar Plot**



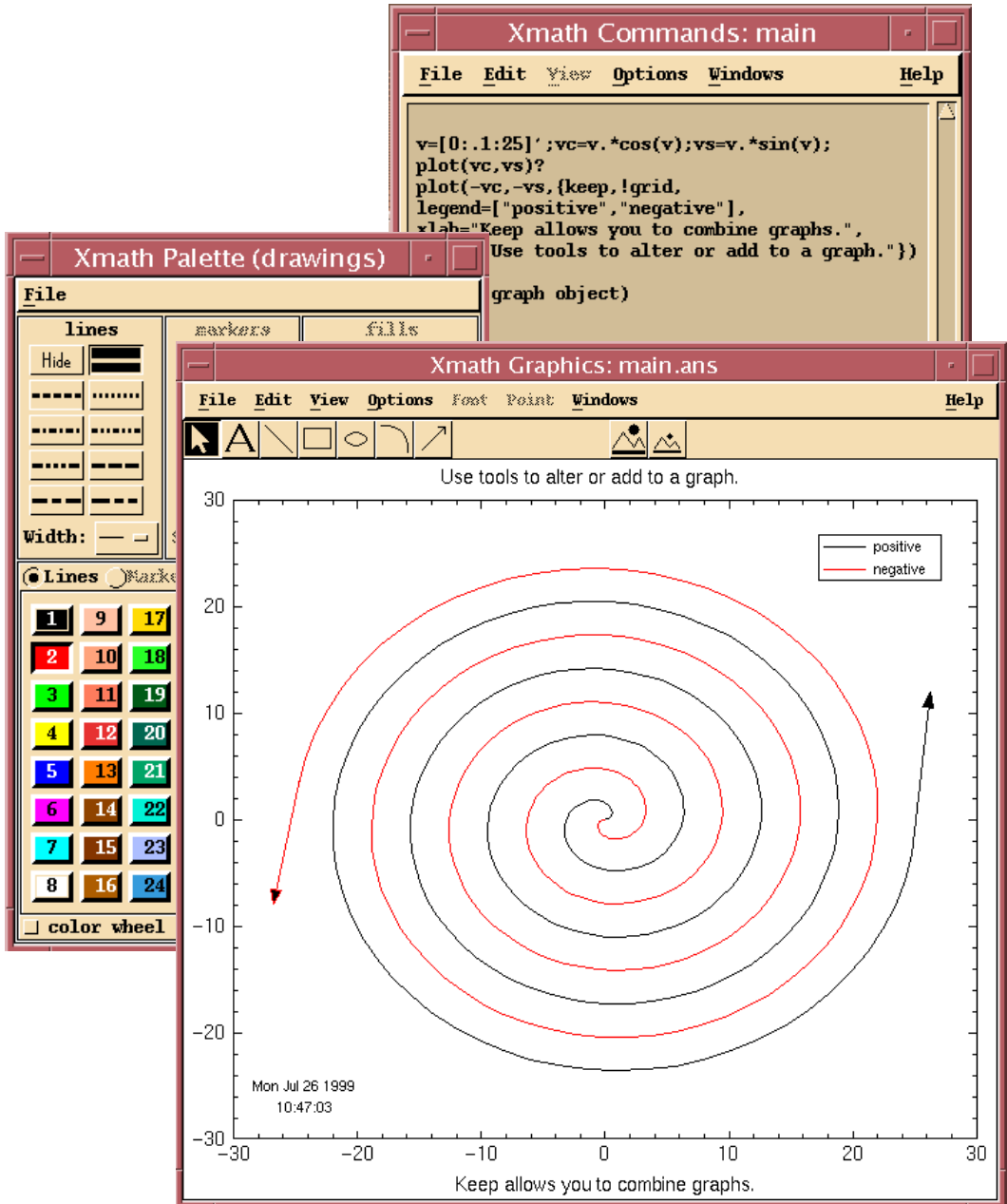
## 4.4 Interactive Xmath Graphics Window

The Xmath Graphics window displays Xmath plots and other graphics. It is typically opened and updated whenever **plot()** (or a function that calls **plot()**, such as **bode()**) is invoked. It provides extensive interactive facilities for building, modifying, and viewing two-dimensional (2D) and three-dimensional (3D) graphics. You can specify graph characteristics, such as labels, placement, and

size, as keywords to `plot()`, or you can add or modify them interactively from the Xmath Graphics window menus or the Xmath Palette.

Graphs are composed of objects such as lines, labels, markers, and axes. Object attributes can be prespecified as keywords when the plot command is issued from the Commands window command area. (Keyword usage is discussed in [4.3 Using Keywords with plot](#), p.131.) You can also manipulate an object's attributes interactively from the Xmath Graphics window's menus or toolbar or from the Xmath Palette. [Figure 4-19](#) shows the graphics environment on UNIX platforms.

Figure 4-19 Xmath Graphics Environment (UNIX Platform)



4

In the example shown in [Figure 4-19](#), the graph originated with a `plot()` function call in the Xmath Commands window command area:

```
v=[0:.1:25]';vc=v.*cos(v);vs=v.*sin(v);
plot(vc,vs)?

plot(-vc,-vs,{keep,!grid,
  legend=["positive","negative"],
  xlabel="Keep allows you to combine graphs.",
  title="Use tools to alter or add to a graph."})
```

Note that two plots were combined using the `keep` keyword (see [p.130](#)). Graphical additions (the arrows, for example) were created with tools from the toolbar. New objects (for example, the timestamp and datestamp) were added from the Options menu in the Xmath Graphics window. The mouse was used to select and position objects (for example, the legend, timestamp, and the datestamp).

#### 4.4.1 Working Interactively

The most common approach is to start with a graph and then use interactive tools to alter it to your satisfaction.

- To make interactive changes, first click on an object to select it.

Xmath selects the closest object to the mouse-click. When you select text, round handles appear on the corners of the text box. When you select a line or curve, it is highlighted and has a thicker appearance.

When you make a selection, the appropriate attributes are enabled for both the Xmath Palette and the Xmath Graphics window menus. For example, when the background is selected, the Xmath Palette shows that only the fill patterns are available (line and marker styles are disabled, and the Fills button is pushed). If a label is selected, the Font and Point menus become available in the Xmath Graphics window; in the Xmath Palette, the Text button is pushed. You can then change the font and point size from the Xmath Graphics window and select a new color from the Xmath Palette.

- Place the pointer over an object and drag to move objects.

You can also use pulldown menus to modify and move selected objects, make global changes (zooming, rotating, and so forth), or add objects (via the Options menu). Click on an object to select it. When you pull down the menus, only the items appropriate for the object selected are displayed.

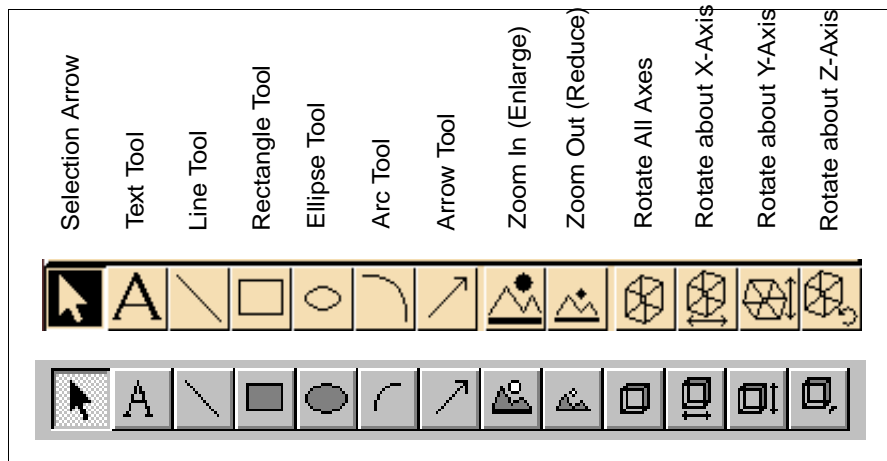
## 4.4.2 Toolbar

The toolbar appears in the Xmath Graphics window by default. This feature provides quick mouse access to simple graphical drawing tools and the zoom and rotate tools. To toggle the toolbar off and on, select Options→Icon Bar (UNIX only). [Figure 4-20](#) shows the toolbar in both UNIX and Windows and shows labels for each tool.



**NOTE:** Note that not all tools are enabled for all plot types. In general, zooming and rotation are disabled for all multiple graph plots (such as strip plots).

Figure 4-20 The Toolbar (UNIX and Windows)



### Selection Arrow

You use the selection arrow to reset the cursor to selection mode after you use a drawing or text tool.

### Text Tool

To use the text tool, click on the Text Tool toolbar button. You receive an I-beam cursor. Move to the graph area and click. An empty text box appears; you may start typing. The text box expands as you type. To create a paragraph (continuous

lines of text) press Return and keep typing. To start a new string, click in a different place. To turn off the text tool, click the selection arrow or another tool.

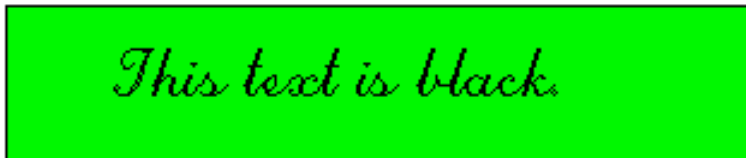
The key commands described in *Editing Text by Selecting, Copying, and Pasting* on p.20 are also active in the text box. Note that the font and size in the text box are not what is displayed in the graph. Figure 4-21 demonstrates this in the center text piece. To edit existing text, click the Text Tool toolbar button, and then click in the text; the text box reappears. Note that the changes you make are not displayed until you click the selection arrow (or another tool).

To format text, select it, and then choose a font style and point size from the Font and Size menus on the Xmath Palette; you can also enable checkboxes for bold and italic font. To change text color, select the text, and then select a color from the Xmath Palette. Figure 4-21 shows reversed text created with a text string and a graphical object.

Figure 4-21 Using the Text Tool and the Xmath Palette

Text can be a single string or a multiple-line paragraph.  
Press Return to break a text string.

**When inputting or editing text,  
you don't see the final font and size  
until you choose the selection arrow.**



 Drawing Tools

The line tool, rectangle tool, ellipse tool, arc tool, and arrow tool are primitive drawing tools that allow you to draw in the Xmath Graphics window. When you



use drawing tools, they remain active until you choose the selection arrow or another tool. To use a tool, click on the desired toolbar button; a crosshairs cursor appears. Press down MB1 and drag until the desired shape is formed; then release MB1.



**NOTE:** You cannot resize or reshape the polygons you create because these are primitive tools.



### Zoom In/Zoom Out

To zoom in on a graph, click the toolbar button with the larger image on the left. Position the mouse over your plot; then click and drag to create a box around all or a portion of the graph; the area captured in your box is enlarged to fill the Xmath Graphics window. Every time you zoom in, the previous view is saved on a stack.

You can use Zoom Out toolbar button (the toolbar button with the smaller image) to undo a series of enlargements. If you zoom out when you are at the default view, the graph is reduced by approximately 10%.

The zoom feature is disabled for multiple graph plots.



### Rotation Tools

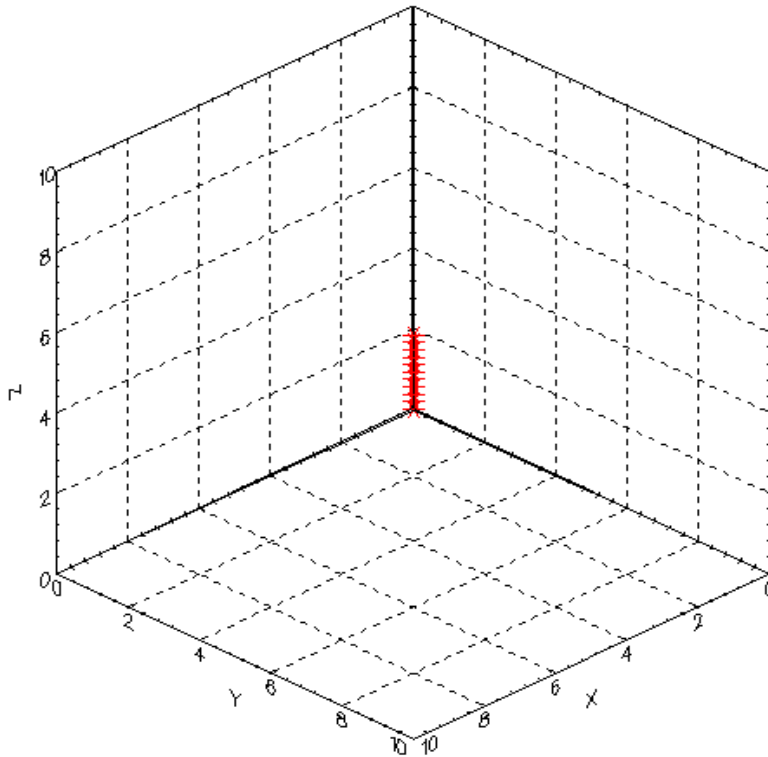
Rotation is only allowed with 3D plots and other contour plots. The first rotation tool allows rotation on all axes. The other tools are constrained to rotate only in the directions indicated by the arrows. Select a tool, and then move the cursor to the plot area. Press down MB1, and slowly drag the cursor in the direction allowed. The data disappears and you see the plot axes turning in response to your mouse movement; when the axes are in the position you wish to view the plot, release MB1, and the data is redrawn. To return to the original graph, select View→Reset.

Consider the following example:

```
x = [0:10];
y = [0:10];
z = [0:10];
graph = plot(x,y,z, {marker=1, x_lab="X", y_lab="Y", z_lab="Z"})
```

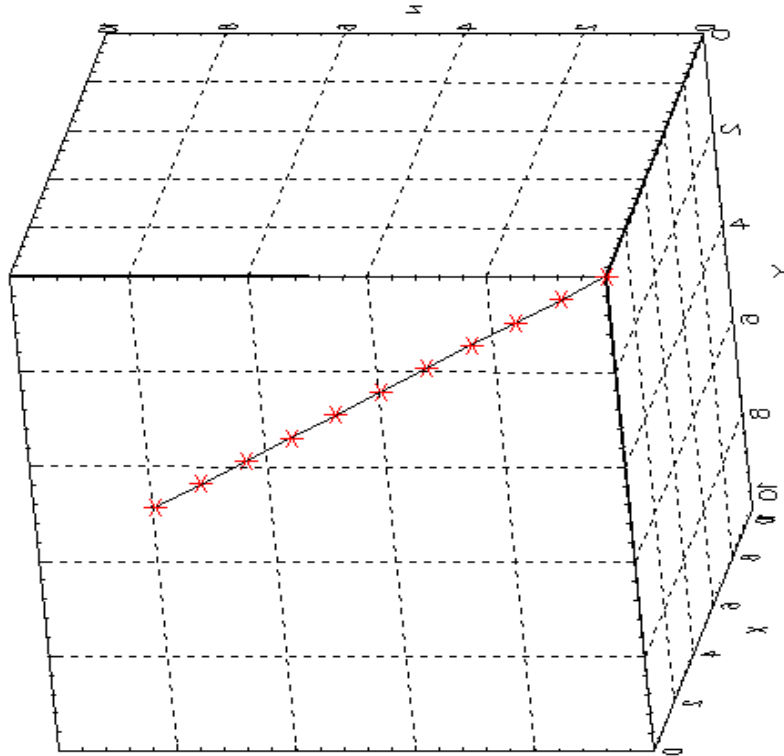
The default plot appears in [Figure 4-22](#); this 3D vector is projected in such a way that it isn't particularly useful.

Figure 4-22 **Default View of 3D Vector**



Using the rotation toolbar buttons, you can rotate this plot in almost an infinite number of ways. [Figure 4-23](#) shows one rotated view, which gives more information than the default plot.

Figure 4-23 Rotated View of 3D Vector

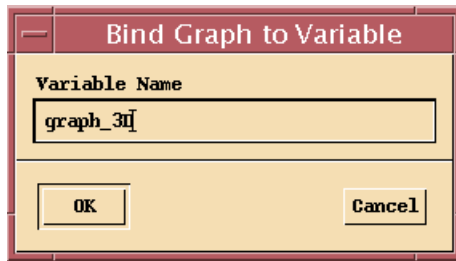


### 4.4.3 Menus

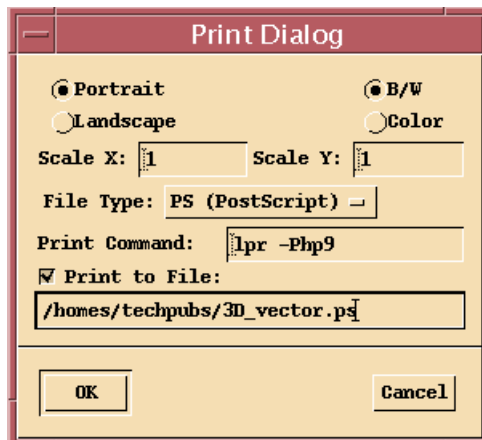
This section discusses the menus that are available in the Xmath Graphics window.

#### *File*

**Bind to Variable** — Saves the current Xmath Graphics window image as a variable that you specify. You can redisplay the plot at a later time by typing the variable name, or you can reuse the graph in another plot by using the **keep** keyword.



Print — Raises the Print dialog (UNIX version shown below), which allows you to send the image in the current Xmath Graphics window to a printer. The UNIX Print dialog allows you to save a graphics file in PS (PostScript), EPS, HPGL, PICT, CGM-ANS, CGM-CAL, or CGM-TXT format.



For UNIX, the default printer shown in the Print Command field is set at the operating system level. The default line printer for your system is assumed. The system's default print command is set using the environment variables **XMATH\_PRINT** and **PRINTER**. **XMATH\_PRINT** defines the default print utility, while **PRINTER** defines the default printer.

On Windows operating systems, this command raises the standard Windows Print dialog from which you can also print to a file.

For all systems, you can print from the Xmath command area using the **HARDCOPY** command (see the `MATRIXx` online Help *HARDCOPY* topic for details).

Iconify Window — Lower the Xmath Graphics window.

Close Window — Kill the Xmath Graphics window.

### Edit

Cut — Delete the selected object.

Move Up, Down, Left, Right — Move the selected object. Distance will be 1-2 pixels, depending on the size of the window.



---

**NOTE:** These commands work only for objects that you place on the graphic interactively or with keywords such as **legend**, **date**, **time**, and so forth.

---

Bring to Front — Bring the selected object to the front.

### View

Reset — Reset to the original graph.

Projection — Change the projection for the current graph. In stretched projection (the default for 2D plots), the plot is scaled to occupy the maximum amount of available space. For orthographic projection (the default for 3D plots), all axes are coscaled to have the same unit length (a circle will look round and a cube will look like a cube, not a shoe box).

Lights — Toggle lights on/off.

### Options

Timestamp — Add the timestamp to the graph. The timestamp appears in *hour:minutes:second* format and is positioned by default in the upper left corner of the graph. You can select it and then change the text attributes or move it. To remove the timestamp, select it then use Ctrl-x.

Datestamp — Adds the datestamp to the graph. The date stamp appears in the *day:month:date:year* format and is positioned by default in the upper left corner of the graph. You can select it and then change the text attributes or move it. To remove the datestamp, select it, and then use Ctrl-x.

Legend — Toggle the legend on/off.

Icon Bar — Toggle the toolbar on/off (UNIX only).

### Font (UNIX Only)

This menu is only available if text is selected in the Xmath Graphics window. PostScript fonts available are Times, Helvetica, and Courier. Stroke fonts available are Simplex, Duplex, Triplex, Complex, Script, and Greek.



---

**NOTE:** Your machine might not be able to display all fonts in all sizes listed in the Point menu; the same is true for printer output. In either case, the device produces the font it can manage.

---

### Point (UNIX Only)

This menu is only available if text is selected in the Xmath Graphics window. Point sizes are 6, 9, 10, 12, 14, 18, 24, 36, and 48. You can also choose the font style: plain, bold, italic, or bold italic.



---

**NOTE:** Your machine might not be able to display all fonts in all sizes; the same is true for printer output. In either case, the device produces the font in the closest size it can manage.

---

### Tools (Windows Only)

You can use this menu to duplicate all the functions on the toolbar. Select the menu item rather than the toolbar button to perform the same function (see [4.4.2 Toolbar](#), p.175).

## Windows

This menu allows you to bring other Xmath windows to the front quickly. Of special note is the Palette, which is used to make interactive changes to graphic objects.

### 4.4.4 Xmath Palette

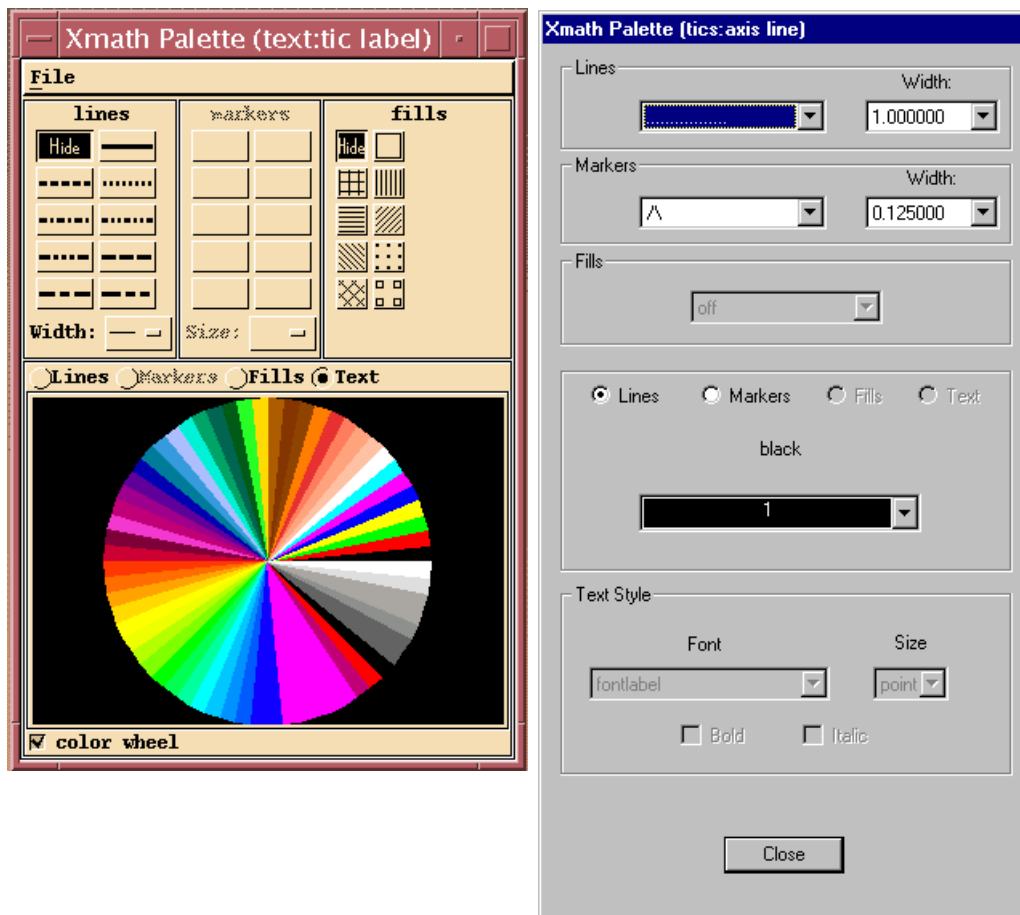
Xmath provides another window from which you can work interactively.

To bring up the Xmath Palette:

Click `Windows→Palette` in the Xmath Graphics window.

The Xmath Palette comes on view. The UNIX version looks a little different from the Windows version, but the functions are essentially the same. The default UNIX version appears in [Figure 4-19](#), p. 173. You can also select colors via a color wheel on the UNIX version; this view of the Xmath Palette appears in [Figure 4-24](#), as well as the Windows version of the window.

Figure 4-24 Xmath Palette: UNIX and Windows Versions



To use the Xmath Palette:

1. Select an object in the Xmath Graphics window.

The object type appears at the top of the Xmath Palette in the title bar.

The radio button for the active option—Lines, Markers, Fills, or Text—is pushed. All items that are available for the selected item are active, whereas others are inactive in the window.



2. Click the radio button for the option that you wish to change: Lines, Markers, Fills, or Text.
3. Make the desired changes for this option.

You can control the color of all attributes. You can turn lines, markers, and fills off or choose the type of each. For lines and markers, you can also choose the width. For text you have a choice of fonts, sizes, plain, bold, italic, or bold italic style. The text choices mimic the options available through the Font and Point menus in the Xmath Graphics window in UNIX.

4. Modify as many attributes for as many objects as you want, and then click Close to close the window.



# 5

## *Data Objects and Operators*

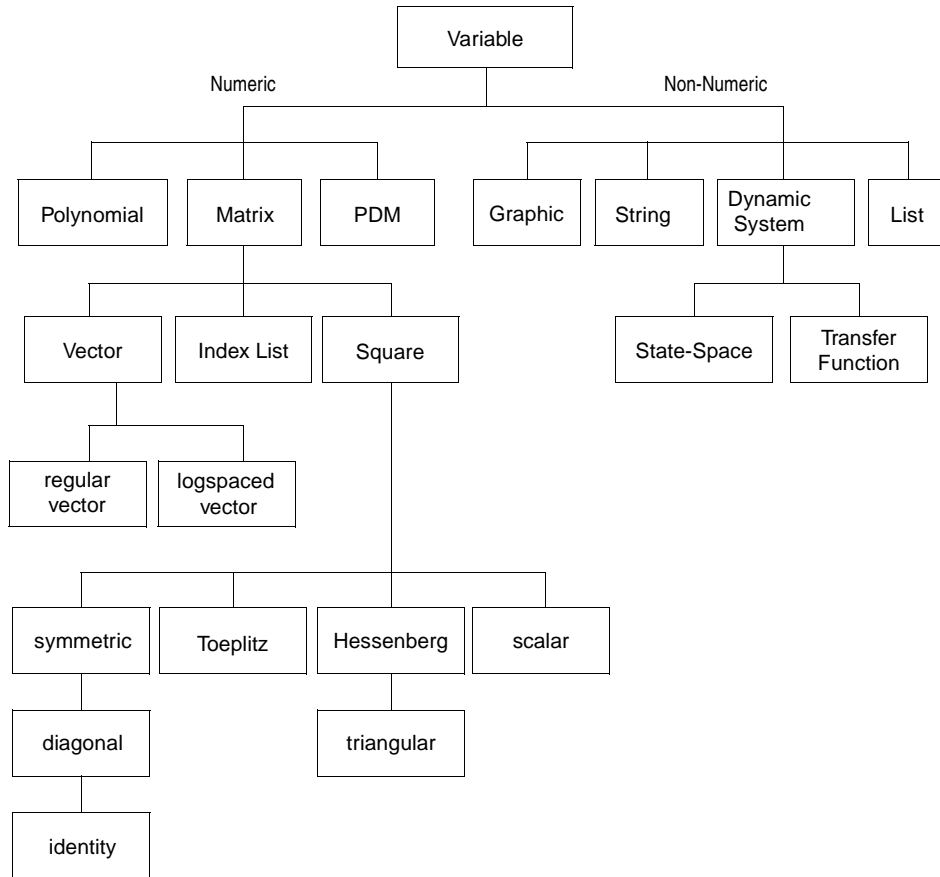
This chapter provides a conceptual overview and detailed descriptions of Xmath data objects and operators.

### **5.1 Data Hierarchy**

Xmath data hierarchy, as shown in [Figure 5-1](#), is divided into numeric and nonnumeric branches.

The matrix, for example, is general. It consists of matrices of various shapes. The square matrix is a specific kind of matrix that requires an equal number of rows and columns, but otherwise *inherits* the characteristics of the matrix. A scalar is a special kind of square matrix with dimensions of  $1 \times 1$ . A scalar is also defined as a special kind of vector, because it is a vector with a length of 1.

Figure 5-1 Object Relationships



Xmath's object-based structure provides three significant benefits:

**simplified data management** — As variables in Xmath can represent complex groupings of data, you don't have to track numerous variables. For example, with a state-space system using **system(A,B,C,D)**, all the data (including input names, output names, etc.) is stored in a single variable. The matrices can be deleted.

**optimized performance** — Many Xmath data objects were designed to take advantage of optimized algorithms. This is especially true of the specialized matrices. The eigenvalues of a symmetric matrix, for example, can be found

more quickly with a symmetric eigensolver rather than a general eigensolver. Xmath recognizes the special properties of a matrix and uses the appropriate, optimized algorithm.

**natural syntax** — Because Xmath recognizes the special properties of each type of data object, operations are intuitive. For example, it is more natural to multiply two polynomials by typing  $p1*p2$  than it is to call `convolve(p1,p2)`.

### 5.1.1 Data Object Descriptions

This chapter describes Xmath data objects in the following order:

- Matrix
- Polynomial
- Parameter-dependent matrix (PDM)
- Dynamic system
- String
- List

Some of the categories are subdivided. For example, dynamic systems include state-space systems and transfer functions, and matrices include the following:

- Vector
  - Regular vector
  - Logspaced vector
- Square
  - Symmetric, Diagonal, Identity, Toeplitz
  - Hessenberg, Triangular
  - Scalar
- Indexlist



---

**NOTE:** To reproduce the examples, cut and paste the **bold courier** text.

---

## 5.2 Matrix

A matrix is an object organizing  $m$  rows and  $n$  columns ( $m \times n$ ) of real or complex numbers (elements). A complex number contains both a real and an imaginary term. A matrix is complex if at least one element is complex; to qualify as a real matrix, all elements must be real.

Matrices are specified with the following syntax elements:

- A matrix specification is enclosed in square brackets.
- Matrix column elements must be separated by commas.
- A semicolon separates rows.

For example, `x=[jay, 4; 3,-1]`. In a formatted matrix, a line feed replaces the semicolon:

```
x=[jay, 4 # Line Feed
  3, -1] # Return
```

(If your machine does not have a Line Feed key, see [Table 1-7](#), p.20.) The matrix specification ends with a right bracket.

Specific types of matrices are also created with functions such as `zeros()`, `random()`, `diagonal()`, etc. These functions require row and column dimensions as inputs:

```
set seed = 0
x=random(3,4)
```

```
x (a rectangular matrix) =
0.211325    0.756044    0.000221135    0.330327
0.665381    0.628392    0.849745      0.685731
0.878216    0.068374    0.560849      0.662357
```

The functions `check()` and `is()` can be used to determine if a variable is a matrix. For brief explanations of `check()` and `is()`, see [6.3.2 Object Query Functions](#), p.238. Sample syntaxes are: `check(x,{matrix})` or `is(x,{matrix})`.

Use `size` to find the row and column dimensions of a matrix:

```
size(x)
ans (a row vector) = 3 4
```

To find the total number of elements, use `length()`:

```
length(x)
```

```
ans (a scalar) = 12
```

Many classes stem from the matrix class, and it is the primary component of several more specialized objects.

### 5.2.1 Matrix Concatenation

*Concatenation* (combining several matrices into a new matrix) is performed using square bracket operators [ ]. Right concatenation is indicated with commas [ , ]; bottom concatenation is indicated by semicolons [ ; ].

For example,

- **[A,B]** concatenates **B** to the right of **A** (where **B** must have the same number of rows as **A**).
- **[A;B]** concatenates **B** to the bottom of **A** (where **B** must have the same number of columns as **A**).

```
x=random(3,2)*12
```

```
x (a rectangular matrix) =
```

```
8.71621    2.38217
6.53109    2.7849
2.77468    2.59756
```

```
x=[x,ones(3,4);ones(2,2),zeros(2,4)]
```

```
x (a rectangular matrix) =
```

```
8.71621    2.38217    1    1    1    1
6.53109    2.7849     1    1    1    1
2.77468    2.59756    1    1    1    1
1          1          0    0    0    0
1          1          0    0    0    0
```

### 5.2.2 Matrix Operators

The operators in [Table 5-1](#) have special meanings for matrices:

```
scalar operator matrix
```

usually means applying the operator elementwise.

```
mat1=[1,1,1,1; 2,2,2,2; 3,3,3,3];
mat2=mat1 * mat1'
```

```
mat2 (a square matrix) =
  4      8      12
  8     16     24
 12     24     36

3 * mat1

ans (a rectangular matrix) =

  3      3      3      3
  6      6      6      6
  9      9      9      9
```

Table 5-1 **Matrix Operations**

Operator	Effect
+	Addition (or unary plus). Matrices must have the same dimensions.
-	Subtraction (or unary minus). Matrices must have the same dimensions.
*	Matrix multiplication. The number of columns in the first matrix must equal the number of rows in the second matrix.
/	Matrix right division. $A/B$ solves the equation $X \forall B=A$ . The number of columns in $A$ must equal the number of rows in $B$ .
\	Matrix left division. $B \backslash A$ solves the equation $B \forall X=A$ . The number of columns in $B$ must equal the number of rows in $A$ .
'	Transpose (unary suffix).
*'	Complex conjugate transpose (unary suffix).
.*	Elementwise matrix multiplication. Matrices must have the same dimensions.
./	Elementwise division (left divided by right). Matrices must have the same dimensions.
.\	Elementwise division (right divided by left). Matrices must have the same dimensions.
^ or **	Raise a square matrix to a scalar power.
.^ or .**	Raise elements to a power. Another matrix of the same size can contain the powers.
.*.	Kronecker product.
./.	Kronecker right division.



Table 5-1 **Matrix Operations** (Continued)

Operator	Effect
<code>.\.</code>	Kronecker left division.
<code>&amp;</code>	Elementwise logical and.
<code> </code>	Elementwise logical <b>or</b> .
<code>!</code>	Elementwise logical not.
<code>&lt;</code>	Elementwise less than.
<code>&gt;</code>	Elementwise greater than.
<code>&lt;=</code>	Elementwise less than or equal.
<code>&gt;=</code>	Elementwise greater than or equal.
<code>==</code>	Elementwise equal.
<code>&lt;&gt;</code>	Elementwise not equal.
<code>=</code>	Assignment.

### 5.2.3 Matrix Indexing

Indexing (extracting a specific subset of matrix elements) is performed using the parentheses operators (`()`). Indices can consist of any one of the following:

- Two integers specifying the desired row and column.

`A(i,j)` extracts from `A` the element located in row `i`, at column `j`. This can be demonstrated using the matrix `mat2` created earlier.

```
mat2
mat2 (a square matrix) =
     4     8    12
     8    16    24
    12    24    36

mat2(2,3)
ans (a scalar) =    24
```

- Two vectors of integers specifying a range of rows and columns.  $A(\mathbf{vector1}, \mathbf{vector2})$  extracts a portion of  $A$  with rows corresponding to vector 1 and columns corresponding to vector 2.

```
mat2(1:2,2:3)

ans (a square matrix) =

     8     12
    16     24
```

- An index list that specifies all desired element locations in terms of row and column indices. An index list can be created with the `find()` or `indexlist()` functions. For more on the index list object, see [5.8 Index Lists](#), p.239.

```
ijList=find(mat2>15)

ijList (an index list) =

     2     2
     2     3
     3     2
     3     3
```

- Note that `find()` returns the row and column coordinates for elements in `mat2` that are greater than 15: (2,2), (2,3), (3,2), and (3,3). You can use indexing to display the values in these index list locations:

```
mat2(ijList)

ans (a column vector) =

    16
    24
    24
    36
```

### Indexing with the Colon Operator ( : )

The colon operator (:) is a wildcard for *all elements*, thus  $A(i, :)$  is the  $i^{\text{th}}$  row of  $A$  and  $A(:,j)$  is the  $j^{\text{th}}$  column of  $A$ .

You can use a wildcard and a decreasing vector to reverse the columns of a matrix.

```
mat2(:, [3:-1:1])

ans (a square matrix) =
```

```

12    8    4
24   16   8
36   24  12

```

Here wildcards are used to extract rows, which are reassembled into a new matrix:

```
mat3=[mat2(1,:);sqrt(mat2(2,:));mat2(3,:)^2]
```

```
mat3 (a square matrix) =
```

```

    4         8        12
2.82843     4    4.89898
144        576    1296

```

## 5.2.4 Vector

The vector class is a subclass (or specialization) of the matrix class. A vector object is a matrix that has a row or column dimension equal to 1. Vectors can be *oriented* as either rows or columns.

Many of the functions defined for matrices apply to vectors as well. Vectors also have many special behaviors. The most important of these are listed below:

- Use `^` to raise elements to a power (for matrices, use `.^`).

```
[1:4]^ [1:4]
```

```
ans (a row vector) = 1    4    27    256
```

- Vectors can be indexed with a single index variable. Thus  $v(i)$  is the  $i^{\text{th}}$  element of the vector  $v$ . A single vector of integers can also be used as an index.

```
a=[2,4,6,8,10]
```

```
a (a row vector) = 2    4    6    8    10
```

```
a([1,3,5])
```

```
ans (a row vector) = 2    6    10
```

- The colon (`:`) wildcard expands vectors in column form. `aVector(:)` is always defined as a column, *regardless* of whether the vector is a row or column.
- The `length()` function is the most natural method of determining the length of vector. `length(aVector)` is defined as `max(size(aVector))`.

- To see if a variable is a vector, invoke **is(var,{vector})** or **check(var,{vector})**.

To determine whether the vector is a row or column, use **is(var,{row})** or **is(var,{column})** (or use **check**). The **{row}** and **{column}** keywords imply **{vector}**. For brief explanations of **check** and **is**, see [6.3.2 Object Query Functions](#), p.238.

### Regular Vector

A regular vector is evenly spaced, with each element a fixed increment from the previous value. If a regular vector is created with the colon operator, Xmath stores it as just three values (start:increment:stop). You can treat it as a vector, but it is displayed in a special manner.

- A regular vector can only be a row vector. Transposing it expands it to full size (turns it into a simple vector).

```
x=0:0.33:1
```

```
x (a regularly spaced vector) = 0 : 0.33 : 1
```

```
x'
```

```
ans (a column vector) =
```

```
0  
0.33  
0.66  
0.99
```

- Putting a regular vector between square braces [ ] will expand it.

```
[x]
```

```
ans (a row vector) = 0 0.33 0.66 0.99
```

A regular vector is internally expanded for most operations, except indexing.

Although a regular vector is stored in compact form (as start, stop, and increment values), it has the same dimensions as if it were created in expanded form. You can view the sizes of all the variables in your current partition with the **who** command. Use the **size** function to view the size of a single variable:

```
size(x)
```

```
ans (a row vector) = 1 4
```

## Logspaced Vector

A logspaced vector is just like a regular vector except that its points are evenly spaced on a log scale. It can only be created with the `logspace()` function. `logspace()` inputs are the initial value, the final value, and the number of points desired in the vector. All the display considerations for a regular vector apply to logspaced vectors.

```
x1=logspace(0.1,10,4)

x1 (a log-spaced vector) = 0.1 : 10 (4 points)

[x1]

ans (a row vector) = 0.1 0.464159 2.15443 10
```

## 5.2.5 Square Matrix

The square matrix class is a subclass of the matrix class. A square matrix object has equal row and column dimensions.

All of the functions that are defined for matrices are also defined for square matrices. However, there are several square matrix functions that are not valid for rectangular matrices. The most important of these are shown in [Table 5-2](#).

Table 5-2 **Functions That Are Only Valid for Square Matrices**

Function	Result
<code>^</code> or <code>**</code>	raise matrix to a power ( $A^3=A \times A \times A$ )
<code>.^</code> or <code>.**</code>	raise each element to a power
<code>cholesky()</code>	Cholesky decomposition
<code>cosm()</code>	matrix cosine (use <code>cos</code> elementwise)
<code>det()</code>	determinant
<code>eig()</code>	eigenvalues
<code>expm()</code>	matrix exponential (use <code>exp</code> elementwise)
<code>hessenberg()</code>	Hessenberg decomposition
<code>inv()</code>	inverse
<code>logm()</code>	matrix logarithm (use <code>log</code> elementwise)

Table 5-2 **Functions That Are Only Valid for Square Matrices** (Continued)

Function	Result
<b>lu()</b>	L-U decomposition
<b>orth()</b>	orthogonal decomposition
<b>polynomial()</b>	characteristic polynomial
<b>polyvalm()</b>	evaluates polynomial function of a matrix
<b>qz()</b>	generalized eigenvalues
<b>rref()</b>	reduced-row echelon form
<b>schur()</b>	Schur form
<b>sinm()</b>	square matrix sine (use <b>sin</b> elementwise)
<b>sqrtm()</b>	matrix square root (use <b>sqrt</b> elementwise)
<b>trace()</b>	find the sum of the diagonal elements of a matrix

## Symmetric

The symmetric matrix class is a subclass of the square matrix class. A symmetric matrix object is equal to its transpose.

For most applications, symmetric matrices act just like square matrices. Certain algorithms take advantage of their special structure to achieve improved results. For example, the eigenvalues of a symmetric matrix can be found more quickly than the eigenvalues of a general matrix; also, the answers are constrained to be purely real.

```
a=[1:4];b=[a;a;a;a]
b (a square matrix) =
    1    2    3    4
    1    2    3    4
    1    2    3    4
    1    2    3    4

is(b,{symmetric})
ans (a scalar) =    0

c=tril(b,1) + tril(b,1)'
```

```

c (a square matrix) =
  2  3  1  1
  3  4  5  2
  1  5  6  7
  1  2  7  8

is(c,{symmetric})

ans (a scalar) = 1

```

### Diagonal()

The diagonal matrix class is a subclass of the symmetric matrix class and the triangular matrix class (see [p.201](#)). A diagonal matrix object has zero in all positions except along the main diagonal.

The **diagonal()** function can be used to extract a diagonal from a matrix. Extract the diagonal from the matrix **c** defined above:

```

d=diagonal(c)

d (a column vector) =
  2
  4
  6
  8

```

If a vector is used as an input, a matrix is created that has the vector on the main diagonal.

```

e=diagonal(d)          # use the vector d as the
                      # diagonal of a new matrix

e (a square matrix) =
  2  0  0  0
  0  4  0  0
  0  0  6  0
  0  0  0  8

```

### Identity

The identity matrix class is a subclass of the diagonal matrix class. An identity matrix object has ones on the main diagonal and zero for all other elements. The function **eye()** creates an identity matrix from row and column dimensions:

```
eye(3,3)
```

```
ans (a square matrix) =
```

```
 1   0   0
 0   1   0
 0   0   1
```

For most applications, identity matrices act like square matrices. Certain algorithms (such as multiplication and inversion) take advantage of their special structure.

### **Toeplitz**

The Toeplitz matrix class is a specialization of the square matrix class with constant entries along the diagonals. A Toeplitz matrix can be described by its first row and first column (if it is symmetric, it can be described by a single vector). The matrix left and right division operations have been overloaded for solving matrix equations of the form  $T \times X = A$  and  $X \times T = A$  (where T is a Toeplitz matrix):

```
t=toeplitz([3,2,1],[1,2,3])
```

```
t (a toeplitz matrix) =
```

```
 3   2   1
 2   3   2
 3   2   3
```

### **Hessenberg()**

The Hessenberg matrix class is a subclass of the square matrix class. A Hessenberg matrix has zeros in all elements below the first subdiagonal or above the first superdiagonal. The **hessenberg()** function puts a matrix  $A$  in Hessenberg form  $H$ , defined such that  $A = T \times H \times T^*$  where  $T$  is a unitary transformation matrix of the same size and type as  $A$ .

```
hessenberg([1,2,3;1,2,3;1,2,3])
```

```
ans (a square matrix) =
```

```
 1          -3.53553          0.707107
-1.41421     5             -1
 0          -3.14018e-16     3.14018e-16
```



## Triangular

The triangular matrix class is a specialization of the Hessenberg matrix class. A triangular matrix object has zeros in all elements above the main diagonal (upper triangular) or below the main diagonal (lower triangular).

```
set seed 0
a=round(rand(4,4)*4)

a (a square matrix) =

    1    3    0    1
    3    3    3    3
    4    0    2    3
    3    1    2    1

12345678 112345678 212345678

aTriu=triu(a)      # an upper triangular

aTriu (a square matrix) =

    1    3    0    1
    0    3    3    3
    0    0    2    3
    0    0    0    1

aTril=tril(a)      # a lower triangular

aTril (a square matrix) =

    1    0    0    0
    3    3    0    0
    4    0    2    0
    3    1    2    1
```

## Scalar

The scalar class is a subclass of the square matrix class. A scalar object is a matrix with a single row and a single column.

Any function or operator defined for a matrix is also defined for a scalar. However, scalars have many special properties when used in combination with other classes of objects, as shown in the samples that follow.

**scalar x matrix** — Each element of the matrix is multiplied by the scalar. The same holds true for vectors and PDMs. Division works the same way.

```
5*[1:5]
```

```
ans (a row vector) = 5 10 15 20 25
```

```
ans/5
```

```
ans (a row vector) = 1 2 3 4 5
```

**scalar x polynomial** — If the polynomial is in factored form, the gain of the polynomial is multiplied by the scalar. (Polynomials are discussed in detail starting on [p.205](#).) If the polynomial is in coefficient form, each coefficient is multiplied by the scalar. Division works the same way.

Using a scalar with a polynomial in roots form:

```
4*polynomial(1:4)
```

```
ans (a polynomial) =
```

```
4(x - 1)(x - 2)(x - 3)(x - 4)
```

Using a scalar with a polynomial in coefficients form:

```
makepoly(1:4)
```

```
ans (a polynomial) =
```

```
3 2  
x + 2x + 3x + 4
```

```
ans/0.5
```

```
ans (a polynomial) =
```

```
3 2  
2x + 4x + 6x + 8
```

**scalar x system** — Multiplies the gain of the system by the scalar. (Dynamic-system objects are discussed in detail starting on [p.227](#).) For transfer functions, the numerator polynomial is multiplied by the scalar. For state-space systems, the C and D matrices are multiplied by the scalar. Division works the same way.

```
system([2,2;2,2],[3;3],[4,4],1);
```

```
2*ans
```

```
ans (a state space system) =
```

```
A  
2 2  
2 2
```

```
B  
3
```

```

3
C
8 8
D
2
X0
0
0

System is continuous

system(makepoly(2:5),makepoly(0:3))

```

ans (a transfer function) =

$$\frac{2x^3 + 3x^2 + 4x + 5}{x^2 + 2x + 3}$$

initial integrator outputs

0  
0  
0

Input Names

-----  
Input 1

Output Names

-----  
Output 1

System is continuous

**ans/2**

ans (a transfer function) =

$$\frac{x^3 + 1.5x^2 + 2x + 2.5}{x^2 + 2x + 3}$$

initial integrator outputs

0  
0  
0

Input Names

-----

Input 1

Output Names

-----

Output 1

System is continuous

**matrix+scalar** — The scalar is added to each element of the matrix. This operation is commutative. The same holds true for vectors and PDMs. Subtraction works the same way.

```
-2+(3+(ones(3,3)))
```

ans (a square matrix) =

```
2 2 2
2 2 2
2 2 2
```

**polynomial+scalar** — Converts the polynomial to coefficient form and adds the scalar to the scalar (order 0) term of the polynomial. This operation is commutative. Subtraction works the same way.

```
p=polynomial(3:5)
```

p (a polynomial) =

```
(x - 3)(x - 4)(x - 5)
```

```
2+p
```

ans (a polynomial) =

```
3 2
x - 12x + 47x - 58
```

**matrix(vector,vector)=scalar** — Copies the scalar to each element of the specified partition of the matrix. The same holds true for vectors and PDMs.

```
o=ones(4,5);
```

```
o([2:3],[2:4])=32
```

o (a rectangular matrix) =

```
1 1 1 1 1
1 32 32 32 1
1 32 32 32 1
1 1 1 1 1
```

## 5.3 Polynomial( )

Polynomials take the form  $x^3 + 9x^2 - 4x + 7$  or  $x(x - 2)(x + 6)$ . The first notation is in coefficients form; its coefficients (1, 9, -4 and 7) are plainly shown. The second polynomial is in roots form (its roots being 0, 2, and -6). Polynomial objects consist of a vector of coefficients or roots and a single independent variable (a text string, usually a single character).

Polynomials can be defined in terms of their roots or coefficients. The **polynomial()** function creates a polynomial object where roots are the elements of a vector or eigenvalues of a square matrix you supply. You can specify a text string for the polynomial variable. **makepoly()** converts a simple vector into a polynomial.

Create a polynomial from its roots with **polynomial()**. The polynomial is displayed in roots form:

```
p1=polynomial([1*jay, -1*jay, 1,
              2*jay, -2*jay, 2,
              3*jay, -3*jay, 3], "j")

p1 (a polynomial) =
      2      2      2
(j - 1)(j - 2)(j - 3)(j + 1)(j + 4)(j + 9)

p2=polynomial([9,8,7])

p2 (a polynomial) =
(x - 7)(x - 8)(x - 9)
```

Create a polynomial from a vector with **makepoly()**; the polynomial will be displayed in coefficients form:

```
p3=makepoly(logspace(1,3,5),"L")

p3 (a polynomial) =
      4      3      2
L + 1.31607L + 1.73205L + 2.27951L + 3

p4=makepoly(1:.5:3)

p4 (a polynomial) =
      4      3      2
x + 1.5x + 2x + 2.5x + 3
```

### 5.3.1 Polynomial Operators

The following operators are valid for polynomials:

- + polynomial addition
- polynomial subtraction
- \* polynomial multiplication
- / creates a transfer function

Operations can only be performed between polynomials that have the same independent variable or between polynomials and scalars.

**p5=p2+p4**

```
p5 (a polynomial) =
      4      3      2
x + 2.5x - 22x + 193.5x - 501
```

**p6=p2\*p2**

```
p6 (a polynomial) =
      2      2      2
(x - 7) (x - 8) (x - 9)
```

**sysp=3/p6**

```
sysp (a transfer function) =
```

```

      3
-----
      2      2      2
(x - 7) (x - 8) (x - 9)
```

```
initial integrator outputs
```

```
0
0
0
0
0
0
```

```
Input Names
```

```
-----
```

```
Input 1
```

```
Output Names
```

```
-----
```

```
Output 1
```

System is continuous

The functions in [Table 5-3](#) can handle parts of polynomials; for more information on inputs and outputs, see the MATRIX<sub>X</sub> online Help.

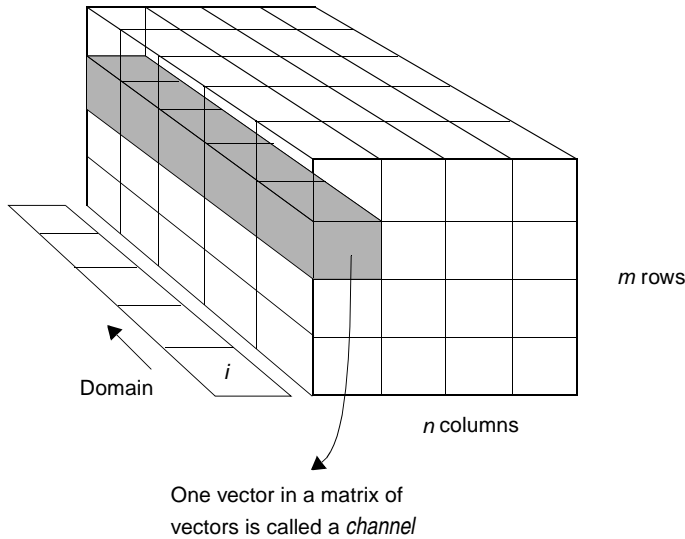
Table 5-3 **Polynomial Handling Functions**

<b>roots()</b>	extracts the roots of a polynomial
<b>makematrix()</b>	extracts the coefficients of a polynomial
<b>domain()</b>	extracts the independent variable from a polynomial or PDM
<b>polyval()</b>	evaluates a polynomial at each element of a given matrix
<b>polyvalm()</b>	evaluates a polynomial over an entire square matrix

## 5.4 Parameter-Dependent Matrix (PDM)

A parameter-dependent matrix (PDM) is a flexible extension of the matrix data type. It consists of a vector of same-size matrices with a vector attached to it. The attached vector (or parameter) is referred to as the *domain* ([Figure 5-2](#)). A PDM also has optional string names for its rows and columns (see [Figure 5-3](#)).

Figure 5-2 Structure of a PDM



PDM data is stored as a series of matrices indexed by a single domain vector. Computations involving the PDM are performed on each matrix separately. Data can also be handled as a series of vectors, called *channels*, having a common domain vector (time or frequency, for example). In this format, the computations are performed on each vector of the data separately.

Used either way, PDMs provide a convenient method for storing data as a function of a parameter and are particularly useful in the analysis of multiple input and/or output dynamic systems, where they can be used to store time or frequency responses.

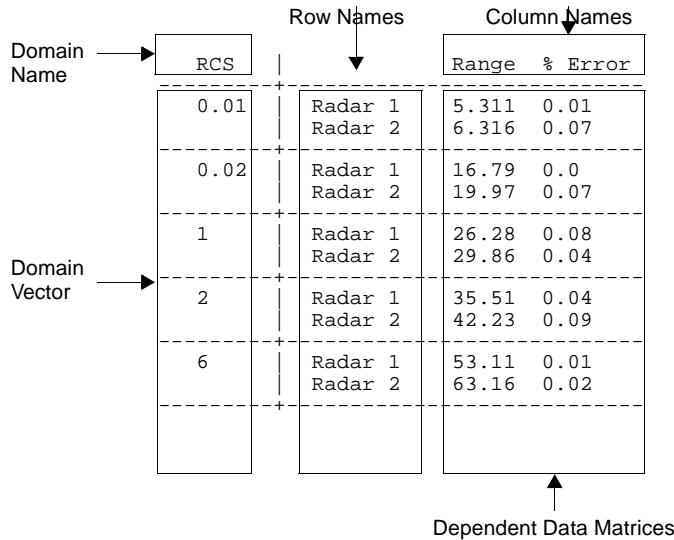
So, for example, when the frequency response of a system with  $n$  inputs and  $m$  outputs is calculated, a PDM is created. Each of the  $n$  columns represents an input, each of  $m$  rows represents an output, and the dependent matrix at element  $i$  of the domain corresponds to the frequency response from each output to each input. Plotting time and frequency responses stored as PDMs are particularly convenient when the `{strip}` keyword is used, in which case a matrix plot is produced where the rows and columns correspond with inputs and outputs, respectively (for information on strip plots, see [4.3.15 Strip Plots](#), p.145). For an explanation of time response, see [5.5.5 Time Response](#), p.234.



### 5.4.1 PDM Organization

Consider the object **radar** as an example of PDM organization (Figure 5-3). Exactly how **radar** is created is outlined in 5.4.2 *Creating PDMs*, p.210.

Figure 5-3 Parts of the PDM radar



Every PDM consists of five main parts:

**Dependent Data Matrix** — Every PDM contains one or more matrices; **radar** has five  $2 \times 2$  matrices in the dependent data area. The matrices must be the same size. There is no limit to the size or number of matrices in this area.

**Domain Vector** — The PDM allows you to group an independent vector of parameter values and a stack of associated matrices. The vector of independent parameter values is called the *domain* of the PDM. The domain usually represents a physical parameter, for example, time, frequency, temperature, pressure, or altitude. If no domain vector is specified, the PDM domain defaults to increasing positive integers starting from one.

**Domain Name** — A label for the domain vector. In **radar**, the domain string is "RCS". If no name is specified, the default string is "domain".

**Row Names** — Each dependent matrix row may have an optional string name. In **radar**, the names are "Radar 1" and "Radar 2". Each matrix has the same row

names associated with it. If no names are specified, the row names are labeled "Row 1", "Row 2", ... "Row N".

**Column Names** — Each dependent matrix column may have an optional string name. In **radar**, the names are "Range" and "% Error". If no column names are specified, the columns are labeled "Col 1", "Col 2", ... "Col N".

## 5.4.2 Creating PDMs

PDMs are created from a single matrix object using the function **pdm()**. Additional optional arguments to **pdm()** specify the domain, domain label, and row and column labels to be associated with the matrices in the final PDM.

For the PDM **radar**, the dependent data is formed from a columnwise concatenation of the vectors **maxrange** and **perr**:

```
maxrange=[ 5.311, 6.313, 16.79, 19.97, 26.28, 29.86, 35.51, 42.23,  
          53.11, 63.16]';  
perr = [0.01, 0.07, 0.0, 0.07, 0.08, 0.04, 0.04, 0.09, 0.01, 0.02]';
```

The final dependent data matrix [**maxrange,perr**] used as an argument to **pdm()** has 2 columns and 10 rows.

The domain vector used in **radar**, **rcs**, has 5 elements.

```
rcs = [0.01,0.02,1,2,6];
```

Use the **pdm()** function to construct the PDM **radar** from the matrix [**maxrange,perr**] and the domain vector **RCS**:

```
radar = pdm([maxrange,perr],rcs,{domainName="RCS",  
        rowNames = ["Radar 1","Radar 2"],  
        columnNames = ["Range","% Error"]})
```

```
radar (a pdm) =
```

RCS		Range	% Error
0.01	Radar 1	5.311	0.01
	Radar 2	6.313	0.07
0.02	Radar 1	16.79	0
	Radar 2	19.97	0.07
1	Radar 1	26.28	0.08
	Radar 2	29.86	0.04
2	Radar 1	35.51	0.04
	Radar 2	42.23	0.09

```

-----+-----
6 | Radar 1  53.11  0.01
  | Radar 2  63.16  0.02
-----+-----

```

You have just recreated the PDM shown on page 209.

The dependent matrix is the only required argument to a PDM. Any additional arguments can modify the structure of the PDM. For example, using `pdm()` with no optional arguments results in a PDM with each dependent matrix having one row.

### 5.4.3 Default PDM Behavior

If you do not use the `rows` or `columns` keywords and do not specify a domain vector, each row of the input matrix becomes one of the output dependent data matrices. For example:

```

r43=rand(4,3)

r43 (a rectangular matrix) =

0.849745    0.685731    0.878216
0.068374    0.560849    0.662357
0.726351    0.198514    0.544257
0.232075    0.231224    0.216463

pdm(r43)

ans (a pdm) =

domain |   Col 1   Col 2   Col 3
-----+-----
    1 | 0.849745 0.685731 0.878216
    2 | 0.068374 0.560849 0.662357
    3 | 0.726351 0.198514 0.544257
    4 | 0.232075 0.231224 0.216463

```

This default behavior also applies if any or all of the `rows` or `columns` keywords, or domain vector, are specified in a way that matches the default case. For example, Xmath generates the same PDM output (the `rows` and `columns` keywords are ignored in this case):

```

pdm(r43,{rows=1,columns=3})

ans (a pdm) =

domain |   Col 1   Col 2   Col 3
-----+-----
    1 | 0.849745 0.685731 0.878216

```

```

2 | 0.068374 0.560849 0.662357
3 | 0.726351 0.198514 0.544257
4 | 0.232075 0.231224 0.216463

```

**pdm(r43,1:4)**

ans (a pdm) =

```

domain | Col 1 Col 2 Col 3
-----+-----
1 | 0.849745 0.685731 0.878216
2 | 0.068374 0.560849 0.662357
3 | 0.726351 0.198514 0.544257
4 | 0.232075 0.231224 0.216463

```

If you specify arguments that deviate from the default, other PDMs are obtained:

**pdm(r43,1:3)**

ans (a pdm) =

```

domain |
-----+-----
1 | Row 1 0.849745
   | Row 2 0.068374
   | Row 3 0.726351
   | Row 4 0.232075
-----+-----
2 | Row 1 0.685731
   | Row 2 0.560849
   | Row 3 0.198514
   | Row 4 0.231224
-----+-----
3 | Row 1 0.878216
   | Row 2 0.662357
   | Row 3 0.544257
   | Row 4 0.216463
-----+-----

```

In the above example, the number of rows of the input matrix (4) is not a multiple of the length of the domain vector (3). However, the number of columns of the input matrix (3) is a multiple. In this case, each column (instead of each row) of the input matrix becomes one of the output Dependent Data Matrices.

When no domain vector is specified, the default vector is [1:1:#rows].

**pdm([maxrange,perr])**

ans (a pdm) =

```

domain | Col 1 Col 2
-----+-----
1 | 5.311 0.01
2 | 6.313 0.07

```

3	16.79	0
4	19.97	0.07
5	26.28	0.08
6	29.86	0.04
7	35.51	0.04
8	42.23	0.09
9	53.11	0.01
10	63.16	0.02

To change the dimensions of the dependent matrices, use the **rows** and **columns** keywords. For example:

```
pdm([maxrange,perr], {rows = 2, columns = 2})
```

ans (a pdm) =

domain		Col 1	Col 2
-----			
1	Row 1	5.311	0.01
	Row 2	6.313	0.07
-----			
2	Row 1	16.79	0
	Row 2	19.97	0.07
-----			
3	Row 1	26.28	0.08
	Row 2	29.86	0.04
-----			
4	Row 1	35.51	0.04
	Row 2	42.23	0.09
-----			
5	Row 1	53.11	0.01
	Row 2	63.16	0.02
-----			

Alternatively, the row and column size is implied in the number of strings entered in keywords **columnnames** and **rownames**:

```
pdm([maxrange,perr],{rowNames = ["Radar 1","Radar 2"],
columnNames = ["Range","% Error"]})
```

ans (a pdm) =

domain		Range	% Error
-----			
1	Radar 1	5.311	0.01
	Radar 2	6.313	0.07
-----			
2	Radar 1	16.79	0
	Radar 2	19.97	0.07
-----			
3	Radar 1	26.28	0.08
	Radar 2	29.86	0.04
-----			
4	Radar 1	35.51	0.04

```

      | Radar 2  42.23  0.09
-----+-----
    5 | Radar 1  53.11  0.01
      | Radar 2  63.16  0.02
-----+-----

```

The dependent matrix size can also be influenced by the domain vector. In the following example, the columns of the PDM matrices are the same as the input matrix. The number of rows of each PDM matrix is equal to the total number of rows in the input matrix divided by the number of elements in the domain vector. The domain `rcs` has 5 elements, and the input matrix has 10 rows. Therefore, each PDM matrix has 10/5 (=2) rows.

```

pdm([maxrange, perr], rcs)

ans (a pdm) =

domain |          Col 1  Col 2
-----+-----
  0.01 | Row 1   5.311  0.01
      | Row 2   6.313  0.07
-----+-----
  0.02 | Row 1   16.79   0
      | Row 2   19.97  0.07
-----+-----
    1  | Row 1   26.28  0.08
      | Row 2   29.86  0.04
-----+-----
    2  | Row 1   35.51  0.04
      | Row 2   42.23  0.09
-----+-----
    6  | Row 1   53.11  0.01
      | Row 2   63.16  0.02
-----+-----

```

The PDM row and column dimensions specified by `rows`, `rowNames`, `columns`, and `columnNames` must agree with the PDM dimensions specified by the domain vector, or an error message is returned:

```

pdm(r43, {rows=1, columns=3})

```

```

Dimensions of PDM do not match specified rows and columns and length of domain
vector

```

#### 5.4.4 PDM Channels

In some circumstances, a PDM is a collection of vectors instead of a collection of matrices. For PDMs, these vectors are called *channels* of the PDM. A channel is a

vector consisting of the same element from each dependent matrix. For example, **radar** has four channels,

```
(1,1) : 5.311, 16.79, 26.28, 35.51, 53.11  
(2,1) : 6.313, 19.97, 29.86, 42.23, 63.16  
(1,2) : 0.01, 0, 0.08, 0.04, 0.01  
(2,2) : 0.07, 0.07, 0.04, 0.09, 0.02
```

and all channels have the common independent variable defined by **rsc**. [Figure 5-2](#) illustrates this idea.

Certain MathScript functions, such as **fft()**, have the option of operating on the dependent matrices or the channels of a PDM. By default, all functions operate on the dependent matrices.

```
Y = fft(radar)
```

If the FFT of each channel is needed, the **channels** keyword must be included.

```
Y = fft(radar, {channels})
```

See [5.4.8 Using Functions with PDMs](#), p.225 for more details on using functions with PDMs.

## 5.4.5 Indexing to Extract Portions of a PDM

### *PDM Dimensions*

Use the **size()** function to see the dimensions of the new PDM:

```
size(radar)  
  
ans (a row vector) = 2 2 5
```

The above result indicates that each dependent matrix has two rows and two columns, and that the length of the PDM (the length of the domain or the number of dependent matrices) is five.

### *Dependent Matrices*

PDM indexing allows you to extract parts of a PDM. The output of any PDM indexing operation is always another PDM. If you want to index to extract a single piece of data (as opposed to a dependent matrix or a channel of a PDM) it may be simpler to use **makematrix()** before indexing (see [p.222](#)).

To extract a single dependent matrix, use a single index corresponding to the domain value of interest. For example, you might want to extract only the data pertaining to objects with RCS value of 1:

**radar(3)**

ans (a pdm) =

RCS		Range	% Error
1	Radar 1	26.28	0.08
	Radar 2	29.86	0.04

To see the third through fifth elements of the PDM, you could index into **radar** using the standard colon notation (see [p.194](#)):

**radar(3:5)**

ans (a pdm) =

RCS		Range	% Error
1	Radar 1	26.28	0.08
	Radar 2	29.86	0.04
2	Radar 1	35.51	0.04
	Radar 2	42.23	0.09
6	Radar 1	53.11	0.01
	Radar 2	63.16	0.02

You can also examine one or more channels of the data in a PDM and see changes over the length of the PDM (as the **RCS** parameter changes). When indexing with both row and column specifications, you extract the  $(i,j)$  channel over the entire domain. The following example extracts the element that resides in the second row and first column of each dependent matrix.

**radar(2,1)**

ans (a pdm) =

RCS		Range
0.01	Radar 2	6.313
0.02	Radar 2	19.97
1	Radar 2	29.86
2	Radar 2	42.23
6	Radar 2	63.16

The standard colon notation can be used to access more than one channel:



```

radar(1:2,1)

ans (a pdm) =

  RCS |           Range
  -----+-----
0.01 | Radar 1   5.311
      | Radar 2   6.313
-----+-----
0.02 | Radar 1  16.79
      | Radar 2  19.97
-----+-----
1    | Radar 1  26.28
      | Radar 2  29.86
-----+-----
2    | Radar 1  35.51
      | Radar 2  42.23
-----+-----
6    | Radar 1  53.11
      | Radar 2  63.16
-----+-----

```

To extract a single value in PDM form, you can use a temporary value:

```

temp=radar(5);
FinalPerr=temp(2,2)

FinalPerr (a pdm) =

  RCS |           % Error
  -----+-----
  6 | Radar 2   0.02

```

Individual PDM elements can be extracted and modified using three scalar indices to specify the row, column, and domain positions, respectively. The returned object is always a scalar. Thus, for the radar example:

```

radar(1,1,1)

ans (a scalar) =   5.311

radar(2,2,5)

ans (a scalar) =   0.02

radar(2,1,3)=radar(1,1,5)

radar (a pdm) =

  RCS |           Range   % Error
  -----+-----
0.01 | Radar 1   5.311   0.01
      | Radar 2   6.313   0.07
-----+-----
0.02 | Radar 1  16.79    0

```

	Radar 2	19.97	0.07
1	Radar 1	26.28	0.08
	Radar 2	53.11	0.04
2	Radar 1	35.51	0.04
	Radar 2	42.23	0.09
6	Radar 1	53.11	0.01
	Radar 2	63.16	0.02

### Domain and Name Information

The domain can be extracted using `domain()`.

```
rsvector = domain(radar)
```

```
rsvector (a row vector) = 0.01 0.02 1 2 6
```

The PDM names can be extracted with the `names()` function. In order to get all three labels, specify three outputs:

```
[rowN,colN,domN]=names(radar)
```

```
rowN (a row vector of strings) = Radar 1 Radar 2
```

```
colN (a row vector of strings) = Range % Error
```

```
domN (a string) = RCS
```

#### Example 5-1 Indexing into a PDM

This example illustrates PDM indexing by plotting a PDM and different combinations of data that can be extracted from it. Note that `plot()` will reuse the row and column labels from your PDM, if possible.

```
x=logspace(1,100,3);F=[1.02:.02:2.5];
s1c=system(makep([sin(x)]),makep(-x*2));
s2c=system(makep([cos(x)]),makep(x*2));
s3c=system(makep([cot(x)]),makep(x));
```

```
s1d=discr(s1c,1);
s2d=discr(s2c,2);
s3d=discr(s3c,3);
```

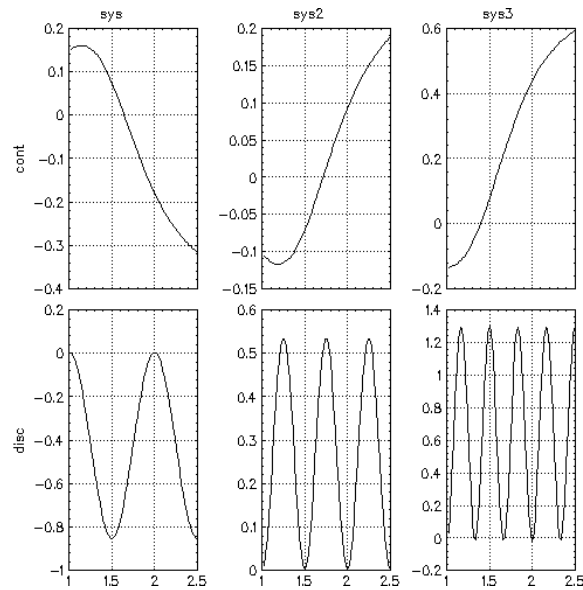
```
f1c=freq(s1c,F);f1d=freq(s1d,F);
f2c=freq(s2c,F);f2d=freq(s2d,F);
f3c=freq(s3c,F);f3d=freq(s3d,F);
```

```
p=pdm([[f1c;f1d],[f2c;f2d],[f3c;f3d]],
```

```
{columnnames=["sys","sys2","sys3"], rownames=["cont","disc"]});
plot(p,{strip})
```

If `strip` is specified alone, each submatrix is plotted in a separate subgraph, as shown in [Figure 5-4](#). Try plotting portions of the PDM with the different `strip` settings shown below.

Figure 5-4 **PDM Plotted with strip**



If the number of strips is specified, the inputs will be plotted accordingly.

```
plot(p,{strip=3})
```

Extract all discrete rows, then plot one plot per subgraph:

```
plot(p(2,:),{strip=1})
```

Plot all rows of the 2nd column with default strip settings.

```
plot(p(:,2),{strip})
```

## 5.4.6 Modifying PDMs

### Substitution

Using PDM indexing (outlined in [5.4.5 Indexing to Extract Portions of a PDM](#), p.215), assignments can be made to replace parts of a PDM. For example, to replace the third dependent matrix of radar with an identity matrix, type:

```
ind = eye(2,2); radar_copy = radar; radar_copy(3) = ind
```

```
radar_copy (a pdm) =
```

RCS		Range	% Error
0.01	Radar 1	5.311	0.01
	Radar 2	6.313	0.07
0.02	Radar 1	16.79	0
	Radar 2	19.97	0.07
1	Radar 1	1	0
	Radar 2	0	1
2	Radar 1	35.51	0.04
	Radar 2	42.23	0.09
6	Radar 1	53.11	0.01
	Radar 2	63.16	0.02

To replace a channel of data, type:

```
ind = [10,20,30,40,50];  
radar_copy(1,1) = ind
```

```
radar_copy (a pdm) =
```

RCS		Range	% Error
0.01	Radar 1	10	0.01
	Radar 2	6.313	0.07
0.02	Radar 1	20	0
	Radar 2	19.97	0.07
1	Radar 1	30	0
	Radar 2	0	1
2	Radar 1	40	0.04
	Radar 2	42.23	0.09
6	Radar 1	50	0.01
	Radar 2	63.16	0.02

### Concatenation

Compatible PDMs can be concatenated in the same manner as matrices. A comma results in right concatenation, and a semicolon results in bottom concatenation.

```
new_radar =[radar,radar(1:2,1)^2]
```

```
new_radar (a pdm) =
```

RCS		Range	% Error	Range
0.01	Radar 1	5.311	0.01	28.2067
	Radar 2	6.313	0.07	39.854
0.02	Radar 1	16.79	0	281.904
	Radar 2	19.97	0.07	398.801
1	Radar 1	26.28	0.08	690.638
	Radar 2	29.86	0.04	891.62
2	Radar 1	35.51	0.04	1260.96
	Radar 2	42.23	0.09	1783.37
6	Radar 1	53.11	0.01	2820.67
	Radar 2	63.16	0.02	3989.19

## Converting PDMs to Matrices

The `makeMatrix()` function converts a PDM into a matrix by discarding the independent parameter (domain) and right concatenating the dependent matrices columnwise. If a PDM is an argument to `makematrix()`, a matrix containing all dependent matrix data is returned:

```
radar_mx = makematrix(radar)

radar_mx (a rectangular matrix) =

      5.311    0.01    16.79    0      26.28    0.08    35.51    0.04 ...
      6.313    0.07    19.97    0.07    29.86    0.04    42.23    0.09 ...
```

All **Radar 1** values are right-concatenated to form the first row, and all **Radar 2** values appear in the second row.

To create a matrix formatted in the same manner as the dependent matrix elements in `radar`, transpose the PDM (this transposes each dependent matrix separately for each domain element), then transpose the result as shown below. Compare this result to `radar` and `radar_mx`.

```
radar_mxTrans = makematrix(radar')'

radar_mxTrans (a rectangular matrix) =

      5.311    0.01
      6.313    0.07
      16.79     0
      19.97    0.07
      26.28    0.08
      29.86    0.04
      35.51    0.04
      42.23    0.09
      53.11    0.01
      63.16    0.02
```

When the `channels` keyword is used, rows of each dependent matrix are right-concatenated to form rows in the resulting matrix:<sup>1</sup>

```
radar_mxChan = makematrix(radar,{channels})

radar_mxChan (a rectangular matrix) =

      5.311  0.01      6.313  0.07
      16.79  0      19.97  0.07
      26.28  0.08      29.86  0.04
```

---

1. This feature can be used to convert time and frequency responses to a format similar to that used in MATRIX<sub>X</sub>.

```

35.51  0.04  42.23  0.09
53.11  0.01  63.16  0.02

```

Sections of a PDM can also be used as an input to **makematrix()**. This makes it easy to extract a desired value. For example, to see the range for **Radar 2** at 0.01:

```

temp=makematrix(radar(1))

temp (a square matrix) =

    5.311    0.01
    6.313    0.07

temp(2,1)

ans (a scalar) = 6.313

```

The **SAVE** command also has the ability to create matrices from PDMs. When **SAVE** is called with the **matrixx** keyword, all saved PDMs are stored as two matrices. The domain is given the name **pdmName\_t** and the dependent matrix data is given the name **pdmName\_u**, where **pdmName** is the name of the original PDM. This handling is designed to map to simulation data.

### 5.4.7 Using PDMs with Operators

Operators defined for matrices are also defined for PDMs. For example, the square of each element in the first dependent matrix of **radar** can be calculated by:

```

radar(1)^2

ans (a pdm) =

RCS |           Range    % Error
-----+-----
0.01 | Radar 1  28.2699  0.05381
      | Radar 2  33.9703  0.06803
-----+-----

```

Notice the output is also a PDM.

Operations between two PDMs are defined such that the operation is performed elementwise on each pair of corresponding matrices. *These operations are restricted to PDMs with identical dimensions.*

For example, the average value of **Row 1** and **Row 2** is calculated by:

```

(radar(1,1) + radar(2,1))/2

RCS |
-----+-----

```

```
0.01 | Radar 1  5.812
0.02 | Radar 1 18.38
1     | Radar 1 28.07
2     | Radar 1 38.87
6     | Radar 1 58.135
```

Operators can also be used between matrix objects (including vectors and scalars as well as matrices) and PDMs. In this case, *the operation is performed between the matrix object and each dependent matrix in the PDM*. The result of the operation is a PDM with the same domain as the PDM operand.

For example, the identity matrix is added to each dependent matrix using the expression:

```
radar + eye(2,2)
```

```
ans (a pdm) =
```

```

RCS |           Range  % Error
-----+-----
0.01 | Radar 1  6.311  0.01
      | Radar 2  6.313  1.07
-----+-----
0.02 | Radar 1 17.79  0
      | Radar 2 19.97  1.07
-----+-----
1     | Radar 1 27.28  0.08
      | Radar 2 29.86  1.04
-----+-----
2     | Radar 1 36.51  0.04
      | Radar 2 42.23  1.09
-----+-----
6     | Radar 1 54.11  0.01
      | Radar 2 63.16  1.02
-----+-----
```

A scalar value can also be used in operators with a PDM. The operation will be applied to each matrix element and the scalar.

```
5.0 * radar(1)
```

```
ans (a pdm) =
```

```

RCS |           Range  % Error
-----+-----
0.01 | Radar 1 26.555  0.05
      | Radar 2 31.565  0.35
-----+-----
```



### 5.4.8 Using Functions with PDMs

When a PDM is used as an input to a function, the function is applied to each dependent matrix (Figure 5-5). If the **channels** keyword is available and is used, the function will be applied to each channel.

Figure 5-5 Functions of PDMs

$$\begin{array}{c}
 \text{ypdm} = f(\text{xpdm}) \\
 \left[ \begin{array}{c} [\text{ypdm}_1] \\ [\text{ypdm}_2] \\ \downarrow \\ [\text{ypdm}_n] \end{array} \right] = \left( \begin{array}{c} [f(\text{xpdm}_1)] \\ [f(\text{xpdm}_2)] \\ \downarrow \\ [f(\text{xpdm}_n)] \end{array} \right)
 \end{array}$$

For example, if **xpdm** is a step response of a system with  $n$  inputs,  $m$  outputs over  $p$  time points, then  $y = \text{max}(\text{xpdm})$  is a  $(1 \times 1) \times p$  PDM whose  $k^{\text{th}}$  element contains the maximum element of the  $k^{\text{th}}$  matrix in **xpdm** (the maximum output for every time point).<sup>2</sup> The result of any function that accepts the **channels** keyword is always a matrix the size of the dependent matrices in the PDM (see Figure 5-6).

PDMs use optimized internal looping to speed up the total computation time. Therefore, using a single PDM as a function input is much more efficient than looping through a set of separate matrices with MathScript commands.

Next you will use the intrinsic function **max()** to illustrate the flexibility of PDMs. **max()** finds the maximum over a specified subset of the PDM data.

2. Where **max(xpdm, {channels})** is an  $n \times m$  matrix where  $(i,j)$  element is the maximum of the vector of the  $(i,j)$  elements of all the dependent matrices.

Figure 5-6 Functions of a PDM Over Channels

$$\begin{aligned}
 & \mathbf{y} = \mathbf{f}(\mathbf{x}_{\text{pdm}}, \{\text{channels}\}) \\
 & \begin{bmatrix} f(v_{11}(d_1:d_p)) \dots f(v_{n1}(d_1:d_p)) \\ \vdots \\ f(v_{1m}(d_1:d_p)) \dots f(v_{nm}(d_1:d_p)) \end{bmatrix} = f \left( \begin{array}{c} \begin{bmatrix} (v_{11}) & \dots & (v_{n1}) \\ \vdots & & \vdots \\ (v_{1m}) & \dots & (v_{nm}) \end{bmatrix} d_1 \\ \downarrow \\ \begin{bmatrix} (v_{11}) & \dots & (v_{n1}) \\ \vdots & & \vdots \\ (v_{1m}) & \dots & (v_{nm}) \end{bmatrix} d_p \end{array} \right)
 \end{aligned}$$

To find the maximum range for both **Radar 1** and **Radar 2** over all **RCS** values, apply the function to all rows of the first column of dependent matrices. Type:

```
maxrad = max(radar(:,1))
```

```
maxrad (a pdm) =
```

RCS	
0.01	6.313
0.02	19.97
1	29.86
2	42.23
6	63.16

**max()** treats the PDM as a series of matrices, returning a PDM with the same domain as **radar**. It loops over all the domain points (values of **RCS**), finds the largest value each dependent vector contains (in this case, the **Range** value), and returns that scalar value as the dependent matrix corresponding to the same domain point in the output PDM.

You might want to know the maximum ranges for **Radar 1** and **Radar 2** separately. In this case, the PDM is treated as a matrix of vectors, each corresponding to a channel of the PDM. To use **max()** in this manner, invoke the **channels** keyword:

```
maxvals = max(radar(1:2,1), {channels})
```

```
maxvals (a column vector) =
  53.11
  63.16
```

The range for **Radar 1** corresponds to the (1,1) channel, and the range for **Radar 2** corresponds to the (2,1) channel. The (1,1) element of the output matrix, 53.11, is the maximum value for the range of **Radar 1** over all the **RCS** values. The second element is the maximum value for **Radar 2**.

## 5.5 Dynamic System

The dynamic system class represents systems of time-dependent equations for modeling input/output relationships. In general, there are many different kinds of dynamic systems, with many different representations.

Xmath supports linear, time-invariant systems. These can be continuous (systems of differential equations) or discrete (systems of difference equations). Two specific representations are provided: state-space systems and transfer functions. Both are created with the `system()` function and are discussed later. Sampling times (0 for continuous systems and nonzero for discrete) are automatically stored within a dynamic system object.

The dynamic system class is closely tied to the PDM class. Simulations or dynamic systems are defined using a PDM to represent inputs, and return a PDM representing the outputs. The `*` (product) operator has also been overloaded (defined) such that `system*input_pdm` performs a simulation over the data in `input_pdm`.

### 5.5.1 State-Space Systems

A state-space dynamic system stores the  $A$ ,  $B$ ,  $C$ , and  $D$  matrices associated with the following equation:

for continuous systems	for discrete systems
$\frac{dx}{dt} = Ax + Bu$	$x_{k+1} = Ax_k + Bu_k$
$y = Cx + Du$	$y_k = Cx_k + Du_k$

$x$  is the state vector (with initial conditions  $X_0$ ),  $u$  is the input vector, and  $y$  is the output vector. All matrices are stored, even if they are null.

- State-space systems can be single-input/single-output (SISO) or multiple-input/multiple-output (MIMO).
- Names can be attached to each of the inputs and outputs and states of a state-space system. This capability is particularly useful with MIMO systems.

## 5.5.2 Transfer Functions

A transfer function is described as:

for continuous systems	for discrete systems
$\frac{y(s)}{u(s)} = H(s) = \frac{\text{num}(s)}{\text{den}(s)}$	$\frac{y(z)}{u(z)} = H(z) = \frac{\text{num}(z)}{\text{den}(z)}$

The notations  $H(s)$  and  $H(z)$  are common for transfer functions.  $s$  represents the Laplace transform variable, and  $z$  represents the z-transform variable. A transfer function represents a dynamic system in terms of numerator and denominator polynomials.

- A transfer function is *proper* if the order of the numerator is less than or equal to the order of the denominator.
- It may sometimes be convenient to use an *improper* or *noncausal* transfer function (to represent an ideal differentiator, for example). Xmath allows you to define an improper transfer function, but restricts the types of analyses you can perform. You can find the frequency response of an improper transfer function, but not the time response. An improper transfer function cannot be connected with state-space systems or converted to state space form.
- Currently, only SISO transfer functions are supported.
- Names can be attached to the inputs and outputs of a system in transfer-function form.
- To perform a time-domain simulation ( $\text{Sys} \times u$ ), multiply a system by a PDM whose columns contain the input vector(s) for the simulation(s). (See [5.5.5 Time Response](#), p.234).

### 5.5.3 Creating Systems

Dynamic systems can be created with the `system()` function. If four compatibly-sized matrices are given as inputs, a state-space system is formed.

```
a=[1,2;3,4]; b=[.1,-.1,1; 2,-.2,2]; c=[3,3]; d=[.4,-.4,4];
ssSys=system(a,b,c,d, {inputNames=["red","white","blue"],
    outputNames=["Flag"], stateNames=["Alaska","Nebraska"],dt=.01})
```

```
ssSys (a state space system) =
```

```
A
1  2
3  4

B
0.1  -0.1  1
2    -0.2  2

C
3    3

D
0.4  -0.4  4

X0
0
0
```

```
State Names
-----
Alaska  Nebraska
```

```
Input Names
-----
red
white
blue
```

```
Output Names
-----
Flag
```

```
System is discrete, sampling at 0.01 seconds.
```

A handy shortcut for creating state-space systems with an all-zero D matrix is to use a NULL-matrix specifier (`[]`) for the D matrix. This automatically sets the D matrix to a zero matrix, with row size equal to the row size of C and column size equal to the column size of B.

If `dt` was not given a value, `ssSys` would have been continuous (`dt` defaults to 0).

The size of a system object is defined by the number of outputs, inputs, and (in the case of a state-space system) the number of states it has. You can use the `size()` function to find these dimensions.

```
[out,in,states]=size(ssSys)?  
  
out (a scalar) = 1  
  
in (a scalar) = 3  
  
states (a scalar) = 2
```

If a pair of polynomials is given, a transfer function results:

```
n=makepoly(polynomial([1,-1;2,-2],"s"));  
d=polynomial([-2,1;1,-2],"s");  
tfSys=system(n,d,{inputNames="In", outputnames="Out"})  
  
tfSys (a transfer function) =  
  
      s(s + 1)  
-----  
  
      (s + 1)(s + 3)  
  
initial integrator outputs  
0  
0  
Input Names  
-----  
In  
  
Output Names  
-----  
Out  
  
System is continuous
```

The various parts of a transfer function or a state-space system can be extracted with the `abcd()`, `numden()`, `period()`, and `names()` functions (see the MATRIX<sub>X</sub> online Help).

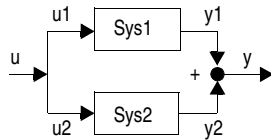
### Using Operators with Dynamic Systems

Operators have also been defined to perform connections between dynamic systems. Suppose you have dynamic systems `Sys1` and `Sys2`, where outputs are `y1` and `y2` and inputs are `u1` and `u2`, respectively. The statements in [Table 5-4](#) would then be true.

Table 5-4 Operations on Dynamic Systems

$\text{Sys} = \text{Sys1} + \text{Sys2}$

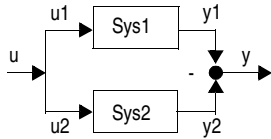
Defined such that  $y = y1 + y2$ .



The inputs are tied together such that  $u=u1=u2$ .

$\text{Sys} = \text{Sys1} - \text{Sys2}$

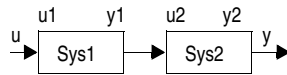
Defined such that  $y = y1 - y2$ .



In the unary case,  $\text{Sys} = -\text{Sys2}$  is defined such that  $y = -y2$  ( $\text{Sys1}=\text{system}([ \ ],[ \ ],[ \ ],[ \ ])$ ).

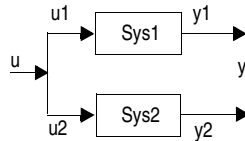
$\text{Sys} = \text{Sys2} * \text{Sys1}$

The cascade connection of  $\text{Sys1}$  followed by  $\text{Sys2}$ . The output of  $\text{Sys}$  is  $y2$  and the input is  $u1$ .



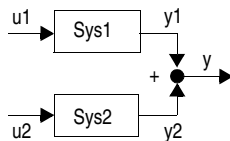
$\text{Sys} = [\text{Sys1};\text{Sys2}]$

Defined such that  $y = [y1;y2]$  and  $u=u1=u2$  (inputs are tied together).



$\text{Sys} = [\text{Sys1},\text{Sys2}]$

Defined such that  $y = y1 + y2$  and  $u = [u1;u2]$ .



### Creating Subsystems by Indexing into Dynamic Systems

You can index into a dynamic system to create a subsystem comprising a subset of the original inputs and outputs, as shown in [Table 5-5](#).

Table 5-5 Indexing Into a Dynamic System

---

<b>Sys = Sys1(i,j)</b>	Defined to be a system such that <b>y=y1(i)</b> and <b>u=u1(j)</b> . <b>i</b> and <b>j</b> can both be vectors as well, in which case multiple inputs and outputs will be extracted.
------------------------	--

---

If you are familiar with input/output notation, you may feel that the above definition (outputs first, inputs second) of indexing seems reversed. It was designed with the traditional definition of a transfer function in mind, where outputs are specified first:  $y(s) = Sys(s) \times u(s)$ . (This definition also led to Xmath's definition of  $Sys \times aPDM$  to perform simulation, since in that case  $y(t) = Sys \times u(t)$ ). For a MIMO system with  $m$  outputs and  $n$  inputs,  $y$  is an  $m \times 1$  vector and  $u$  is  $n \times 1$ ; thus, it makes sense for  $Sys$  to be  $m \times n$ . We can see this if we index into `ssSys` from p.229:

**Sys2=ssSys(1,3)**

Sys2 (a state space system) =

```
A
1   2
3   4
```

```
B
1
2
```

```
C
3   3
```

```
D
4
```

```
X0
0
0
```

```
State Names
-----
Alaska   Nebraska
```

```
Input Names
-----
blue
```

```
Output Names
-----
Flag
```

System is discrete, sampling at 0.01 seconds.



The output is a SISO dynamic system containing the third column of the B and D matrices.

### 5.5.4 Functions for Manipulating Dynamic System Objects

Table 5-6 briefly describes functions commonly used to manipulate systems. To see a full description of each function, see the MATRIX<sub>X</sub> online Help.

Table 5-6 Functions Commonly Used to Manipulate Systems

<b>abcd()</b>	<p>Extracts the component A, B, C, and D matrices from a state-space system object. In addition, it returns the initial conditions on the states if a fifth output argument is requested.</p> <p><b>abcd()</b> can be called on systems in either state-space or transfer-function form. If the system is a transfer function, the conversion to state-space is done internally to return A, B, C, and D, although the format of the variable itself remains unchanged. The transfer function must be proper.</p>
<b>discretize()</b>	Converts a continuous system to discrete form.
<b>makecontinuous()</b>	Converts a discrete system to continuous form.
<b>numden()</b>	<p>Returns the numerator and denominator polynomials comprising a SISO system in transfer function form. If the system is in state-space form, an internal conversion is performed to find the transfer function equivalent, but the format of the system variable itself remains unchanged. State-space systems used as inputs to <b>numden()</b> must be SISO. Note that common roots in the numerator and denominator polynomials are not canceled.</p>
<b>period()</b>	<b>period()</b> extracts the sample period (in seconds) of a system. If the system is continuous, <b>period()</b> returns zero.
<b>names()</b>	Extracts matrices of strings representing the input, output, and (if the system is in state-space form) state names of a system. It works much the same as described for PDMs on <a href="#">p.218</a> .
<b>check()</b>	<p>Can be used to return a Boolean indication of whether a system is in transfer-function or state-space form, discrete, continuous, or stable. In addition, <b>check</b> can be used with the <b>convert</b> keyword to change a system's representation between SISO state-space and transfer-function forms.</p>

### 5.5.5 Time Response

The behavior of a dynamic system as a function of time in response to external stimuli is referred to as the system's *time response*. Xmath can simulate the response of a dynamic system to various inputs to obtain the system's time response. This is accomplished with the \* operator between dynamic systems and parameter-dependent matrices (PDMs) and with one or more of the functions in [Table 5-7](#).

Table 5-7 Time Response Functions

<b>impulse()</b>	Computes the impulse response of a system.
<b>initial()</b>	Computes the unforced response of a system to a given initial condition.
<b>step()</b>	Computes the step response of a system.
<b>defTimeRange()</b>	Computes a default time vector for simulations.

Borrowing from the convenient frequency response notation for a system where  $y(s) = H(s)*u(s)$ , Xmath defines the operation **system \*PDM** as a time domain simulation. Thus, for any dynamic system **Sys** (continuous or discrete) and for a PDM **u** representing the external stimulus as a function of time, the operation **y=Sys\*u** creates a PDM **y** that contains the outputs of the system as a function of time.

For a dynamic system with  $n_y$  outputs and  $n_u$  inputs, the input vector is defined to be  $n_u \times 1$  and the output vector is  $n_y \times 1$ . Thus, the input PDM  $u$  should be  $n_y \times 1 \times N_{samp}$ , where  $N_{samp}$  is the number of time points in  $u$ .

- The input PDM must have a regular domain.
- If the system is discrete, the domain intervals must be equal to the system's sampling period.
- If the system is continuous, it is discretized using the exponential (zero-order hold) method, with the sampling interval set equal to the input domain interval spacing. For accurate results, make sure this sampling interval is small enough that discretization effects are negligible.

If you desire to run several simulations with different inputs, you can define a PDM where columns contain the input vectors for the different simulations. Then **u** will be  $n_y \times q \times N_{samp}$ , where  $q$  is the number of different simulations to be run. The resulting **y** will be  $n_y \times q \times N_{samp}$ , with each column of the PDM corresponding to a different simulation.

See [4.3.15 Strip Plots](#), p.145 for an explanation of how PDMs are plotted.

## 5.6 Strings

A string object is a sequence of characters enclosed by double quotes. To be recognized as a string, an object must be created with double quotes or be the output of the `string()` function, which converts numbers to strings.

- You can concatenate strings with the plus (+) operator.

```
c="California";s="Sacramento";
str="\nThe capital of "+c+ " is "+s+ "."

str (a string) =
The capital of California is Sacramento.
```

- You can concatenate strings and then use them on the Xmath command line.

```
alias mypath "C:/myhomedir/myexamples/"
display mypath + "engine"
execute file = mypath + "engine"
```

- You can group multiple strings into string matrices (also called *tables*) using the same punctuation as matrices.

```
r=" rest"; i=" ice"; c=" compression"; e=" elevation";
rice=[r,i,c,e];ouch=[82,73,67,69];
sport=[char(ouch)',rice']
```

```
sport (a rectangular matrix of strings) =
```

```
R    rest
I    ice
C    compression
E    elevation
```

- For strings, `size()` returns the number of rows and columns of the whole string matrix.

```
size(sport)

ans (a row vector) =    4    2
```

To find the total number of elements (characters) in a string, use `length()`.

```
length(sport)
ans (a scalar) = 35
```

### 5.6.1 Converting Strings and Numbers

Numbers can be converted to strings using the `string()` function, and strings to numbers using `makematrix()`.

```
aStr=string(32)
aStr (a string) = 32 # result is a string
aNum=makematrix(aStr)
aNum (a scalar) = 32 # result is a scalar
```

The displayed result looks the same; only the object type has changed.

The `ascii()` function returns the ASCII representation of a single character. The `char()` function returns the character representation of a single character.

```
ascii("A")
ans (a scalar) = 65
char(65)
ans (a string) = A
```

### 5.6.2 Special Characters in Strings

Sometimes you may want to format your string output. You can insert a newline with the sequence `"n` or `char(10)`. To insert a tab, use the sequence `"t` or `char(9)`. To cause double quotes to appear in a string, use a pair of double quotes (`"`) or `char(34)`.

```
str="n2 feet, 3 inches can be shortened to " + "2'3".";
display str
2 feet, 3 inches can be shortened to 2'3".
```

You can use the `DISPLAY` command to display a string, variable, or the result of an expression; only the string is displayed (the message `ans (a string) =` is omitted.)

```
str1="A string must be enclosed in ";
str2="quotation marks. For example:"; nl=char(10); q=char(34);
test=nl + str1 + nl + str2 + nl + char(9) + char(10) + q +...
      "What's next?" + q;
display test
```

A string must be enclosed in quotation marks. For example:

```
"What's next?"
```

For more examples see the `MATRIXX` online Help `DISPLAY` topic.

5

### 5.6.3 Manipulating Substrings

You cannot use conventional indexing (see [p.194](#)) to index into a string, but you can index into a matrix of strings.

Create a matrix of strings:

```
mat=[65:69;97:101];m=char(mat(1:2,:))
```

m (a rectangular matrix of strings) =

A	B	C	D	E
a	b	c	d	e

Index into a matrix of strings:

```
bball="The N"+m(1,2)+m(1,1)+...
      " is where the action is."
```

bball (a string) = The NBA is where the action is.

You can use the `index()` function to find the starting location of a substring within a string.

```
i=index(bball,"ac")
```

i (a scalar) = 22

As mentioned earlier, `length()` returns the total number of characters in a string. The function `stringex()` extracts a substring from a string, and the function `delsubstr()` deletes all instances of a substring. Look up these functions in the `MATRIXX` online Help, and note how you can use them to alter a string, as shown in the following example:

```
bball2=stringex(bball, i, length(bball))
```

bball2 (a string) = action is.

```
bball3=delsubstr(bball, bball2)
```

```
bball3 (a string) = The NBA is where the
```

```
bball4=bball3+"money is."
```

```
bball4 (a string) = The NBA is where the money is.
```

## 5.7 Lists

Lists are created with the `list()` function. A list object can be thought of as a collection or set of other objects. Each element in the list can be of any arbitrary class, including another list. This makes nested lists possible. A list is one-dimensional, in that it can only be addressed with a single index. The following is an example list:

```
title="Gasoline Prices"; t=1:12; d=1:100;  
fg=makepoly([1,2,-.9],"t"); p="p=polyval(fg/t)/d;";  
L=list(title,t,d,fg,p)
```

```
L (a list with 5 elements) =
```

```
1:  
  Gasoline Prices  
  
2:  
  1 : 1 : 12  
  
3:  
  1 : 1 : 100  
  
4:  
  2  
  t + 2t - 0.9  
  
5:  
  p=polyval(fg/t)/d;
```

A single index can be used to access entire objects from the list.

```
p=polyval(L(4),L(2))'
```

```
p (a column vector) =
```

```
  2.1  
  7.1  
 14.1  
 23.1  
 34.1
```

```

47.1
62.1
79.1
98.1
119.1
142.1
167.1

```

The plus (+) operator can be used to concatenate two lists.

## 5.8 Index Lists

An index list contains a list of indices or pointers into a vector, matrix, or PDM. An index list looks like a matrix, but matrices cannot be used as lists. The function `find()` outputs an index list, and you can create your own with `indexlist()`.

An index list has either one, two, or three columns. If it has one column, it can be used to index into a vector. If it has two columns, it can be used to index into a matrix; the first column contains row pointers, and the second column pointers. If it has three columns, it can be used to index into a PDM; the first column is used for domain pointers, the second for row pointers, and the third for column pointers.

```

set seed 0
m=hessenberg(random(4,4))

m (a square matrix) =

    0.211325    -0.563151    0.529676    0.288135
   -1.31969     1.47381     0.313928    0.0170223
    0           -0.599434    0.164669    0.00777988
    0           0           0.173159   -0.217164

```

Find the row and column location of each element smaller than 0, and assign the value 3 to it:

```

lis=find(m<0)

lis (an index list) =

 1  2
 2  1
 3  2
 4  4

```

```
m(lis)=3
```

```
m (a square matrix) =
```

```

0.211325    3    0.529676    0.288135
3           1.47381  0.313928    0.0170223
0           3           0.164669    0.00777988
0           0           0.173159    3

```

The following example shows the use of a three-column indexlist with a PDM. (For a complete discussion of PDMs, see [5.4 Parameter-Dependent Matrix \(PDM\)](#), p.207.) Using the above matrix, create a PDM with two dependent matrices:

```
mpdm=pdm(m,[1,2])
```

```
mpdm (a pdm) =
```

domain		Col 1	Col 2	Col 3	Col 4
1	Row 1	0.211325	3	0.529676	0.288135
	Row 2	3	1.47381	0.313928	0.0170223
2	Row 1	0	3	0.164669	0.00777988
	Row 2	0	0	0.173159	3

The goal is to find all elements of **mpdm** in row 2 of a dependent matrix that are greater than 0 and less than .5 and set them to 0.1. To do this, first find the location of all elements of **mpdm** that meet the criteria:

```
mli=find((mpdm > 0) & (mpdm < .5))
```

```
mli (an index list) =
```

```

1    1    1
1    1    4
1    2    3
1    2    4
2    1    3
2    1    4
2    2    3

```

The first column shows the domain, the second the row, and the third the column. Extract the portions of the index list that index elements in row 2 of the dependent matrices.

```
row2=find(mli(:,2)==2)
```

```
row2 (an index list) =
```

```

3
4
7

```



Create an indexlist that locates only the elements in row 2 of a dependent matrix that meet the criteria used to create **mlis**.

```
rlic=indexlist(mlis(row2,:))
```

```
rlic (an index list) =
```

```
  1   2   3
  1   2   4
  2   2   3
```

Now, set all elements of **mpdm** that are greater than 0, less than .5, and in the second row to 0.1:

```
mpdm(rlic)=0.1
```

```
mpdm (a pdm) =
```

domain		Col 1	Col 2	Col 3	Col 4
1	Row 1	0.211325	3	0.529676	0.288135
	Row 2	3	1.47381	0.1	0.1
2	Row 1	0	3	0.164669	0.00777988
	Row 2	0	0	0.1	3



# 6

## *MathScript Programming*

This chapter describes how you can combine MathScript expressions, statements, commands, and functions to create MathScript programs.

Xmath handles MathScript functions (MSFs) and MathScript commands (MSCs) you write in the same manner as it does Integrated Systems commands and functions (see [3.5 Using Predefined Functions and Commands](#), p.85). MSCs and MSFs can call other MSCs and MSFs, or call themselves recursively.

### **6.1 Overview**

This section explains how to create a MathScript function (MSF) and a MathScript command (MSC), giving you a brief overview of the scripting process along the way. In subsequent sections, scripting will be explained in detail, and we will use these samples as a point of reference.

#### **6.1.1 Creating a Sample MSF**

User-defined MSFs behave exactly like predefined functions; they take input arguments, perform the statements in the body of the function using these arguments, and return one or more outputs. Input arguments are not modified.

The sample MSF **halfwave** (Example 6-1) converts all values less than zero to the value of zero. Go to your Xmath working directory and use a text editor to create a file named **halfwave.msf** as shown.

Example 6-1 **halfwave.msf**

```
#{  
  Function halfwave() has 1 required input argument  
}#  
  
Function out1 = halfwave(in1)  # function declaration line  
  out1 = in1  
  out1(find(in1 < 0)) = 0  
endFunction
```

The file begins with an optional block comment (text enclosed in `#{ }#`). If supplied, the comment serves as Help on this function if you supply a Help file (see [6.1.6 Creating Online Help for User-Defined MSFs and MSCs](#), p.249).<sup>1</sup> The function declaration is required. This declaration defines the function name, the number and type of input arguments, and the number of output arguments.

To use **halfwave**, call the function just like any intrinsic MathScript function.

```
y = [1,0,-1,0,1,0,-1,0];  
z = halfwave(y)  
  
z (a row vector) = 1 0 0 0 1 0 0 0
```

## 6.1.2 Creating a Sample MSC

While MSFs return one or more new objects as outputs and cannot modify input arguments (pass by value). MSCs do not return any values, but they can modify input arguments (pass by reference).

As an example of a typical MSC, consider the command **graphit** (shown in [Example 6-2](#)), which takes a single input and plots it on a log-log scale; a legend is supplied if the input is a matrix. Inputs other than a vector or matrix invoke an error message. Go to your Xmath working directory and use a text editor to create the file shown in [Example 6-2](#).

Example 6-2 **graphit.msc**

```
#{  
  GRAPHIT plots a numerical input.
```

1. This text will be displayed in the Local Help window when you type **help halfwave** in the Command window command area.

```

}#

Command graphit indata    # command declaration
if !is(indata,{scalar}) & !is(indata,{string})
    if is(indata,{matrix,!vector}) == 1
        plot (indata,{legend})?
    else
        plot (indata,{xlog,ylog,xmax=length(indata),
            ymin=min(indata), ymax=max(indata)})?
    endif
else
    error("Input is not worth plotting!","C")
endif

endCommand

```

The first line of the file after the optional block comment (`{# }`) section is the command declaration. The command declaration is required. It defines the command name, and the number and type of input arguments. Notice that the arguments are not in parentheses as they are in functions.

To test this command, call it as follows:

```

a=[1:.01:3];[,c]=size(a);
k = a(1,100:125);
m = k .* sin(a);
v=[a*5, a*2, a*4, a];
graphit c
graphit m
graphit v

```

In these examples, the argument to `graphit` is a single variable that requires no parsing; in cases where the argument is a simple token—a single variable or constant, you can separate the command name from the first argument with white space only, and it works. If the first argument is more complex, such as an expression, you *must* also place a comma after the command name. A comma separating the command name from the first argument *always* works. The example below illustrates this point.

Create the following MSC in your working directory:

```

Command add3nums arg1, arg2, arg3
arg1+arg2+arg3?
endCommand

```

The following usages of this command all work:

```

add3nums 1,2,3
add3nums a,b,d
add3nums a,b-c,d
add3nums a,b,d-c

```

The following produces an error message:

```
add3nums a-c,b,d
```

If you place a comma after the command name, however, the command works:

```
add3nums, a-c,b,d
```

### 6.1.3 General Rules for MathScript Programs

There are two types of names in MathScript programming: the MathScript name and the filename.

- MathScript names follow the same rules as variable names (see [Rules for Names](#) on p.69).
- MSF and MSC filenames must be lowercase, and they must match the MathScript name.
- All filenames must be unique. For example, creating both **name.msf** and **name.msc** is ambiguous (the filename for Xmath to call is undefined).

### 6.1.4 MathScript File Formats

The file formats are shown in [Figure 6-1](#) and [Figure 6-2](#).

Figure 6-1 **MSF File Format**

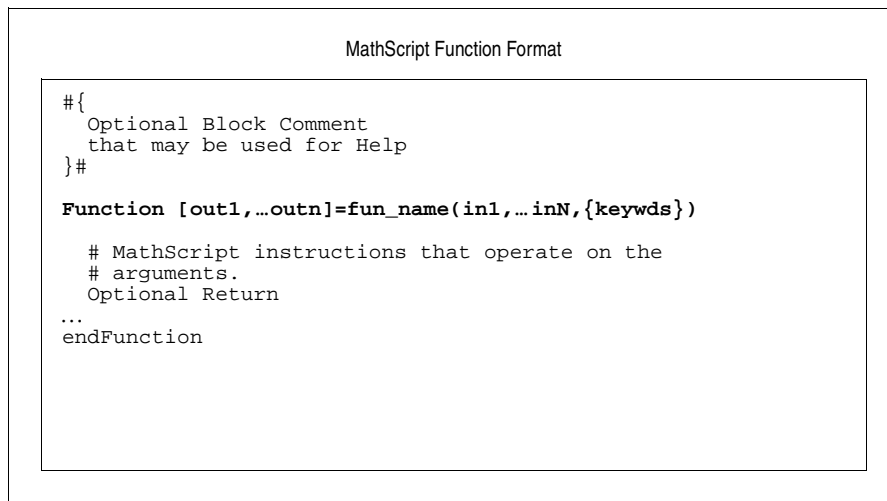
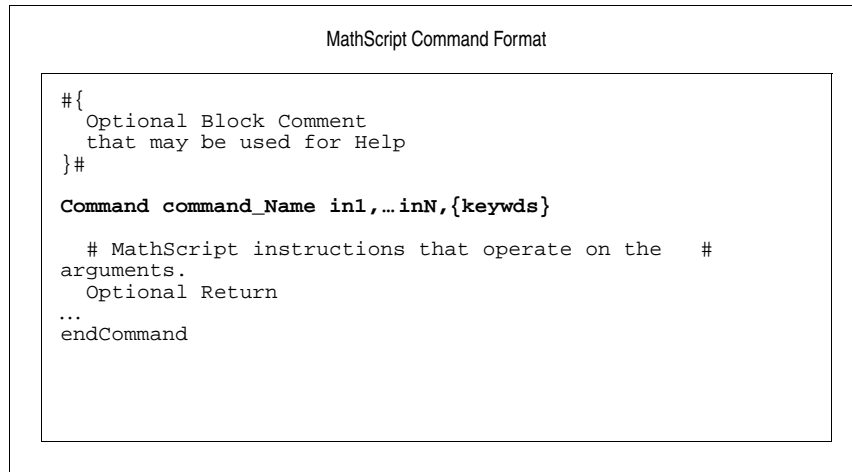


Figure 6-2 **MSC File Format**

### Comment Header

The optional comment at the top of the file may serve as the online Help entry for your MSF or MSC. To display your Help for your MSF or MSC in the Local Help window type:

```
help script_name
```



---

**NOTE:** See [6.1.6 Creating Online Help for User-Defined MSFs and MSCs](#), p.249 for additional information about creating online Help for your MSF or MSC.

---

### Declaration

The first line of code following the comment Help block is the declaration, which defines the number of input and output arguments. Required arguments are placed before the braces, while keywords are defined inside the braces.

- Files must end with the appropriate end statement (**endCommand** or **endFunction**) followed by a carriage return (blank line).
- There can only be one user-defined command or function in an MSF or MSC file (see [Example 6-3](#), p.254 and [Example 6-4](#), p.255 for extended examples of

MSCs and MSFs). MSOs (see [p.259](#)) allow for more than one function or command to be declared in a file. An optional **return** statement can be used to exit before the **endFunction** or **endCommand** statement.

### Void Function Declaration

Although the discussion and examples show both input and output arguments, you can define a void function that has no outputs. The syntax of this function declaration is as follows:

```
function [ ] = void_func_name(in1,...inN, {keywds})
```

## 6.1.5 MathScript Programming

This section gives an overview of MathScript programming. Some of the functions mentioned here are also discussed in [6.3 Programming](#), p.256.

For a detailed description of any function or command provided by Integrated Systems, see the MATRIX<sub>X</sub> online Help.

### Assigning Default Values

Optional arguments and keywords typically have default values that will be used if the argument is not specified. The **DEFAULT** command assigns a default value to the specified argument. In the following function syntax, **kw1** is given a default value of 5.0, and **kw2** is assigned "Earth" by default.

```
function [out1,out2,out3]=funName(in1,in2,{kw1,kw2})  
    DEFAULT kw1 = 5.0  
    DEFAULT kw2 = "Earth"  
    ...
```

### Output Keywords

For MathScript programs, output keywords provide a feature whereby desired output can be selected directly by name rather than positionally. For example, consider an MSF defined with this prototype:

```
[o1,o2,o3] = function myfun(i1)
```

To access only the third output of an MSF, use one of the following methods:



- Skip the first two outputs like this:

```
[,,thirdout] = myfun(a)
```

- Use output keywords like this:

```
[thirdout = o3] = myfun(a)
```

For a general discussion of keywords, see [p.86](#).

### Calling Void Functions

When you call a void function, you must use the following syntax:

```
[ ] = void_func_name(...)
```

See [Void Function Declaration](#) on [p.248](#).

### Variable Scoping

All variables created within MathScript functions and commands are local unless you use an explicit partition name (**partitionName.variableName**). Remember, you cannot change partitions within a program.

For MSFs, input arguments are passed by value. This means that functions cannot alter the values of their arguments. Output arguments requested by the caller are copied back to the scope of the caller.

For MSCs, arguments are passed by reference and can alter the values of their arguments, rename them, or delete them altogether if the argument is a variable name.



---

**NOTE:** If you get an error message from Xmath indicating that your file is incomplete, your file may be missing an ending carriage return.

---

#### 6.1.6 Creating Online Help for User-Defined MSFs and MSCs

You can provide online Help for your MSF or MSC in one of the following ways:

- Provide a Help file in the same directory as your MSF or MSC.
- Allow Xmath to use the block comment at the top of your MSF or MSC if you do not provide a Help file.

When you try to bring up Help for your MSF or MSC by typing the following command

**help** *script\_name*

Xmath follows these steps:

1. Xmath searches for the Help topic name in the standard Xmath Help project file (**help.hpf**).

If Xmath finds your Help topic, it displays it in the MATRIX<sub>X</sub> Help window.

2. If Xmath does not find an Xmath Help topic, it looks in the local Help project file (**local.hpf**).

If Xmath finds your Help topic, it displays it in the Local Help window.



---

**NOTE:** On UNIX systems, **local.hpf** is in your home directory. On Windows systems, **local.hpf** also exists in your home directory if the home directory is defined; otherwise, you can find **local.hpf** in %XMATHTMPDIR%.

---

3. If Xmath does not find the Help topic in the local Help project file, it looks in the same directory as your MathScript file for a Help file with the name *script\_name.html*, *script\_name.htm*, or *script\_name.txt*.

If Xmath finds your Help file, it displays it in the Local Help window and appends the topic name to the **local.hpf** file.

4. If Xmath does not find a Help file, it goes to the MathScript itself, extracts the text in the comment section at the top, and creates a text file (*script\_name.txt*) that contains the extracted information.

On UNIX systems, Xmath stores the *script\_name.txt* file in your home directory. On Windows systems, Xmath stores the *script\_name.txt* file in your home directory if the home directory is defined; otherwise, Xmath stores the file in %XMATHTMPDIR%.

Xmath displays the Help topic in the Local Help window and appends the topic name to the **local.hpf** file.

### 6.1.7 Using User-Defined MSFs and MSCs

Your MSF or MSC can be called in the same way as Xmath functions and commands. However, Xmath must know where to look for them.

## Search Paths

When you call a MathScript program, Xmath looks for it in the search path using the following criteria:

- The current working directory (.) is put in the search path when Xmath starts up.
- The search path is searched only upon the first call to the MSF or MSC.
- You can use **DEFINE** and **UNDEFINE** to select or deselect an MSF or MSC.

For example, if **halfwave.msf** is not found in the search path, you receive the following message:

```
File halfwave not found
```

If this occurs, add the new directory to the search path with the command **set path "directory"** where **directory** can be any valid directory path string. Assuming **halfwave.msf** is in the subdirectory **myScripts**, you add its path as follows:

```
set path "myScripts"  
show path  
  
1) .  
2) myScripts
```

## Manipulating Search Paths

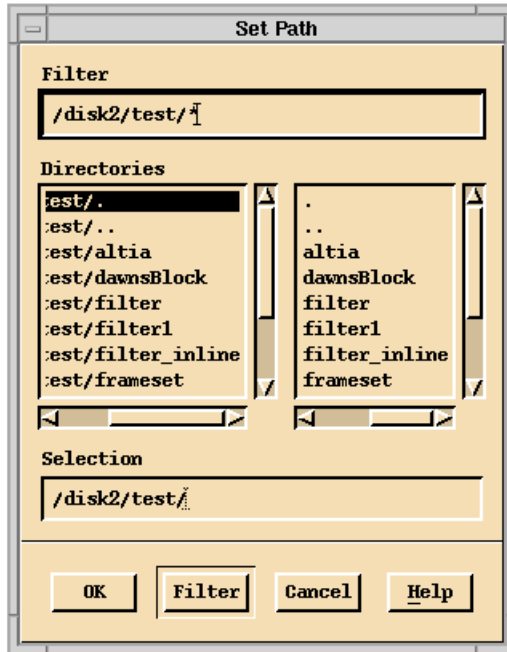
If the file **graphit.msc** is in the directory **test**, you can add this entry to the Xmath search path as follows:

```
set path "test"  
show path  
  
1) .  
2) myScripts  
3) test
```

To remove an entry from the Xmath search path, use the **REMOVE** command and the path number.

```
remove path 2  
show path  
  
1) .  
2) test
```

To handle paths through a file selection box, use Select File→Set Path. In the Directories field (of the Set Path dialog shown below), double-click on the directory you want for your search path, and then click OK.



## DEFINE

By default, Xmath looks for built-in functions and commands (see [3.5 Using Predefined Functions and Commands](#), p.85) before searching user paths. The **DEFINE** command explicitly associates an MSF or MSC with a MathScript name. It is useful for accessing functions that are not in the search path. For example, the Xmath function **hilbert()** is stored in the following location:

```
whatis hilbert
```

```
hilbert is an ISI function (path/hilbert.xf)
```

where *path* is the path to your Xmath installation. Suppose we have an MSF called **hilbert.msf** located in a subdirectory called **funs**, and we would rather use it for Hilbert computations. To make and verify the change, type:

```
define hilbert, {directory = "funs"}
whatis hilbert

hilbert is a Mathscript function (funs/hilbert.xf)
```

All calls to the **hilbert()** function will now use the function located in **funs** instead of the predefined function. To retrieve the predefined function, release your local version of **hilbert()** (and then verify with the **whatis()** function):

```
undefine hilbert
whatis hilbert

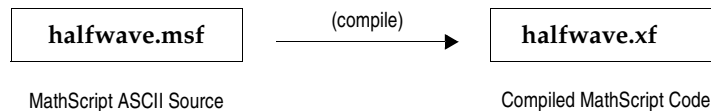
hilbert is an ISI function (path/hilbert.xf)
```

For more information on **DEFINE** and **UNDEFINE**, see the **MATRIX<sub>X</sub>** online Help.

### MathScript Program Compilation and Execution (.xf, .xc)

When a program is defined or called for the first time, Xmath compiles the program and stores the resulting binary code in an **.xf** or **.xc** file, depending on the file type. See [Figure 6-3](#).

Figure 6-3 **Compile Process for an MSF**



When **halfwave** is called again, the Xmath interpreter checks the last modified dates of **halfwave.xf** and **halfwave.msf**. If **halfwave.msf** is more recent, the ASCII **.msf** file is recompiled, overwriting the existing **.xf** file. After compilation, the new **halfwave.xf** binary file is executed.

You can use the following command to turn off file usage time stamp checking:

```
SET AUTOCOMPILE OFF
```

If you know that you will not be modifying a source file, this can improve the speed of a task such as calling an MSF in a loop.

If a new version of Xmath is installed, old local `.msf` and `.msc` files are automatically recompiled.

## 6.2 Examples

[Example 6-3](#) provides a sample user-defined MSF called `pdm2mx`. This MSF changes a PDM to a matrix of the same dimensions as the input matrix. This is to reverse any PDM formatting so that you can compare a PDM's dependent matrices with the source matrix for the PDM.

Example 6-3 `pdm2mx.msf`

```
#{-----  
Destructs a PDM to a matrix of the same dimensions as the input  
matrix. Idea is to reverse any PDM formatting so that you can compare  
a PDM's dependent matrices with, for example, the original matrix the  
PDM was created from.  
  
Syntax:      Function [mat,same]=pdm2mx(m,p)  
Inputs:      m      A matrix to compare to the elements of a PDM.  
             p      A PDM with elements you wish to organize in a  
                   matrix of the same dimensions as m.  
  
Outputs:     mat     A matrix of elements of p formatted according to m.  
             same    If mat and m are the same, same=1. If not, same=0.  
-----  
}#  
Function [mat,same]=pdm2mx(m,p)  
[mr,mc]=size(m); [pr,pc,pl]=size(p);  
  
if is(m,{matrix}) & is(p,{pdm}) & mr*mc==pr*pc*pl  
  
    if mr==pr & mc==pc  
        mat=makem(p)';  
    else  
        mat=makem(pdm(makem(p)'),{rows=mr, columns=mc});  
    endif  
    if any(mat-m) <> 0; same=0; else same=1; endif  
else  
    error("Matrix and PDM must have same number of elements.", "C")  
endif  
endFunction
```

A call to `pdm2mx` might be:

```
b=rand(6,3)? bp=pdm(b,{rows=3,columns=6})
[,same]=pdm2mx(b,bp)
```

The command `plotspectrum` in [Example 6-4](#) takes PDM input and plots the original wave and its magnitude spectrum in the Graphics window. `plotspectrum` uses `check` to see if the input is a PDM. If the input is a PDM, the length of the PDM channels is returned from `length()` to the variable `len`.

#### Example 6-4 `plotspectrum.msc`

```
#{plotspectrum first uses check() to see if the input is a PDM. If the input is
a PDM, the length of the PDM channels is returned from length() to the variable
len. The domain of the PDM is a vector stored as in am.}#

command plotspectrum input

stat = check(input, {pdm,abort});len = length(input);
dm = domain(input);

#compute the fft of the input, and the frequency range

qPDM=fft(input,{channels});
res =(len-1)/(len*(dm(len)-dm(1)));
dmF=(0:res:(len-1)*res);
output = pdm(abs(makematrix(qPDM)), dmF);

#{set up the frequency axis label. The x label on the spectral graph is
generated using + to append strings together. The final string is stored in
xLab.}#

xLab = "Frequency (resolution = " + string(res) + ")";

#{ The first call sets up the plot format. By default in the first graph, the
time series graph is placed in row 1.}#

t = plot(input,
        {rows=2,title = "original wave",
        y_lab = "amplitude", x_lab = "time (sec)"})?

#{The second plot call plots the spectrum in the second row.}#

t = plot (output, {keep, row=2, y_log, x_lab = xLab,
        y_lab = "Log Magnitude", title="Spectrum"})?
endCommand
```

A typical call to `plotspectrum` looks like:

```
time = 1:1:256; wave = pdm(cos(5*time), time);
plotspectrum wave
```

The Graphics window will display the time and spectrum plots.

## 6.3 Programming

This section describes MathScript functions, commands, and constructs used for programming.

### 6.3.1 Iterative and Conditional Looping Statements

Loops provide the ability to repeat a command or sequence of commands, either for a fixed number of iterations, or until some criterion is met. You can also exit a loop with the EXIT statement as described in the MATRIX<sub>x</sub> online Help.

#### **For**

The **For** command executes a statement or a set of statements for a specified number of iterations. If a statement contains a variable on which the **loop\_variable** operates, the order of execution is as follows:

- If the variable is a column vector, the order is top to bottom of the column vector.
- If the variable is a matrix, the order is by columns, moving from left to right.
- If the variable is a row vector, the order is from left to right.

The **For** loop syntax is as follows:

```
For loop_variable = vector
    statements
endFor
```

A line break acts as a terminator in this construct. A comma, a semicolon, or the **DO** keyword can be used. For example, the following formats are correct:

```
For x=1:n, statements; endFor
For x=1:n; statements; endFor
For x=1:n DO statements; endFor
```

#### **While**

A **While** loop iterates as long as a conditional expression is TRUE. The **While** loop can be structured as follows:



```

While conditionalExpression
  statements
endWhile

WHILE conditionalExpression, statements; ENDWHILE
WHILE conditionalExpression; statements; ENDWHILE
WHILE conditionalExpression DO statements; ENDWHILE

```

**If**

If executes a statement or set of statements when a particular condition is met; if the condition is not met, any **else** or **elseif** statements are executed.

The syntax for an if statement is:

```

If condition
  statements
elseif condition
  statements
else
  statements
endif

```

A line break acts as a terminator in the above construct, or a comma, a semicolon or the **THEN** keyword can be used. For example, the following variations are correct:

```

IF condition, statements ELSE, statements ENDIF
IF condition; statements ELSE, statements ENDIF
IF condition THEN statements ELSE, statements ENDIF

```

For example:

```

if input < cost
  display "Please deposit: "+ string(cost-input)+ " cents"
elseif input > cost
  display "Your change is: "+ string(input-cost)+ " cents"
else
  display "Thank You."
endif

```

Or, for example:

```

IF in1 | in2 < 1 THEN x=0; ELSE x=1; ENDIF

```

## Goto and Labels

A **goto** and corresponding label can be defined in a MathScript function (MSF), MathScript command (MSC), or MathScript object (MSO) file (not in a .tt file); **goto** cannot be used interactively. The **goto** command causes a jump to a specific label in the program. A label is a name enclosed in angle brackets; labels must be unique within a script.

For example, an MSF, MSC, or MSO file might have the following:

```
If input > cost & change < input-cost
  GOTO exact          # jump to <exact>
endif

#{
definition of label exact
}#
<exact>

display "Please use exact change only."
```

### 6.3.2 Object Query Functions

These functions are useful for testing the validity of input arguments of MathScript entities. To see the full set of available keywords for each function, see the MATRIX<sub>X</sub> online Help.

#### **exist()**

**exist()** checks to see if an object is defined with the given name. **exist()** returns TRUE (1) if the object is defined, and FALSE (0) otherwise.

```
a = 1; exist(a)
ans (a scalar) = 1

delete a
exist(a)
ans (a scalar) = 0
```

**check()**

**check()** performs multiple checks on a variable and prints out error messages (by default); **check()** is similar to **is** (see p.260), but has additional features including error reporting, two-input comparisons, and conversions between different object types. Both functions are useful in programming and often used interchangeably.

**check()** only operates on variable names (you can use **is** if your input is an expression); **check()** can also compare certain properties of two inputs, such as **sameClass** or **sameRate**. See the *check* topic in MATRIX<sub>X</sub> online Help for a listing associated keywords.

- By default, **check()** automatically reports an error when the keyword list does not match the input object. If you type:

```
a = [1,2,3,4];
t = check(a,{symmetric})
```

**t (a scalar) = 0** is displayed in the log area, and the following message appears in the error log window:

```
Specified argument to check must be symmetric.
```

- To turn off reporting, specify **!report** in the keyword list; the status of **check()** is still displayed in the log area, but the message is suppressed.

The **abort** keyword highlights a specific argument and returns an error message; the statement does not execute until the appropriate correction is made.

- **check()** can accept two inputs, and compare them:

```
a = [1:4]; b = [3:5];
check(a,b,{samelength, !report})
```

```
ans (a scalar) = 0
```

- **check()** can be used to make the following conversions:

- single channel PDM ↔ vector
- polynomial ↔ vector
- row ↔ column

When the **convert** keyword is used, the input is a variable; if all keyword requirements are met, the input variable is converted to the appropriate keyword format.

```
p=pdm([4:-.675:2])

p (a pdm) =

domain |
-----+-----
      1 | 4
      2 | 3.325
      3 | 2.65

[status,p]=check(p,{real,matrix,convert})

status (a scalar) = 1

p (a row vector) = 4 3.325 2.65
```

**check()** converts **p** from a PDM to a vector. See the MATRIX<sub>X</sub> online Help *check* topic for a complete description of **check()** and its keywords.

### **is()**

**is()** accepts a variable name or an expression as an input, and then determines if the input variable is of the type specified in the keyword argument. **is()** returns 1 if TRUE and 0 if FALSE.

```
tmatrix = [1,3;0,1];
is(eig(tmatrix), {identity})

ans (a scalar) = 0

is(tmatrix, {triangular})

ans (a scalar) = 1
```

**is()** can be used to report errors as follows (note that the **error()** function can only be used in a MathScript program):

```
if !(is (a,{symmetric})); error("Argument must be symmetric.")? endif
```

Many keywords can be used with **is()**; see **is()** and **check()** in the MATRIX<sub>X</sub> online Help for details about these keywords.

### **6.3.3 User Interface Functions**

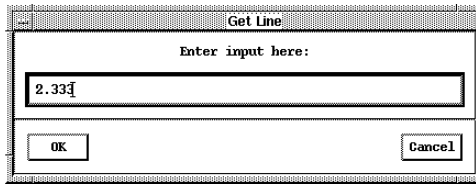
Xmath provides the simple graphical user interface functions **getline()**, **getchoice()**, **pause()**, **error()**, and **beep()**. For more sophisticated tools, see [Chapter 9](#).

***getline()***

**getline()** pops up a dialog box with a prompt asking for input.

```
response = getline("Enter input here:");
```

The dialog box appears:



You will not be able to enter text in the Command window until the dialog box is closed. If the string returned from **getline()** must be converted into a number, use the **makematrix()** function (it is overloaded to handle strings).

```
response (a string) = 2.333
```

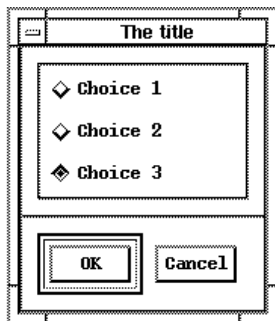
```
response = makematrix(response)
```

```
response (a scalar) = 2.333
```

***getchoice()***

**getchoice()** pops up a dialog with choices defined by an input string matrix. By default the dialog will have radio buttons, which allow only one choice. If the **multiple** keyword is used, the dialog will have check boxes, which allow more than one selection. If the keyword **defaultChoice** is specified, certain choice(s) are pre-selected when the dialog appears.

```
choice = getChoice("The title",["Choice 1";  
"Choice 2";"Choice 3"],{defaultChoice=3})
```



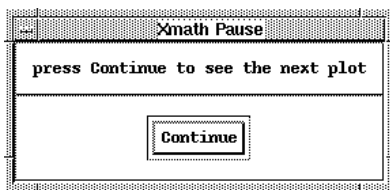
The output variable returns the user's choice(s) as a scalar or vector.

### ***pause()***

This command displays a dialog with a button that must be pressed before Xmath will continue. **pause()** is commonly used in **.ms** files to view a graph in the Graphics window.

If a string is added to the **pause()** command, that string will appear in the Xmath Pause dialog.

```
plot(1:10)
pause "press Continue to see the next plot"
plot(random(1,10))
```



You can disable **pause()** with the following command:

```
set pause off
```

**error()**

**error()** can only be used inside MathScript entities. You supply a severity code of **W**, **C**, **S**, or **F** to signify the type of error: warning, confirmation, strong warning, or fatal. The operating system and the error severity determine where the error is displayed:

- For all operating systems, **F** aborts execution; the instruction remains in the command area with the error highlighted, and the error message displayed in the message area.
- On Windows operating systems, all error messages remain in the Xmath Commands window; **W**, **C**, and **S** settings display your message in the log area.
- On UNIX, **C** and **S** settings display a dialog with the error message you specified. **W** writes your message to the message area.

See the  $\text{MATRIX}_X$  online Help for additional details.

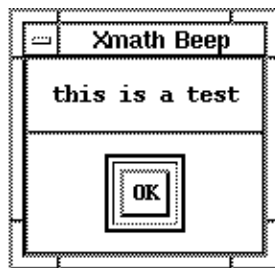
```
if is(Input2, {!matrix})==1
    error("Not a matrix!", "F", Input2)
endif
```

If the error criterion is met, the string **Not a matrix!** is written to the commands window message area.

**beep()**

**beep()** causes an audible beep; on UNIX, it also displays a popup.

```
beep "this is a test"
```



### 6.3.4 Indexing Functions

This section is a brief overview of indexing functions that are useful in programs. For detailed descriptions of these functions, see the MATRIX<sub>X</sub> online Help.

#### *index()*

**index()** finds the starting location of a substring within a string. If the substring is not found -1 is returned.

```
s="What is the meaning of this?";  
i=index(s,"this")
```

```
i (a scalar) = 24
```

#### *find()*

**find()** returns an index list of the elements in the matrix that meet the specified condition. An index list is a matrix containing the row and column locations (the indices) of all elements that meet the condition.

```
a = [20,4,-14;30,-65,0;48,582,29]
```

```
a (a square matrix) =
```

```
20     4    -14  
30    -65     0  
48    582    29
```

```
elements = find(abs(a)>25)
```

```
elements (an index list) =
```

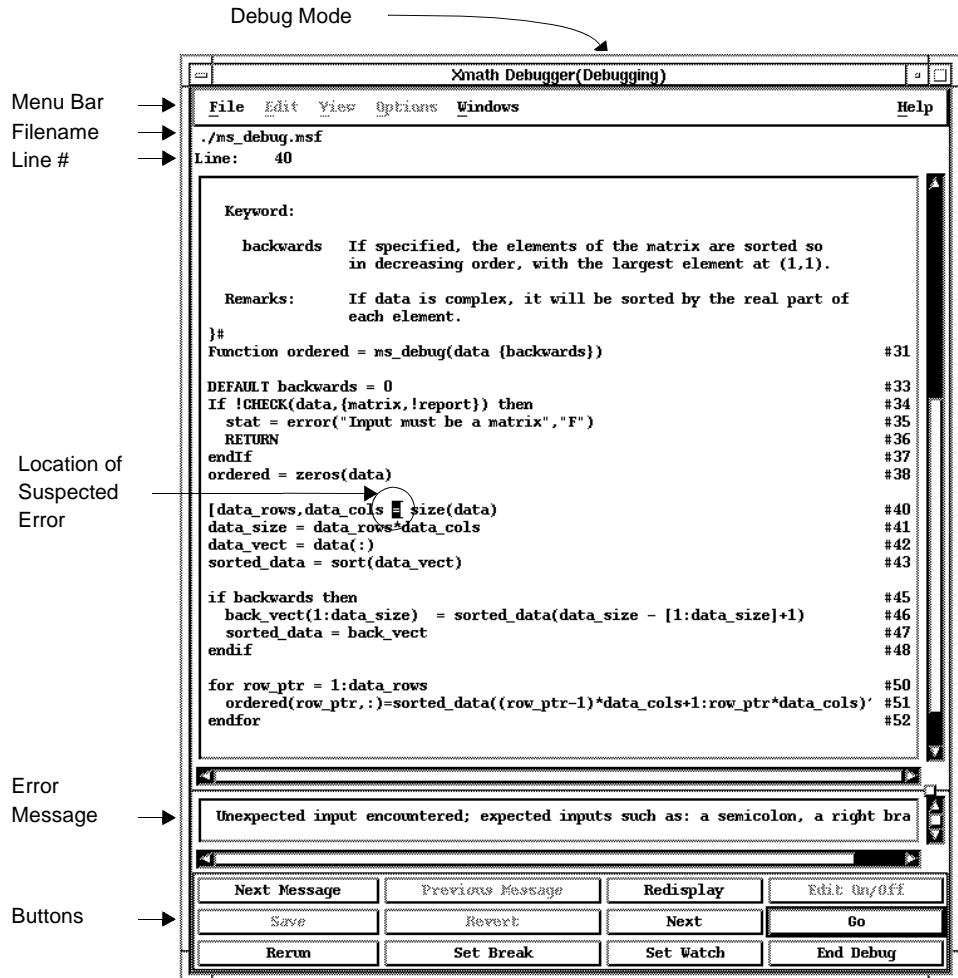
```
2     1  
2     2  
3     1  
3     2  
3     3
```



## 6.4 Using the Xmath Debugger

The Xmath Debugger can be controlled interactively from the Debugger window (Figure 6-4), or from the Commands window command line. The command line debugger is the only available method for Windows users and anyone running the tty version. This section describes both interfaces.

Figure 6-4 Xmath Debugger Window in Debug Mode (UNIX)



Note that debug mode starts under three circumstances:

- A call is made to a program that is set up for debugging.
- A program contains a syntax error. A syntax error is an error in punctuation, for example, a missing brace: `plot(a,{xlab="A missing brace")`.
- A program contains a runtime error. A runtime error occurs when an instruction is impossible to process. The following statement would cause a runtime error because the objects are incompatible: `x=5 + "hello"`.

### 6.4.1 Debug

You can use the **debug** command to define and set break for a program. In the command window command area, type:

```
debug program_name
```

If you activate debug for a program, the debugger opens automatically on the first executable line in the script whenever you call the entity. While in debug mode you can step through your file and evaluate any expression or run any command. In addition, the **NEXT** and **SET BREAK** commands can be used to debug nested functions.

### 6.4.2 Debug Mode

In addition to the above cases (where you are intentionally debugging a specific MSF or MSC), a programming error also invokes the Debugger window in debug mode (see [SET DEBUGONERROR](#)).

#### Entering Debug Mode

- All windows say “(Debugging)” in the title area (at the top) when you are in debug mode. In the Debugger window, the full filename of the entity being debugged is displayed just below the menu bar.

In the command line debugger, the command prompt will change to:  
**(program)Debug>**.

- If the Debugger window was opened because the file contains a syntax error, the Next button is enabled (see [Figure 6-4](#)). If there are multiple errors, the Next

Message and Previous Message button is also enabled. You can repair a syntax error, then continue to step through your file or look at the previous message.

### Stepping Through a Script

- In the command line debugger you can step forward, using the **next** command, or continue execution with the **go** command.
- You can set and remove break points from the Debugger window or the command line (see [6.4.3 Setting, Showing, and Removing Breakpoints](#), p.268).
- You can set and remove watch points from the Debugger window or the command line (see [6.4.4 Setting and Removing Watchpoints](#), p.269).

### Exiting Debug Mode

- To stop debugging from the Debugger window, click End Debug.
- To stop debugging from the command line, type **abort**.

In the Debugger window, the word Debugging disappears from the title area of all windows. This mode is referred to as Edit mode.

To close the window select File→Close Window. To stop debugging and close the debugger in one step, place the cursor over the Debugger window and type Ctrl-W (for workstations only).

### Editing a File in the Debugger Window

When the Debugger window is not in debug mode, it acts as an editor. To fix your script, click into the Debugger window and make your change. If you modify the script via the Debugger window, the Save and Revert buttons become active, and you can no longer step through. Before saving, make sure that the script file is not open in any other editor.

The Debugger window provides the same simple editing capabilities available from the Xmath Commands window command area ([1.6.3 Command Area](#), p.18). You can manually open the debugger by selecting Windows→Debugger. To edit a file, select File→Open. The file appears in the window. Once you make a change, the Save and Revert buttons are activated.

## SET DEBUGONERROR

The environmental setting **debugonerror** determines the mode in which the debugger will appear.

- The default setting is **On**. If an error is detected in a program, Xmath opens the debugger and redirects focus to the Xmath Debugger window (see p.266).
- If **debugonerror** is set to **On**, and you have activated debugging for a program with **debug program\_name**, the debugger opens in debug mode whenever the entity is called.
- If **debugonerror** is set to **Off**, and you have activated debugging for a program with **debug program\_name**, the debugger opens whenever the entity is called, but focus stays in the Commands window.

### 6.4.3 Setting, Showing, and Removing Breakpoints

A breakpoint causes the debugger to stop execution at a specific line number in the source, provided that **set debugonerror on** is in effect (the default).

- If you issue the command **DEBUG NAME** a break is automatically set on the first executable line of the script, causing the debugger to open whenever that script is called.
- You can set a breakpoint interactively in the Debugger window, or from the Commands window command area.



---

**NOTE:** In order to set a breakpoint interactively, the file in which you wish to set or remove breakpoints must currently be open in the Xmath debugger in debug mode.

---

- To set a breakpoint in the Debugger window, position the cursor in the line where you want to break execution, then press the Set Break button. Note that when you position the cursor in the Debugger window, the line number is shown below the filename on the upper left.  
  
or
- Go to the command area and type:  
  
`SET break lineNumber`
- To see a list of the breakpoints you have set, go to the command area in the Commands window and type:

```
SHOW break
```

A list of breakpoints will appear in the format **fileName:Line\_Number**. You will see breakpoint line numbers for all entities that have debugging enabled.

- Breakpoints can be removed via the Commands window with the **REMOVE** command. (Again, you must be viewing this script in debug mode.) Go to the command area and type:

```
REMOVE break lineNumber
```

As mentioned earlier, all scripts that have been called or explicitly defined automatically have a breakpoint set on the first executable line. Type **SHOW debug** to see the files you are debugging.

- To run a file without stopping at its breakpoints, go to the command area and type:

```
DEBUG program_name off
```

Note, however, if the script contains an error, the debugger will open regardless.

#### 6.4.4 Setting and Removing Watchpoints

A watchpoint causes the debugger to stop execution whenever a watched variable is modified.

You can set a watchpoint interactively in the Debugger window, or from the Commands window command area. The script containing the variable you want to watch must currently be shown in the debug window in debug mode:

- To set a watchpoint interactively, go to the Xmath Debugger and highlight the variable you want to watch, then press the Set Watch button.
- To set a watchpoint with the **set** command, go to the commands window command area and type:

```
set watch varName
```

Now you can use the Commands window to display the values of variables that are local to the current MSF or MSC.

To see a list of the variables you are watching, go to the Commands window command area (while in debug mode) and type:

```
show watch
```

A listing appears in the format **functionName:varName**.

Watchpoints can be removed via the Commands window with the **remove** command. The entity containing the watchpoints you want to remove must currently be shown in the debug window in debug mode. Go to the Command window command area and type:

```
remove watch varName
```

If you want a function to run without stopping at the watchpoints but you do not want to remove them, type

```
debug program_name off
```

in the command area.

### 6.4.5 Debugger Window Interface

This section describes the Xmath Debugger's user interface.

#### Fields

The filename of the function being debugged is displayed just below the menu bar.

The top field in the window contains the source of the MSF or MSC that you are debugging. The line that is about to be executed is highlighted (unless there are errors in the function, in which case the highlighted line points to the error). The source field is read-only unless you have write privileges to the source file. The middle field is the message area. Status and error messages that occur while debugging are displayed here.

#### Menus

The enabled menus are the File menu and the Windows menu. The File menu allows you to edit another MSF or iconify the debugger. The Windows menu allows you to quickly find other Xmath windows and bring them to the foreground.

## Buttons

**Next Message** — Enabled when there are multiple errors. This button highlights the next line that contains an error (assuming you are not at the bottom of the list).

**Previous Message** — Enabled when there are multiple errors. This button highlights the previous line that has an error (assuming you are not at the top of the list).

**Redisplay** — Refreshes the window.

**Edit On/Off** — Toggles the source to be editable or read only. You may want to toggle edit off to prevent accidental edits.

**Save** — Enabled whenever you make changes to the source. Pressing this button saves your changes to the file.

**Revert** — Discard edit changes and load the last saved version of the file.

**Next** — Executes the next line of code.

**Go** — In debugging, causes the function to run until a break point is encountered, a watched variable is modified, or the end of the file is reached.

**Rerun** — Enabled after source changes have been saved. Press to rerun the function with previous inputs.

**Set Break** — Sets a breakpoint on the current line (where the cursor is in the source field). Xmath will pause function execution at any breakpoint(s) you set. To do this from the command area, see the `MATRIXX` online Help under **set break**.

**Set Watch** — Sets a watch on a variable. To watch a variable, use the pointer to highlight the variable name, then press this button.

**End Debug** — Exit debug mode; no arguments will be returned from the function.

## 6.5 Advanced Topics

This section includes the following topics:

- Variable arguments
- Executing a function at a specific directory
- Partition and variable directory functions
- MathScript command output and error capture
- Programming for platform independence

### 6.5.1 Variable Arguments

When you use the colon (:) index operator in a MathScript entity declaration, the program handles a variable number of inputs, outputs, or keywords. The function **argn()** returns the number of a program's arguments, while **argv()** extracts the value and name of the argument.

#### **argn()**

**argn()** returns the number of inputs (the default), keywords, or outputs for a MathScript entity (see [Example 6-5](#)). To get the number of keywords, specify the keyword, **keywords**; to get the number of outputs, specify the keyword, **outputs**.

Example 6-5 **argn()**

```
function [args]=howmany(:)
    args=argn()
endfunction
```

[Example 6-5](#) counts the number of inputs. For example, **howmany(1,1,1,1)** returns 4.

#### **argv()**

**argv()** allows you to index into the inputs, keywords, or outputs for a program. **argv()** can return the value and/or name of the argument; for **argv()** to return the name of the argument, however, it must be a keyword. To return the name of an



output, the calling statement must use output keywords (see [Output Keywords](#) on p.248).

### Using `argn` and `argv`

[Example 6-6](#) uses the `argn()` to determine the number of inputs and loop over them accordingly. `argv()` gets the value of each argument, and then the length is determined for the output.

Example 6-6 **`argv()` combined with `argn()`**

```
function out=howlong(:)
n=argn();
for i=1:n
    in=argv(i)
    out(i)=length(in);
endfor
endfunction

x=howlong(rand(2,3),1:7,pdm(ones(4,5),{rows=2})))?

x (a column vector) =

     6
     7
    10
```

[Example 6-7](#) accepts any number of scalars; it displays a message when the keyword `reply` is specified but not otherwise.

Example 6-7 **`msg.msf`**

```
function [out]=msg(,{reply})

ni=argn()
nk=argn({keywords});
[v,n]=argv(ni);
ni=ni-nk;
if n=="reply"
    key=1;
else
    key=0;
endif

for i=1:ni
    if is(argv(i),{!scalar})
        error("Scalars Only!", "C");
    else
        out(i)=argv(i);
    endif
endfor
```

```
    if key==0 & i==ni
        out;
    endif
    if key==1 & ni==1
        display "Thanks for the scalar!" ;
    elseif key==1 & i==ni
        display "Thanks for the " + string(ni) + " scalars!" ;
    endif
endfor

endfunction
```

```
msg(1,1000,pi,{reply})
```

Thanks for the 3 scalars!

ans (a column vector) =

```
    1
 1000
 3.14159
```

```
msg(5,5,9)
```

ans (a column vector) =

```
    5
    5
    9
```

[Example 6-8](#) provides the function `varargs()`, which has a variable number of outputs, inputs, and keywords. In the following call:

```
[out1=fop1,out2=fop2]=varargs(1,2,3,{k=9})
```

Note that we define two outputs (**fop1**, **fop2**), three inputs, and one keyword (**k**).

Within the function, `argn()` is used to determine the number of arguments, and `argv()` is used to determine the name of the arguments. Note the use of the `[value,name]=argv(i,{keywords})` syntax for inputs and keywords and the `name=argv(i,{outputs})` syntax for outputs. Note also that the function itself does not assign a value to the outputs.

The output of the above call appears in [Example 6-9](#). The names of the keyword and the outputs appear in the output stream; the names of other input arguments are **NULL**.

Example 6-8 **varargs.msf Using argn and argv**

```
function [:] = varargs(:,{:})
```

```

for i=1:argn({keywords})
[v,n] = argv(i,{keywords})?           # display value and name of
end                                   # keyword inputs
display "-----"n"

for i=1:argn(!keywords)
[v,n] = argv(i,!keywords)?           # display value and name
end                                   # of non-keyword inputs
display "-----"n"

for i=1:argn()
[v,n] = argv(i)?                     # display value and name
end                                   # of all inputs
display "-----"n"

for i=1:argn({outputs})
n = argv(i,{outputs})?               # display name of all outputs
end

endfunction

```

Example 6-9 **Output of varargs.msf**

```

v (a scalar) = 9
n (a string) = k
-----
v (a scalar) = 1
n is null
v (a scalar) = 2
n is null
v (a scalar) = 3
n is null
-----
v (a scalar) = 1
n is null

v (a scalar) = 2
n is null
v (a scalar) = 3
n is null
v (a scalar) = 9
n (a string) = k
-----
n (a string) = fop1
n (a string) = fop2

```

To assign values to the outputs fop1 and fop2, the function needs an assignment statement(s), which must be a text string. For example, the following loop assigns the outputs with the values 1 and 2, respectively:

```

for i=1:argn({outputs})
    n = argv(i,{outputs})

```

```
execute n + "=" + string(i) + ","; # assign i to the i'th output
endfor
```

### 6.5.2 Executing a Function at a Specific Directory

The function assignment syntax used in calling an LNX in background mode allows a directory to locate the function to be specified with a keyword. For example:

```
[out] = (define myfunc, {directory="mydir"})(1,2,3)
```

where Xmath calls the MSF or LNX function **myfunc()** in the directory **mydir**, leaving an existing definition of **myfunc()** unchanged.

### 6.5.3 Partition and Variable Directory Functions

The function **directory()** allows directory listings of Xmath partitions and variables to be captured as vectors of string names. The **directory()** function requires one input, a string containing a wildcard as used in the command **WHO**, and produces one output, a vector of names of partitions and variables as produced by the command **WHO** using the specified wildcard. The names are always full names, and the partition name is always prefixed. The syntax is shown in the following example:

```
out = directory("main.*")
```

where the variable **out** will contain a vector of strings of the variable names found in **main** (for example, **main.a**, **main.b**, etc.).

### 6.5.4 MathScript Command Output and Error Capture

The following syntax allows the textual output and error messages of a MathScript command to be captured in MathScript variables as string values:

```
[outputs = format, errors] === statement
```

or

```
[outputs, errors] === statement
```

where **outputs** and **errors** are MathScript variable names and **statement** can be any valid MathScript statement. The **format** keyword formats the output in a command-dependent way; see the examples below for details.

If the **outputs** variable is specified, the textual (nongraphical) outputs of **statement**, if any, are inserted into the **outputs** variable instead of displaying in the Xmath log area of the Commands window. If the **outputs** variable is omitted, the output of **statement** is displayed normally.

If the **errors** variable is specified, Xmath will suppress normal processing (error location highlighting, bringing up the Debugger window, and stopping command execution) of any errors generated by **statement**. Instead, the error messages are converted to text and inserted into the **errors** variables. If **errors** is omitted, Xmath performs normal error processing of errors generated by **statement**.

This error capture feature allows a program to perform error handling of commands that may fail as shown in the following [Examples](#) section.

### Examples

In the following example of error handling, if the variable name contained in the string **varname** is a legal Xmath variable name, **err** would be a null; otherwise, **err** would contain an error string. For example:

```
varname = getline("Please enter an Xmath variable name:");
[,err]===execute varname + "=1;"
```

In the following example of error handling, any error calling **myfunc** is converted into an error message and inserted into **err** as a text string:

```
[,err] === myfunc(123)
```

In a similar example, the variable **out** captures the output of the Windows **dir** command in a string:

```
[out] === oscmd("dir")
```

In the following example, **out** contains a formatted version of the captured output:

```
[out=format] === statement
```

Currently, the **WHO** and **SHOW PARTITIONS** commands support this formatting. The **directory()** function described in [6.5.3 Partition and Variable Directory Functions](#), p.276 uses both these commands. For example,

```
DIRECTORY("main.*")
```

actually executes this statement:

```
[out=format] === who main.*
```

The captured output is a vector of strings containing the names of the variables in the partition **main**.

When **[out=format]** is used with other statements that don't support formatting, the captured output will be a vector of strings, each of which contains a line of output. By default, the length of the row vector **out** is the number of strings (and therefore the number of lines in the captured output). You can transpose **out** to see the output strings as they are normally displayed in the Xmath log area.

```
[out=format] === rand(2,2)
size(out)
out'?
```



---

**NOTE:** This syntax cannot be nested.

---

## 6.5.5 Programming for Platform Independence

While MathScript is portable across UNIX and Windows platforms, calls to the operating system are platform-dependent. For example:

```
oscmd("ls *.xmd") # UNIX
oscmd("dir *.xmd") # Windows
```

With the MathScript function **platform()**, you can program a command so that it can be run on either platform. For example:

```
if platform() == "UNIX"
    oscmd("ls *.xmd") # UNIX
else
    oscmd("dir *.xmd") # Windows
endif
```

Another problem area with cross-platform programming is the directory path name syntax difference. The **get({path})** function is useful in reconciling these differences. The **COPYFILE** command, for example, makes use of the **get({path})** function to provide a platform-independent way of copying files. For more information, see the MATRIX<sub>X</sub> online Help.

# 7

## *MathScript Objects*

This chapter outlines the procedure for writing and using your own MathScript object (MSO). Before writing an MSO you should have a good understanding of object-oriented concepts and Xmath objects in particular. [Chapter 5, Data Objects and Operators](#), introduces each intrinsic Xmath object and the operators that are overloaded for that object. You should also be proficient in the MathScript language ([Chapter 3](#) and [Chapter 6](#)).

As described in [Chapter 5](#), you can easily augment these intrinsic objects by designing your own custom objects using MathScript.

### **7.1 MSO Overview**

The MathScript object feature enables you to create custom high-level objects for use in the Xmath environment. Object development in Xmath fundamentally involves determining what data defines the instance of an object, writing the initializer function and creating the various commands, functions, and operators which can manipulate object instances. The complete definition of an object and its behavior is encapsulated within an MSO file. The structure and contents of an MSO file are described in greater depth in subsequent sections.

Careful thought should be used when developing objects, especially those which will be shared among a number of people. The object author should design, test, and document objects *before* allowing others to use them. Once an MSO is in use,

any changes to the definition of the class variables will create inconsistencies between current and future instances that may be difficult to identify.

### 7.1.1 Object Instantiation

Once an object is defined by creating an MSO file, object instances can be created from the Xmath command line or within any script using the following syntax:

```
instance = myobject(parameters);
```

This statement executes the object's initializer function with the supplied input parameter(s). The output of this expression is an object instance. An object instance is recognized as an Xmath variable; this implies that it can be operated on by Xmath commands such as **SAVE**, **LOAD**, and **DELETE**, copied with the assignment operation, passed as a parameter to a function or command, and returned as a function output.

The object instance is a container that stores the persistent *class variables* that characterize a particular instance. The syntax for accessing a class variable is the same as the syntax for addressing a variable in another partition. For example, if an object named **myobject** contains a class variable named **sigma**, then that variable can be accessed with the following statement:

```
instance.sigma
```

### 7.1.2 MSO File Format

MSO file format structure adheres to the rules in Sections 6.1.3 and 6.1.4 on p.226, with one exception. The MSO file format accommodates multiple constructs in a single file. This enables you to use a single file to define the object, overload or create pertinent functions and commands, and overload operators to support the new object. [Example 7-1](#) illustrates the structure of an MSO file.

#### Example 7-1 Sample MSO File Format

```
#{  
Block comment used as Help for this object.  
}#  
  
Object[x1,...] = mymso(in1,..., {kwds})  
    ... MathScript statements  
endObject
```



```

Operator z1 = +(<type>left,<type>right)
    ... MathScript statements
endOperator

Function[y1,...] = memFun(<type>a,..., {kwds})
    ... MathScript statements
endFunction

Command memCmd <type>input {kwds}
    ... MathScript statements
endCommand

```

- If MATRIX<sub>X</sub> online Help is desired, supply a Help file or begin the file with commented text that will serve as the Help text.




---

**NOTE:** You provide online Help for MSOs the same way as for MSFs and MSCs; see [6.1.6 Creating Online Help for User-Defined MSFs and MSCs](#), p.229 for more details.

---

- The body of the file consists of programming constructs. The first construct in the file must be the *initializer function* for the object. The initializer function contains the MathScript statements which are executed by Xmath whenever a new instance of this object is created. The initializer function is explained in greater detail in [7.2 Initializer Function](#), p.282.
- Optional constructs to define or overload MathScript functions and commands that act on your object can follow the initializer function in any order, as discussed in detail in [7.4 Member Functions](#), p.289.
- Optional constructs to overload operators can also appear anywhere after the initializer function, as discussed in detail in [7.3 Operator Overloading](#), p.285.

### 7.1.3 Using MSOs in Xmath

The process for defining an MSO is identical to that for other MathScript entities (see [6.1.7 Using User-Defined MSFs and MSCs](#), p.230). Just include the MSO files you need in your Xmath path. Alternatively, you can define them explicitly with the **DEFINE** command:

```
define mymso,{directory="/myHome/myobjects/my_mso"}
```

Xmath dynamically loads an MSO definition into memory only when it is necessary.

## 7.2 Initializer Function

The initializer function is a special function that is executed to create a new instance of an object. It is the only required component in an MSO, and it must be the first construct in the MSO file following the optional Help text. The syntax for an initializer is the same as MathScript functions, except that the initializer is declared between the statements **Object** and **endObject**. All other rules in Sections 6.1.3 and 6.1.4 on p.226 apply.

A simple initializer function is shown below.

```
Object[y1]=mymso(a1,{b1})  
    ... MathScript code  
endObject
```

### 7.2.1 Class Variables

An object instance is characterized by persistent variables that are stored within the object instance, similar to the way variables are stored within a partition. The initializer is responsible for creating an instance and storing the class variables within the instance. After object instances have been created, any other constructs defined in the **MSO** file can access the class variables.

There are three types of class variables: required, optional, and computed. Examine the following code fragment:

```
Object[y1]=mymso(a1,{b1})  
    ... MathScript code  
endObject
```

- Required variables, such as **a1** in the example above, must be specified by the user when the object instance is created.
- Optional variables, like **b1**, are optional input arguments to the initializer.
- Computed variables, such as **y1**, are calculated by the initializer, typically as a function of the input arguments.

Any number of required, optional, or computed class variables may be defined for an object. The **DEFAULT** command is sometimes useful to give optional and computed variables a default value.

When the initializer completes execution, a class variable that exists within the function scope will be stored within the object instance. The MathScript statements within the initializer can modify or delete any class variable. As a

result, required, optional, or computed arguments may or may not exist within an object instance, depending on statements in the initializer.

Variables created in the body of the initializer that are not class variables are considered temporary and are automatically deleted when the initializer completes execution. If you want a variable to be persistent, specify it as a computed variable.

When an object initializer is called, the result of that statement is always a single instance of the new object. The defined outputs, such as **y1** in the above initializer function, are used to create a computed class variable (as opposed to the output of an ordinary function).

The following is a sample initializer function for the new object **mysys**. Note that this object does not have any computed variables. They are not required.

```
Object mysys(a,b,c,d, {dt})  
  ... MathScript code  
endObject
```

You would create an instance of **mysys** as follows:

```
inst= mysys(1,2,3,4);
```

After the input variables are created within the object and given their appropriate values, the initializer is called in the scope of the **inst** object. The initializer checks the arguments for correctness, sets any optional arguments that require a default value, and then calculates the output arguments based on the inputs. When the initializer is complete, all local variables are deleted from the object.

## 7.2.2 Nested Objects

Any class variable can be an instance of another object. As a result, you can create quite complex nested object hierarchies. If a required or optional class variable is an object, the user must create an instance of the nested object and supply it as an input to the initializer. If a computed class variable is an object, the initializer itself will create the instance of the nested object.

Let's say you had the following nested object embedded within two other objects.

```
Object nested(z)  
  ... MathScript code  
endObject  
  
Object supplied(<nested> x)  
  ... MathScript code  
endObject
```

```
Object [x] = computed(y)
  x=nested(y)
  delete y
endObject
```

To create an instance of the object **supplied**, the user would type the following:

```
a = nested(1);
b = supplied(a);
```

However, to create an instance of the object **computed**, the user only types the following:

```
c = computed(1);
```

### 7.2.3 Type Declaration

Type declarations are qualifiers that can optionally precede each input argument for functions, commands, and operators defined in an MSO. They create a restriction that an argument must be an instance of a particular type of object.

The syntax of a type declaration is to specify the name of an MSO within a set of angle brackets immediately before any input argument.

```
Object[x]= mymso(in1,<alien>in2)
  ... MathScript
endObject
```

In the initializer function shown above, the type declaration **<alien>** specifies that any instance of an object of type **alien** will be accepted as the second argument.

Arguments that do not have a type declaration indicate that any object will be accepted when this function, command, or operator is called.

The Xmath interpreter uses type declarations for two purposes:

- Ensure that parameters passed to user-defined functions and commands are the correct type. If a mismatch is encountered, Xmath will automatically generate an error message.
- Facilitate function, command, and operator overloading by limiting the use of certain constructs to a specific combination of input arguments. The use of type declarations to achieve overloading is described in detail in a later section.

## 7.3 Operator Overloading

The ability to customize the behavior of operators in Xmath to manipulate MSOs is called operator overloading. Operator definitions containing MathScript statements that should be executed to achieve the desired behavior are placed within an MSO file. The syntax of an operator definition is similar to that of a function definition, with the exception that the operator behavior is declared between the **Operator** and **endOperator** statements. For example, to define the plus (+) operator to add two **apple** objects together, you would insert the following construct in **apple.mso**.

```
Operator y = + ( <apple>left, <apple>right )
    ... MathScript code
endOperator
```

Multiple operator definitions may be required for the same operator to completely define all possible object combinations. For example, if you have an **apple.mso** and an **orange.mso**, you would need the following three operator definitions in addition to the one above to describe all possible combinations of adding apples and oranges.

```
Operator y = + ( <apple>left, <orange>right )
    ... MathScript code
endOperator

Operator y = + ( <orange>left, <apple>right )
    ... MathScript code
endOperator

Operator y = + ( <orange>left, <orange>right )
    ... MathScript code
endOperator
```

Operator definitions can be inserted in any of the MSO files that are declared as arguments. So the two operators that combine apples and oranges can appear in either the **apple.mso** or the **orange.mso**. However, because Xmath searches MSO files from the left argument to the right argument, it is more efficient to put the operator definition in the MSO file corresponding to the first argument.

Type declarations, like **<apple>**, tell the Xmath interpreter which operator definition to choose from when performing operations that deal with objects. For unary and binary operator definitions, at least one of the arguments must have a type declaration for the MSO in which the operator definition resides.

Type declarations are not required on all arguments. If a type declaration is not specified, Xmath will accept any variable for that argument. For example, the

following operator will add an **apple** object to any type of object including intrinsic Xmath objects such as **matrices**, **strings**, etc.

```
Operator y = + ( <apple>left, right )  
... MathScript code  
endOperator
```

The MathScript code within such an operator should check unqualified arguments and restrict inputs to the object types that the MathScript code can properly handle; an error should be returned if the conditions are not met.

Operators that can be overloaded are listed in [Table 5-1](#), p.172.

Unary operators act on a single variable and their operator definitions will have only one input argument. Binary operators act on two variables and their definitions will have two input arguments. The - operator is both a unary and binary operator and Xmath will automatically select the correct definition from an MSO file based on the number of declared arguments.

```
Operator y = - ( <apple> arg)  
... MathScript code  
endOperator
```

```
Operator y = - ( <apple>left, <apple>right )  
... MathScript code  
endOperator
```

The comma and semicolon operators are special operators that can accept two or more operands. For example the following operator definitions describe two combinations of different types of objects manipulated by the comma operator.

```
Operator y = , ( <obj1>one, <obj2>two )  
... MathScript code  
endOperator
```

```
Operator y = , ( <obj1>one, <obj2>two, <obj3>three )  
... MathScript code  
endOperator
```

The comma operator definitions above would correspond to the following two types of expressions, assuming **a**, **b**, and **c** are of the appropriate type:

```
case1 = [a,b];  
case1 = [a,b,c];
```

The comma and semicolon operators can also be used in compound expressions. In the following example, **a** and **b** would first be resolved using the appropriate comma operator to produce an intermediate result, then, **c** and **d** would be resolved with the appropriate comma operator to produce a second intermediate

result. Finally, the two intermediate results would be resolved with the appropriate semicolon operator.

```
result = [a, b; c, d];
```

When the comma or semicolon operators act on an operand of heterogeneous types, a separate operator definition is required for each specific combination of operands, as was illustrated in the above examples. However, the variable argument **construct** (:) can be used when all operands are of the same type (see [6.5.1 Variable Arguments](#), p.252). The variable argument **construct** also has the advantage that a single operator definition can generically handle any number of operands. The following definition of the comma operator illustrates the variable argument syntax:

```
Operator y = , ( <special>: )
    n = argn();
    for i = 1:n
        x = argv(i);
        y = ...
    endfor
endOperator
```

The colon argument (:) instructs Xmath that any number of operands will be accepted by this definition, all of which must be of type **special**. The **argn()** function, which requires no inputs, will return the number of operands. The **argv(i)** function accepts an integer between 1 and the number of operands and will return a copy of the requested operand. Consequently the variable argument operator definitions can be generically programmed with loops to handle any number of homogeneous operands.

The insertion and extraction index operators are also special operators. The insertion index operator enables indexing into an object instance on the left side of the equal sign in an expression. In the following example, **inst** is an instance of an MSO called **myObj**, and the following expression attempts to insert 10 into the second element of the **inst** object.

```
inst = myObj(a);
inst(2) = 10;
```

The extraction index operator enables indexing into an object instance on the right side of the equal sign in an expression. For example, the following expression attempts to extract the value from the fifth element of the **inst** object.

```
ans = inst(5);
```

The definition of the insertion and extraction index operators would have the following structure and would reside in the **myobj.mso** file.

```
Operator Object(i) = y
... Mathscript code
endOperator
Operator y = Object(i)
... Mathscript code
endOperator
```

The argument *i* would contain the element indices 2 and 5 from the above examples at runtime. The argument *y* would contain the value to be inserted or the result to be extracted to or from the object. The MathScript code within the index operator should check and restrict the input arguments (*i* and *y*) to only object types with values that the MathScript code can properly handle; an error should be returned if the conditions are not met.

The word **Object** in the above declarations is a reserved token which instructs Xmath that this is a special operator that will execute directly within the scope of the object instance. In other words, the MathScript code within these operators can directly access the class variables within the instance. For example, let's say the variable *x* is a class variable of **myObj**. The MathScript code within a binary plus (+) operator would have to reference *x* with the statement **left.x** or **right.x**, but the index operator can reference *x* directly with the statement *x*. Take care that the declared arguments of the operator (*y* and *i*) do not overwrite the class variables of the object.

The index operators can accept any number of operands, as long as an operator definition with the appropriate number of arguments resides in the object's MSO file. To also handle two-dimensional indexing for the **myObj** example object, the following two operators, each with two index arguments, *i* and *j*, would be required.

```
Operator Object(i,j) = y
... Mathscript code
endOperator

Operator y = Object(i,j)
... Mathscript code
endOperator
```

The index operators also support the variable argument construct to handle any number of operands generically. The following extraction index operator illustrates the variable argument syntax for the index operator.

```
Operator y = Object(:)
  n = argn();
  for i = 1:n
    x = argv(i);
    y = ...
  end
endOperator
```



## 7.4 Member Functions

Your MSO should include any functions or commands that use your object.

- Member functions and commands behave like MSFs and MSCs with the exceptions that they cannot be debugged individually unless they are uniquely named.

Once your MSO is defined, MSO member entities can be called from the Xmath command area, or other MathScript files.

- You can overload existing commands and functions to operate on your object. For example, the following function overloads the function `max()` to accommodate the MSO type `group`.

```
function [out]=max(<group>a)
    out=max(a.data)
endfunction
```

When a function or command is overloaded, its behavior is limited to the cases specified in the function header. For example, the overloaded version of `max` will only be enabled if the input is a group object.

- The file need not contain all the code for each new function or command. Using LNXs for complex numerical operations will speed up execution considerably.
- You can identify member functions with the `whatis` command. For example:

```
whatis other

other is a member function (./other.mso)
```

- Member function and command definitions do not include Help text; their Help text should be included with the Help text for the MSO.

### 7.4.1 Sample MSO

The MSO shown in [Example 7-2](#) defines an object named `group`. This MSO will accept any single row matrix. This MSO overloads the `min()` and `max()` functions to support this object. It also overloads binary and unary minus (`-`), `*`, `+`, and binary and unary equality. You can find this example in `$XMATH/examples/mso/group.mso`.

Example 7-2 **group.mso**

```
#{-----
The group object is an unordered collection of unique whole numbers which can
be manipulated by operators that adhere to conventional set theory. We are
using the name "group" for this object so it does not conflict with the "set"
command in Xmath.
```

A new group is defined using the group initializer. For example:

```
s1 = group( [1,2,3,4] );
s2 = group( [3,4,5,6] );
```

Binary group operators are defined as follows:

```
A + B = union of A and B
A - B = difference, the elements of A
       which are not in B
A * B = intersection of groups A and B
```

Unary group operators are defined as follows:

```
- A = inverse of all the elements of A
```

```
-----}#
```

```
Object group( data )
if( !check(data,{rows=1,!report}))
    error("Parameter 'data' must be a single row matrix","F")
    return
endif

data = sort(data); // check for duplicate elements
[,n] = size(data);
for i = 1:n-1
    if data(i) == data(i+1)
        error("Non-unique element","F",data);
    endif
endfor
endObject
```

```
#{-----
# Overload of max
#{-----
function [out]=max(<group> a)
    out=max(a.data)
endfunction
```

```
#{-----
# Overload of min
#{-----
function [out]=min(<group> a)
    out=min(a.data)
endfunction
```

```
#{-----
# Unary Minus
```

```
#-----  
Operator y = - (<group> a)  
    y = group(-a.data);  
endOperator  
  
#-----  
# Difference  
#-----  
operator y = -(<group> a, <group> b)  
    [,cols]=size(a.data)  
    y = null;  
    temp = null;  
    for i = 1:cols  
        loc = find(a.data(i) == b.data )  
        if( loc == null )  
            temp = [temp,a.data(i)];  
        endif  
    endfor  
    if (temp <> null)  
        y = group(temp);  
    endif  
endoperator  
  
#-----  
# Intersection  
#-----  
operator y = * (<group> a, <group> b)  
    [,cols]=size(a.data)  
    y = null;  
    temp = null;  
    for i = 1:cols  
        loc = find(a.data(i) == b.data )  
        if( loc <> null )  
            temp = [temp,a.data(i)];  
        endif  
    endfor  
    if (temp <> null)  
        y = group(temp);  
    endif  
endoperator  
  
#-----  
# Union  
#-----  
operator y = + (<group> a, <group> b)  
    c = b - a;  
    y = group( [a.data,c.data] );  
endoperator  
  
#-----  
# Equality  
#-----  
operator y = == (<group> a, <group> b)  
    y = 0  
    [,acols]=size(a.data)  
    [,bcols]=size(b.data)
```

```
    if( acols <> bcols )
        return
    endif
    res = a.data==b.data
    if( check(res,{nonzero,!report}) )
        y = 1
    endif
endoperator

#-----
# Index Operators
#-----
Operator Object(i) = y
    [r,c]=size(y);
    if ( r <> 1 & c <> 1)
        error("Invalid insertion data","F",y);
    endif
    data(i) = y;
endOperator

Operator [y] = Object(i)
    y = data(i);
endOperator

#-----
```

## 7.4.2 Limitations

- Member entities and operators cannot have their own online Help.
- You cannot explicitly define or debug a member function, command, or operator, only the object initializer. Consequently, if you alter the definition of a member entity, you must **UNDEFINE** it before the new definition can be used.
- A MathScript object cannot be passed into an LNX, but the class variables from a given instance can be passed into an LNX as other variables are.
- You cannot assign or access a variable using an expression that contains more than one dot. This implies that if an object instance contains another MSO as a class variable, you cannot directly access the class variables of the nested object. For example, the following syntax is not allowed:

```
x = obj1.obj2.var;
```

This limitation can be circumvented if you use a temporary variable:

```
temp = obj1.obj2;
x = temp.var
```

# 8

## *External Program Interface*

This chapter describes the three Xmath interfaces for user programs written in C, C++, or FORTRAN:

- The User-Callable Interface (UCI) mechanism allows a user program to call Xmath as a server.
- The LNX (LiNked eXecutable) mechanism allows a subroutine in a user program to be callable by Xmath as if it were a regular MathScript function.
- Any C or C++ program can call the functions **XmathSave()** and **XmathLoad()** to save and load Xmath data files.

### **8.1 Overview**

A user program using the LNX or UCI mechanism is termed an LNX or UCI program, or simply an LNX or UCI. [Table 8-1](#) summarizes the differences between an LNX and a UCI.

Table 8-1 **LNX and UCI Comparison**

<b>Feature</b>	<b>Comparison</b>
Purpose	A UCI starts Xmath; an LNX is started by Xmath.
Data Structure	Both use the same data structure, the <b>externType</b> .

Table 8-1 **LNx and UCI Comparison** (Continued)

Functions	<p>UCI: Must use <b>XmathStart()</b> and <b>XmathStop()</b>; must not use <b>XmathMain()</b>.</p> <p>LNx: Must use <b>XmathMain()</b>; must not use <b>XmathStart()</b> or <b>XmathStop()</b>.</p>
Build	Both must include a C header file called <code>xmathlib.h</code> and link with a library called <b>libXmath.a</b> (for UNIX) and <b>xmath.lib</b> (for Windows).
Running	<p>UCI: Start Xmath with <b>-call</b> (a switch that triggers the UCI), the program that is calling Xmath, and any other desired startup options.</p> <p>LNx: An LNx can be called just like any other MathScript function.</p>

Xmath also provides two functions, **XmathSave()** and **XmathLoad()**, which allow an external program to save and load Xmath data.

The Xmath directory `$XMATH/src` contains code examples for the LNx and UCI utilities, as well as a sample **makefile**. `$XMATH/include` has include files for LNx and UCI scripts.

### 8.1.1 LNx

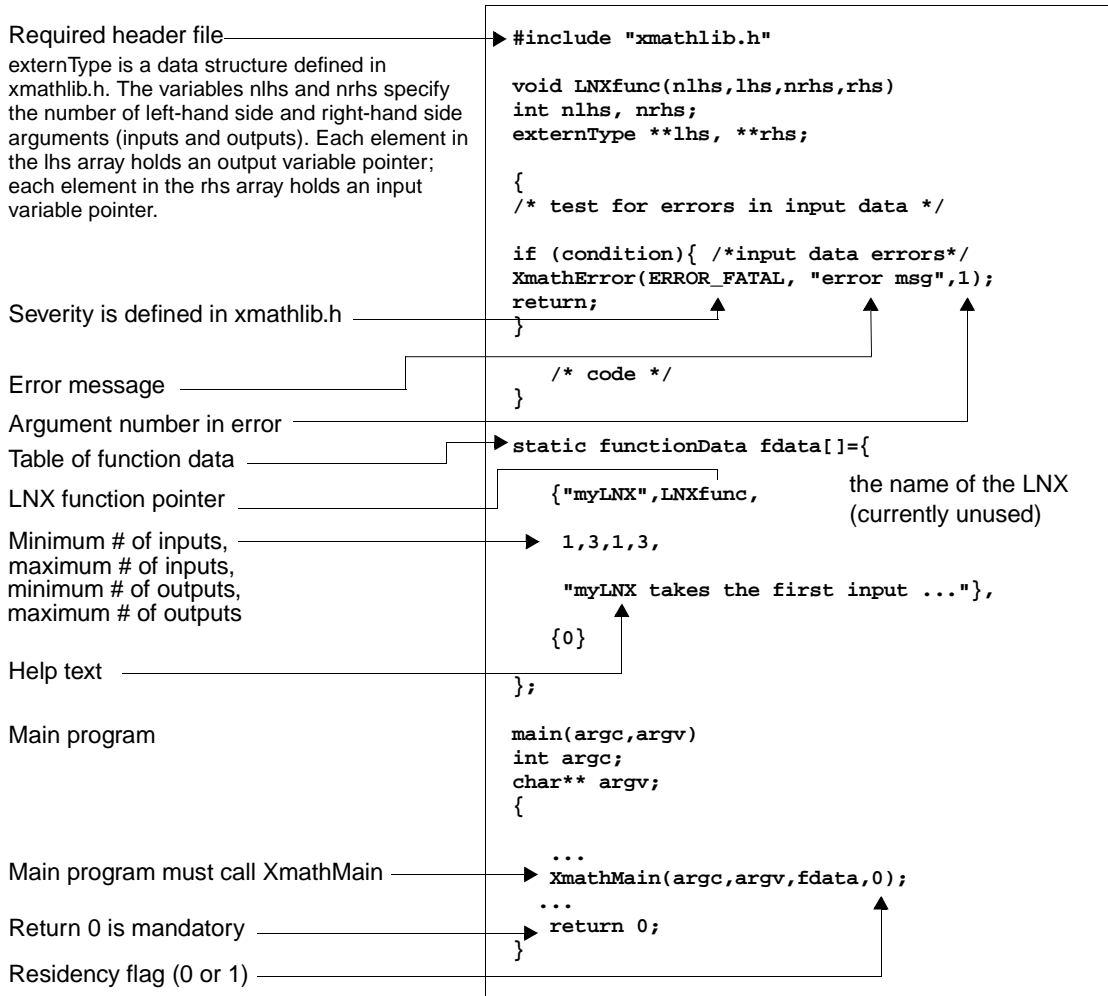
The LNx utility allows you to invoke C, C++, or FORTRAN subroutines from within Xmath. Once an LNx is built, it can be used in the same manner as any MathScript function. Furthermore, an LNx can be invoked in background mode so that it can run in parallel with Xmath.

#### Sample LNx Program

An LNx written in C program has the layout shown in [Figure 8-1](#). Each LNx program contains one LNx function. The LNx function performs a specified calculation and has the following format:

```
void LNxfunc(nlhs, lhs, nrhs, rhs)
int nlhs, nrhs;
externType **lhs, **rhs;
{
    ...
}
```

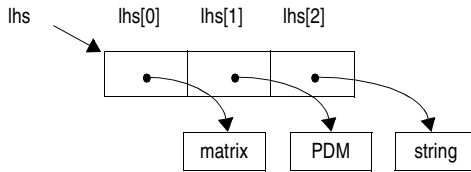
Figure 8-1 Typical C Language LNX Program Format



The input arguments reside in an array of **externType** pointers to which the variable **rhs** (right-hand side) points. The integer **nrhs** (number of right-hand side arguments) defines how many **externType** pointers are in the array.

An LNX function writes its outputs to **lhs**, which is an array of **nlhs** pointers allocated by Xmath. For example, if **nlhs=3** (indicating that your LNX was called

with three outputs), you might allocate a matrix for the first input, a PDM for the second, and a string for the third:



### 8.1.2 UCI Programs

The User Callable Interface (UCI) lets an external C program invoke Xmath as a child process, send and receive data to and from Xmath as shown in [Figure 8-2](#), and execute MathScript statements. A UCI has the layout shown in [Figure 8-3](#).

Figure 8-2 **Calling Xmath from an External Program (UCI)**

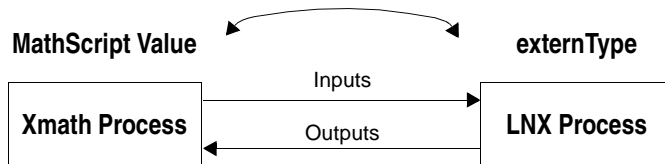


Figure 8-3 **Typical C Language UCI Program Format**

```

Required header file → #include "xmathlib.h"

Main program →
{
    ...
    XmathStart(""); /* Starts Xmath */
    /* Calls to XmathGet, XmathPut,
    XmathExecute, etc...*/
    XmathStop(""); /* Stops Xmath */
    return 0;
}

```



### 8.1.3 Compatibility

If an existing LNX or UCI compiled for an older version of `MATRIXX` is intended to be run in a new version of `MATRIXX`, we recommend that you rebuild the LNX or UCI using the new version of `MATRIXX` to maintain currency with the new compiler, DLLs, and OS supported by the new version of `MATRIXX`.

Sometimes the IPC protocol in the Xmath LNX or UCI library changes due to bug fixes and enhancements. An existing LNX or UCI must be rebuilt using the new version of the `MATRIXX` LNX or UCI library (`libXmath.a/xmath.lib`). If you attempt to run a previous version of an LNX or UCI, Xmath displays the following message:

```
Process failed to load (incompatible ipc version).
```

## 8.2 externType Data Types

The file `$XMATH/include/xmathlib.h` contains the data structures for `externType` data types and related function declarations. This file must be included in all LNX and UCI programs and programs that call `XmathSave()` (see [p.311](#)) and `XmathLoad()` (see [p.312](#)).

An `externType` is an external version of an Xmath data value such as matrix, string, and PDM. These are detailed in the following subsections.

If you allocate memory for the `externType` data type with an `Allocate*()` function, you need to remember to deallocate the memory with the corresponding `Delete*()` function, especially before re-using the variable. The function tables in this section provide the names of these functions for each data type.

### 8.2.1 Matrix Data Type

The `externType` `et_matrix` corresponds to a MathScript scalar matrix value.

```
typedef struct {
    externType et;
    int rows, columns, isReal;
    double *real, *imag;
} et_matrix;
```

The Boolean member **isReal** indicates whether the matrix is complex (**isReal** = 0) or real (**isReal** = 1).

[Table 8-2](#) lists the functions provided in the LNX functions used to allocate a new matrix, convert arrays to the matrix structure, and delete existing matrices.

Table 8-2 **et\_matrix** Functions

Function	Description and Prototype
<b>AllocateMatrix()</b>	<p>Allocates a matrix:</p> <pre>et_matrix* AllocateMatrix(int rows,int columns,int isReal);</pre>
<b>WrapMatrix()</b>	<p>Converts single or double arrays into a real or complex matrix.</p> <pre>et_matrix* WrapMatrix(int rows,int columns double* real,double* imag);</pre> <p>Both input arrays must be previously allocated and of type <b>double</b>. If the matrix is real, use the <b>NULL</b> pointer 0 as the <b>imag</b> argument.</p> <p>This function does not copy the input data; therefore, do not delete the original arrays after calling <b>WrapMatrix()</b>.</p>
<b>DeleteMatrix()</b>	<p>Deallocates storage associated with the <b>et_matrix</b> input argument.</p> <pre>void DeleteMatrix, (et_matrix* the_matrix)</pre>

## 8.2.2 String Data Type

The externType **et\_string** corresponds to the MathScript string value.

```
typedef struct {
    externType et;
    int len, rows, columns;
    char *buf;
    char **array;
} et_string;
```

- **array** is an array of **char\*** with dimensions defined by **rows** and **columns**.
- **buf** points to the string in the first row, first column of **array**. The integer **len** defines the length of this string. **len** does not have any significance for any of the other strings in **array**.

For a summary of the **et\_string** type functions, see [Table 8-3](#).

Table 8-3 **et\_string** Type Functions

Function	Description and Prototype
<b>AllocateStringMatrix()</b>	<p>Creates an <b>et_string</b> structure that can hold strings up to length <b>len</b>.</p> <pre>et_string* AllocateStringMatrix(int rows,int columns,int len)</pre> <p>The length of the string does not include the termination character.</p>
<b>WrapString()</b>	<p>Converts a previously defined string to the <b>et_string</b> data type:</p> <pre>et_string* WrapString(char *buffer);</pre>
<b>WrapStringMatrix()</b>	<p>Converts a previously allocated array of strings to a string matrix object (LNX string data type).</p> <pre>et_string* WrapStringMatrix(int rows,int columns,char** buffer)</pre> <p>Wrapping functions <b>WrapStringMatrix()</b> and <b>WrapString()</b> do not perform any copying of strings; therefore, do not delete the original input strings after calling a wrap function.</p>
<b>DeleteString()</b>	<p>Deallocates storage associated with the structure <b>et_string</b>.</p> <pre>void DeleteString, (et_string* the_string)</pre>

### 8.2.3 PDM Data Type

The PDM data structure **et\_pdm** is defined as shown below:

```
typedef struct {
    externType et;
    et_matrix *iv;
    et_string *name;
    et_string *columnNames;
    et_string *rowNames;
    et_matrix *theData;
    int rows, columns;
} et_pdm;
```

The meaning of each member is described in the following PDM:

```
testpdm=pdm([1:3; 4:6; 7:9; 10:12],101:1:104,{rowNames = "leaves",
columnNames =["birch", "elm", "oak"], domainName = "time"}):
```

Figure 8-4 shows the PDM `testpdm` and the `et_pdm` struct mapped to its parts.

Figure 8-4 Mapping the `et_pdm` Structure to a PDM

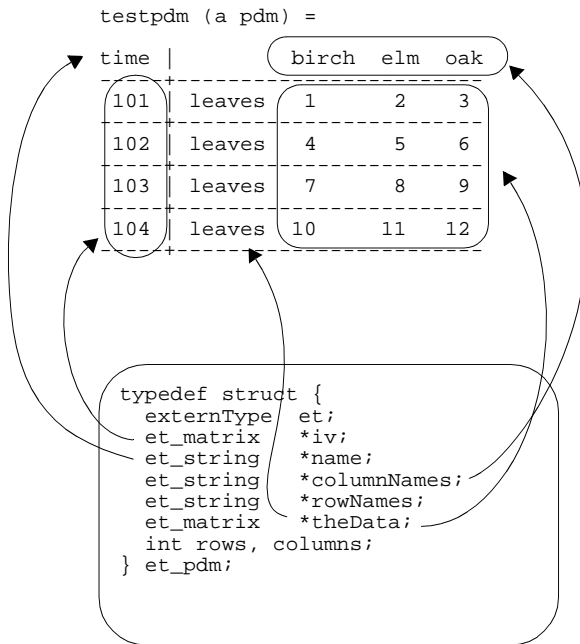
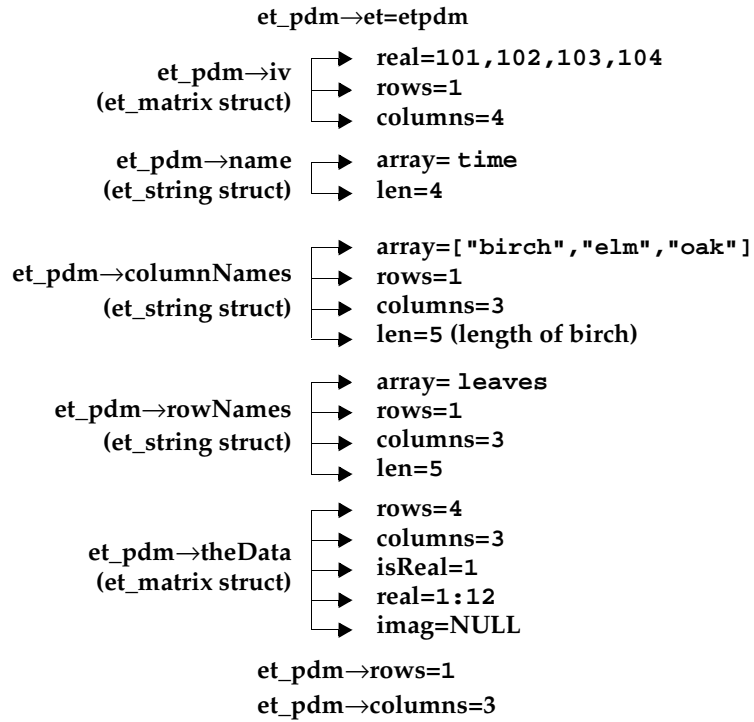


Figure 8-5 shows how the information from `testpdm` is assigned to the fields of the `et_pdm` structure. Use `AllocateMatrix()` and `AllocateStringMatrix()` to build the PDM components, and `WrapPDM()` to form the PDM. For a summary of these functions, see [Table 8-4](#).

Figure 8-5 et\_pdm Data Structure



8

Table 8-4 **et\_pdm Functions**

Function	Description and Prototype
<b>WrapPDM()</b>	<pre>et_pdm* WrapPDM(et_matrix *iv,                 et_matrix *theData,                 int rows,                 int columns,                 et_string *name,                 et_string* columnNames,                 et_string* rowNames)</pre> <p>Inputs must be previously defined using <b>AllocateMatrix()</b> and <b>AllocateStringMatrix()</b>. Like the other wrapping functions, no copying is done, so don't delete the input after the call.</p>
<b>DeletePDM()</b>	<p>Deallocates storage associated with the <b>et_pdm</b> input argument.</p> <pre>void DeletePDM(et_pdm* the_pdm)</pre>

## 8.2.4 List Data Type

The externType **et\_list** corresponds to a MathScript list object.

```
typedef struct {
    externType et;
    int nElem;          /* The number of elements in the list */
    externType** item; /* an array of pointers to the list elements */
} et_list;
```

For a summary of **et\_list** functions, see [Table 8-5](#).

Table 8-5 **et\_list Functions**

Function	Description and Prototype
<b>AllocateList()</b>	<p>Allocates a list:</p> <pre>et_list* AllocateList(int N)</pre>
<b>DeleteList()</b>	<p>Deallocates storage.</p> <pre>void DeleteList(et_list* N)</pre>

### 8.2.5 Null Data Type

The NULL data type corresponds to the Xmath NULL value (I)

Table 8-6 **et\_null Functions**

Function	Description and Prototype
<b>AllocateNull()</b>	Allocates a null: <code>et_null* AllocateNull()</code>
<b>DeleteNull()</b>	Deallocates storage. <code>void DeleteNull(et_null* N)</code>
<b>DeleteAny()</b>	(Generic deallocation) Deallocates any <b>externType</b> you allocate. <code>void DeleteAny(externType*)</code>

### 8.3 LNX and UCI Functions

The functions available for use in LNX and UCI programs (described in 8.4, p.316) are described in the following sections. A summary of these functions appears in [Table 8-7](#).

Table 8-7 **LNX Functions**

Function	Description	See Page
<b>XmathMain()</b> (for LNX only)	Sets up the communication facility and transmits information about the LNX back to Xmath; it then transfers control to your LNX function. Upon completion, the results are transmitted back to Xmath.	<a href="#">304</a>
<b>XmathCommand()</b>	Executes Xmath commands and provides access to command and error output.	<a href="#">306</a>
<b>XmathDisplay()</b>	Displays a message to the Xmath log window.	<a href="#">307</a>

Table 8-7 LNX Functions (Continued)

Function	Description	See Page
<b>XmathError()</b>	Allows you to report errors and make log entries. Severity levels are described in the file <code>\$XMATH/include/xmathlib.h</code> . The argument in error will be highlighted in the Command Window command area.	307
<b>XmathExecute()</b>	Executes Xmath commands. Xmath windows (except for the commands window and the debugger) are created as needed. <b>XmathExecute()</b> returns 0 if successful and an error string otherwise.	308
<b>XmathGet()</b>	Retrieves the value of a variable from Xmath. <b>XmathGet()</b> returns 0 if successful and an error string otherwise.	308
<b>XmathLoad()</b> (for any C or C++ program)	Creates <b>externType</b> values from an Xmath data file.	311
<b>XmathPut()</b>	Copies the contents of a data structure to the Xmath environment.	308
<b>XmathSave()</b> (for any C or C++ program)	Saves <b>externType</b> values to an Xmath data file.	311
<b>XmathStart()</b> (for UCI only)	Starts Xmath. <b>option</b> is a <b>char*</b> that is reserved for future Xmath invocation options. The option must be an empty string ("") for this version.	314
<b>XmathStop()</b> (for UCI only)	Terminates the Xmath process immediately. Modified variables will not be saved.	314

### 8.3.1 XmathMain() (for LNX only)

**XmathMain()** sets up the communication facility and transmits information about the LNX back to Xmath; it then transfers control to your LNX function. Upon completion of the LNX function, the results are transmitted back to Xmath. For an example, see [Figure 8-1](#), p.295.

```
int XmathMain(int argc, char **argv, functionData* fData, int flag);
```



The **flag** argument to **XmathMain** specifies whether the process remains resident. If this argument has the value **LNX\_RESIDENT**, the process is *resident*. It remains in memory across invocation until Xmath is exited or the LNX is undefined by issuing the **UNDEFINE** command in Xmath. If the **flag** argument to **XmathMain()** is 0, the process is *nonresident*. It is terminated after each invocation and a new process started.

If an LNX function is called often, then it is advisable to make the process resident. If the user function allocates a large amount of memory and is called infrequently, then it is more memory efficient to make the LNX nonresident.

The **functionData** data structure is typically used as follows:

```
static functionData fdata[] ={
    {"userFun",userFun,minIn, maxIn,minOut,maxOut,help},
    {0}
}
```

Figure 8-1, p.295 shows **functionData** in relation to the rest of an LNX.

- **fdata** is the name of an array that holds the function data. Although it is an array, Xmath currently uses only the first element.
- **"userFun"** is the name of this LNX; the lowercase version of this name must match the filename of the executable LNX program.
- **userFun** is the pointer to the function itself.
- **minIn**, **maxIn** are the minimum and maximum number of input arguments, respectively. For example, if **userFun** must be called with no less than two, and no more than four inputs, **minIn** is 2, and **maxIn** is 4.
- **minOut**, **maxOut** are the minimum and maximum number of output arguments, respectively.

Every time **userFun** is called, Xmath automatically verifies that the number of input and output arguments is in the valid range.

- The optional Help text entry is a char\* pointer; 0 can be used if there is no Help. The Help text can span multiple lines (as shown in Example 8-1). For an additional example on formatting Help, see `$XMATH/src/fasthilb.c`.




---

**NOTE:** You can provide a Help file for your LNX just as you can for MSFs, MSCs, and MSOs in the same directory as your LNX. If Xmath finds no Help file, it uses the optional Help text within the LNX itself. See 6.1.6, p.229 for details.

---

- The mandatory array terminator {0} comes last.

Example 8-1 **Sample Help Text**

```
/* Define the online Help */

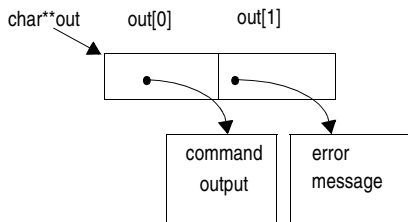
#define Help "\
Description: Produces an n x n matrix\n\
with each element multiplied by -1.\n\
\n\
Syntax:      C = negate(A)\n\
\n\
Inputs:      A is a matrix or PDM.\n\
\n\
Outputs:     C is a matrix or PDM.\n\
\n\
Examples: a = 1:10; negate(a)?\n\
\n\"
```

### 8.3.2 XmathCommand()

**XmathCommand()** is an enhanced version of **XmathExecute()** providing access to command and error output. The syntax is as follows:

```
char **XmathCommand(char *command,int options);
```

The return value of **XmathCommand()** is a static array of two pointers of type **char\***. The first pointer points to command output, or 0 if none or not requested. The second pointer points to an error message caused by the command, or 0 if none or not requested. This can be illustrated as follows:



The **options** parameter is a bit mask defined with the following macros:

- XMCMOUT** Returns command output.
- XMCMERR** Returns command errors.

Both of these macros are used in the following example:

```

char *xmcmd = "foo(bar)?";
char **out = XmathCommand(xmcmd, XMCMD_OUT | XMCMD_ERR);
if (out[0]) {
    printf("The output of \"%s\" is %s\n", xmcmd, out[0]);
    free(out[0]);
}

else
    printf("\"%s\" has no output\n", xmcmd);
if (out[1]) {
    printf("\"%s\" resulted in the error: %s\n", xmcmd, out[1]);
    free(out[1]);
}
else
    printf("\"%s\" has no errors\n", xmcmd);

```




---

**NOTE:** The error message string returned by `XmathCommand()` is memory allocated with the C library function `malloc()`. To free this string, use the C library function `free()`.

---

### 8.3.3 *XmathDisplay()*

`XmathDisplay()` displays a message to the Xmath Log window. The syntax is as follows:

```
void XmathDisplay(char *message);
```

An example of using this function follows:

```
XmathDisplay("Have a nice day.");
```

This output appears in the Xmath Log window.

### 8.3.4 *XmathError()*

`XmathError()` allows you to report fatal and warning errors as well as log entries. The syntax is as follows:

```
void XmathError(errorType error, char* message, int argNum)
```

Severity levels are described in the file `$XMATH/include/xmathlib.h`. You can specify `ERROR_FATAL`, `ERROR_WARNING` or `ERROR_LOG`. You must also specify the input argument number that is in error (a scalar between 1 and the number of right-hand side arguments), or specify 0 to indicate the function itself. The argument in error will be highlighted in the Command Window command area.

The following code fragment uses `XmathError()` to check whether the first input is a matrix.

```
if (*rhs[0] != ETMATRIX) {  
    XmathError(ERROR_FATAL, "Input must be a matrix!", 1);  
    return;  
}
```

This code fragment checks if the matrix is real:

```
x=(et_matrix*)rhs[0]  
if (!x->isReal) {  
    XmathError(ERROR_WARNING, "Matrix is not real!", 1);  
}
```

In the above example, we cast the first input into `x`, an `et_matrix` pointer, then check to see if it is real.

### 8.3.5 `XmathExecute()`

`XmathExecute()` executes Xmath commands. Xmath windows (except for the commands window and the debugger) will be opened as needed. `XmathExecute()` returns 0 if successful and an error string otherwise.

```
char *XmathExecute(char *cmd)
```

For example, this call opens the Graphics window:

```
XmathExecute("plot(random(2,3))?");
```

This call opens the Help window:

```
XmathExecute("help bode;");
```



---

**NOTE:** The command string must end with a question mark (?) or semicolon (;).

---

For an example of how to use `XmathExecute()`, see [Example 8-2](#).



---

**NOTE:** The error message string returned by `XmathExecute()` is memory allocated with the C library function `malloc()`. To free this string, use the C library function `free()`.

---

### 8.3.6 `XmathGet()` and `XmathPut()`

`XmathGet()` and `XmathPut()` retrieve and modify Xmath variable values.

## *XmathGet()*

**XmathGet()** retrieves the value of an Xmath variable. It sets the second argument to **externType\***. **XmathGet()** returns 0 if successful and an error string otherwise. The syntax is as follows:

```
char *XmathGet(char* name, externType** data)
```

For example:

```
er_string = XmathGet("data", (externType**)&data);
if (er_string != NULL)
    printf("ERROR: %s", er_string);
switch(*data) {
case ETMATRIX:
    M = (et_matrix*)data;
    break;

case ETSTRING:
    S=(et_string*)data;
    break;

case ETPDM:
    P=(et_pdm*)data;
    break;
}
```

Notice how the **externType** pointer is dereferenced to determine the actual data type.

**XmathGet()** allocates storage for Xmath variables. If you re-use the variable, be sure to deallocate the storage prior to an XmathGet call. For an example of how to use **XmathGet()**, see [Example 8-2](#).



---

**NOTE:** The error message string (**er\_string**) returned by **XmathGet()** is memory allocated with the C library function **malloc()**. To free this string, use the C library function **free()**.

---

## *XmathPut()*

**XmathPut()** creates or modifies an Xmath variable with a given data value. The first argument (*name*) must be a valid Xmath variable name. The second argument (*data*) is a pointer to one of the external types described in the [externType Data Types](#) section on [p.297](#). **XmathPut()** returns 0 if successful and an error string otherwise. The syntax is as follows:

```
char *XmathPut(char *name, externType* data)
```

For example:

```
/* allocate a real-valued Matrix struct */
x = AllocateMatrix(n, 1, 1);

/* fill up some local data */
ptx = x->real;
pty = y->real;

for (i = 0; i < n; i++) {
    *ptx = (double)i;
    *pty++ = sin(*ptx);
    *pty++ = cos(*ptx++);
}

/* send local x over to Xmath as variable x */
er_string = XmathPut("x", x);

if (er_string != NULL) {
    printf("ERROR: %s", er_string);
    free(er_string);
}
```

For an example of how to use `XmathPut()`, see [Example 8-2](#).



---

**NOTE:** The error message string (`er_string`) returned by `XmathPut()` is memory allocated with the C library function `malloc()`. To free this string, use the C library function `free()`.

---

### 8.3.7 Example Using `XmathGet()`, `XmathPut()`, and `XmathExecute()`

[Example 8-2](#) combines the use of the last three functions discussed.

Example 8-2 Using `XmathGet()`, `XmathPut()`, and `XmathExecute()`

```
n = 10;
y = AllocateMatrix(n, 2, 1);
/* fill up some local data */
pty = y->real;
for (i = 0; i < n; i++)
    *pty = (double)i;

/* copy data over to Xmath*/
er_string = XmathPut("y", y);
if (er_string != NULL) {
    printf("ERROR: %s", er_string);
    free(er_string);
}

/* execute the function */
er_string = XmathExecute("y = log(abs(y));");
```

```

if (er_string != NULL) {
    printf("ERROR: %s", er_string);
    free(er_string);
}

/* Free up existing memory associated with y
   before executing XmathGet() */
DeleteMatrix(y);
er_string = XmathGet("y", (externType**)&y);
if (er_string != NULL) {
    printf("ERROR: %s", er_string);
    free(er_string);
}

```

### 8.3.8 XmathSave( ) and XmathLoad( )

**XmathSave( )** and **XmathLoad( )** make it possible for a C or C++ program to save and load files in Xmath format without starting Xmath. Both functions make use of the **externVar** data structure:

```

typedef struct {
    char *name;
    externType *value;
} externVar;

```

The variable **name** points to the full name of the Xmath variable, which consists of the partition name and the variable name (for example, **main.var**). **value** is the standard LNX data structure pointer.

**XmathSave( )** and **XmathLoad( )** both work with an array of pointers to **externVars**, one for each Xmath variable. The name field of the last element of such an array must be a **NULL** pointer.

#### **XmathSave( )**

**XmathSave( )** has the following prototype:

```
char *XmathSave (char *filename, externVar *data, int type)
```

where

*filename* is the name of the file to be saved

*data* is an array of **externVar** defined above

*type* parameter is an integer that lets you select ASCII (value 0) or binary format (value 1)

**XmathSave** returns a **NULL** pointer for success. If this function fails, it returns a string that describes the error.

### **XmathLoad()**

**XmathLoad** has the following prototype:

```
char *XmathLoad (char *filename, externVar **data)
```

where

*filename* is the name of the file to load

*data* is an array of **externVar** defined above

**XmathLoad()** loads the specified file and constructs an array of **externVars**, one for each variable loaded, and stores the address of the array into **data**.

**XmathLoad()** returns the **NULL** pointer for success. If this function fails, it returns a string that describes the error.

### **Standard Library Linkage**

**XmathSave()** and **XmathLoad()** are declared in the LNX header file and defined in the LNX library. Therefore, a C or C++ program that calls **XmathSave()** and **XmathLoad()** should be built and invoked as an LNX or UCI.

For an alternative method of library linkage on UNIX only, see [8.6.2](#), p.335.

### **Example of XmathSave and XmathLoad**

The following example illustrates how to use **XmathSave()** and **XmathLoad()**.

Example 8-3 **XmathSave()** and **XmathLoad()**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xmathlib.h"

#define N          10
#define NAME      "main.m1"
#define FILE_NAME "call15.xmd"
```



```

int main(void)
{
    int k;
    char  name[] = NAME;
    char * status;
    et_matrix * matrix1;
    externVar * my_data, * my_data_1;

    /*== Allocate mem. for 2 data struct. type "externVar" ==*/
    my_data = (externVar *)malloc (sizeof(externVar)*2);
    /*== Backup the pointer ==*/
    my_data_1 = my_data;

    /*=====
      MUST set field "name" of LAST (#2) structure to NULL
      =====*/
    (my_data + 1)->name = NULL;

    /*== Allocate mem. for field "name" of struct. "my_data" ==*/
    my_data->name = (char *)malloc (sizeof(char) * (strlen(name)+1));

    /*== Copy str. NAME to field "name" of struct. "my_data" ==*/
    strcpy (my_data->name, name);

    /*== Allocate mem. for "et_matrix" data struct. ==*/
    matrix1 = AllocateMatrix(N, 1, 0);

    /*== Fill in some data ==*/
    for (k = 0; k < N; k++) {
        (matrix1->real)[k] = k;
        (matrix1->imag)[k] = k+1;
    }

    /*== Fill in field "value" after cast to "externType" ==*/
    my_data->value = (externType *)matrix1;

    /*== Save matrix1 (Xmath format) in file = FILE_NAME ==*/
    if (status = XmathSave(FILE_NAME, my_data)) {
        printf ("status = %s\n", status);
        return 1;
    }

    /*== Free each field of every struct. type "externVar"
      Do it in for loop until field "name" = NULL ==*/

    for (my_data = my_data_1; my_data->name; my_data++) {
        free (my_data->name);
        /*== Free mem. from AllocateMatrix() above ==*/
        DeleteAny(my_data->value);
    }

    /*== Free array of "externVar" ==*/
    free (my_data_1);

    return 0;
}

```

### 8.3.9 XmathStart( ) and XmathStop( )

The file `$XMATH/include/xmathlib.h` defines the `XmathStart( )` and `XmathStop( )`, which allow your program to communicate with Xmath. Each routine description below is followed by a prototype.

#### **XmathStart( )**

`XmathStart( )` starts Xmath. `option` is a `char*` that is reserved for future use. Currently, the option must be an empty string (`""`). This function returns the Xmath process ID (pid) if successful and 0 if unsuccessful.

```
int XmathStart(char *option)
```

#### **XmathStop( )**

`XmathStop( )` terminates the Xmath process immediately. Modified variables will not be saved. This function returns 0 if successful and 1 if unsuccessful.

```
int XmathStop()
```

### 8.3.10 Sample LNX Demonstrating Most Functions (myfun)

`myfun( )` has one input and one output. The syntax to invoke `myfun( )` is the same as for any other MathScript function:

```
y = myfun(x)
```

[Example 8-4](#) provides sample code for most of the external program interface functions.



---

**NOTE:** On UNIX systems, the filename for an LNX must be in lowercase letters.

---

Example 8-4 `myfun.c`

```
#include "xmathlib.h"
void myfun(int nlhs, externType **lhs, int nrhs,externType **rhs)
{
  et_matrix *x,*y;

  /* This function is written to indicate how you would use your  */
  /* own C code to perform operations on Xmath data objects, and is*/
  /* thus quite general. In this example, we manipulate the real  */
```

```

/* and imaginary components of the data separately. Note that */
/* these elements are DOUBLES. The next line defines storage */
/* variables for the real and imaginary components of the */
/* output data matrix. */

double *val, *ival;
int i; /* a counter variable */

/* Do some error checking. */

if (*rhs[0] != ETMATRIX) {
    XmathError(ERROR_FATAL,"Input must be a matrix!",1);
    return;
}
x=(et_matrix*)rhs[0];
if (x->columns !=1) {
    XmathError(ERROR_FATAL,"Can only work on column vectors!",1);
    return;
}
if (x->isReal) {
    XmathError(ERROR_WARNING,"Need complex input!",1);
    x->imag=(double*)calloc(x->rows,sizeof(double));
    x->isReal =0;
}

/* Pre-allocate the output y as a matrix having the same size */
/* as input x. */

y=AllocateMatrix(x->rows, 1, x->isReal);

/* The following five lines assign the real and imaginary data */
/* to the variables val and ival respectively. Then 2 is added */
/* to each of the real components and 3 to each of the imaginary */
/* components. Instead of using the dummy example here, you */
/* replace these lines with a call to a more sophisticated */
/* function of your own. */

val = y->real; ival = y->imag;
for (i = 0; i < x->rows; i++) {
    val[i] = 2.0+x->real[i];
    ival[i] = 3.0+x->imag[i];
}

/* Return y as the first--and in this case, only--output of */
/* the left side of the function call. */

lhs[0]=(externType*)y;
}
static char help[]={"This is the Help text.\n No Help yet."};
static functionData fdata[]={
    {"myfun",myfun,1,1,1,1,help},
    {0,0,0,0,0,0},
};

main(argc,argv)
int argc;

```

```
char** argv;
{
  XmathMain(argc,argv,fdata,0);
  /* This must always return 0.          */
  return 0;
}
```

## 8.4 Building and Calling LNX and UCI

In this section, we use the sample LNX file `myfun.c` (Example 8-4) to illustrate how to build an LNX. A UCI is built exactly the same as an LNX.

### 8.4.1 Building on a UNIX System

To build a makefile and call an LNX on a UNIX system:

1. Copy the sample program `myfun.c` from `$XMATH/src` to your working directory as follows:

```
copyfile "$XMATH/src/myfun.c"
```

2. `$XMATH/src/Makefile` is the makefile used to build an LNX or UCI. Copy the makefile template to your working directory:

```
copyfile "$XMATH/src/Makefile"
```

3. Edit the template to put `myfun.c` on the `NAME` line and `myfun.o` on the `USEROBJECTS` line. In addition, specify the appropriate compiler command (for example, `acc`) on the `LINK` line and appropriate compiler libraries (for example, `$(CLIBS)`) on the `LIBS` line.



---

**NOTE:** You can skip this step and use the expanded form of the make command below.

---

4. Enter the make command from the Xmath command area:

```
oscmd("make")
```

or

```
oscmd("make NAME=myfun USEROBJECTS=myfun.o
LIBS='-L$(XMATH)/lib -lXmath' LINK=acc")
```



**NOTE:** Use the simple form only if you edited the makefile.

- Once the make has run successfully, you can call `myfun()` as a regular Xmath function:

```
myfun(1 + jay)
```

### 8.4.2 Sample makefile (UNIX)

[Example 8-5](#) provides a sample makefile for an LNX or UCI. This example includes several lines that are user-editable, such as the NAME and DEFS lines. Comments in the example explain the required user inputs. In this sample, `myfun.c` is the name of the sample LNX. The required user-input fields appear in bold type, but these are normally blank and require your modification.

Example 8-5 **Sample makefile for Solaris Platform**

```
# Basic MAKEFILE for creating callable interface/lnx executable

# Following fields must be set (Makefile or command line)

# NAME          Prefix name of program that uses the callable
#               interface or of the lnx file you wish to create
# USEROBJECTS  List of .o files you wish to link with

# LIBS          Name of compiler-specific libraries (suggested
#               Solaris SC4.0 libraries pre-defined in CLIBS, CCLIBS,
#               and FLIBS
# LINK          Name of compiler or link editor

# Following fields are user-settable

# USERLIBS     List of library search paths and/or libraries
#               (e.g. library, -Lpath, and/or -llibname)
# DEFS         C or C++ pre-processor define directive
#               (e.g. -DXTFUNCPROTO)
# UCFLAG       User CFLAGS, i.e. options the user wants sent to
#               C compiler (e.g. -g)
# UCCFLAG      User CCFLAGS, i.e. options the user wants sent to
#               C++ compiler (e.g. -g)
# UFFLAG       User FFLAGS, i.e. options the user wants sent to
#               FORTRAN 77 compiler (e.g. -g)
# ULDFLAG      User LDFLAGS, i.e. options the user wants sent to
#               linker (e.g. -v)
# INCLUDE      List of directories that are searched for
#               #include files
# CC           Name of C compiler
# CCC         Name of C++ compiler
# FC           Name of FORTRAN compiler
```

```
        NAME = myfun
USEROBJECTS = myfun.o
USERLIBS =
        DEFS = -DSOLARIS
        UCFLAG =
        UCCFLAG =
        UFFLAG =
        ULDFLAG =

INCLUDE = -I. -I$(XMATH)/include

        CLIBS = -L$(XMATH)/lib -lXmath
        CCLIBS = -L$(XMATH)/lib -lXmath_cxx
# F77 and M77 are Solaris Fortran SC2.0 runtime libraries
#        FLIBS = -L$(XMATH)/lib -lXmath_cxx -lF77 -lM77
# F77, M77, and sunmath are Solaris Fortran SC3.0 and SC4.0 runtime
# libraries
        FLIBS = -L$(XMATH)/lib -lXmath_cxx -lF77 -lM77 -lsunmath
        LIBS = $(CLIBS)

        CC = acc
        CCC = CC
        FC = f77 -temp=$(HOME)
        LINK = $(CC)

        CFLAGS = $(DEFS) $(UCFLAG) $(INCLUDE)
        CCFLAGS = $(DEFS) $(UCCFLAG) $(INCLUDE)
        FFLAGS = $(UFFLAG) $(INCLUDE)
        LDFFLAGS = $(ULDFLAG)

.SUFFIXES : .o .c .cxx .C .f .F
.c.o:
        $(CC) $(CFLAGS) -c $< -o $@
.cxx.o:
        $(CCC) $(CCFLAGS) -c $< -o $@
.C.o:
        $(CCC) $(CCFLAGS) -c $< -o $@
.f.o:
        $(FC) $(FFLAGS) -c $< -o $@
.F.o:
        $(FC) $(FFLAGS) -c $< -o $@

$(NAME): $(USEROBJECTS)
        $(LINK) $(LDFFLAGS) -o $@.lnx $(USEROBJECTS) $(USERLIBS) $(LIBS)
$@echo " Done."
```

### 8.4.3 Building on a Windows System

To build a makefile and call an LNX on a Windows system in Xmath:

1. Copy the sample program `myfun.c` from `%XMATH%\src` to your working directory as follows:

```
copyfile "%XMATH%\src\myfun.c"
```

2. Enter the following command from the Xmath command area:

```
oscmd("makelnx myfun.c")
```

In general, to build LNXs and UCIs for Xmath use on a Windows system, enter the `makelnx` command with the following syntax:

```
> makelnx -debug "file1 file2 ..."
```

For the above command, the default is to build “nodebug” objects unless you specify the `-debug` option.

The above command is a batch file that calls the makefile. Here is the path to the batch file and makefile:

```
%XMATH%\bin\makelnx.bat  
%XMATH%\bin\makelnx.mk
```

Typically, you will not need to edit or change these files to perform routine build tasks. If you do need to customize your build procedures, you can copy these files to your local project directory and edit them as required.

If you do not specify a source module filename or list of filenames in the command area, the script by default will look in your local directory for a specific argument file containing the list of filenames. These default argument files require a filename extension of `.arg` and must have a name that matches the name of the corresponding build command. For example, `makelnx.arg` is used by `makelnx.bat`. In these argument files you include a list of your files to compile and link.

The filenames can be separated by spaces or placed on separate lines and any text on a line following ‘\ ’ (backslash space) will be treated as comment text.



---

**NOTE:** Filenames can be separated by spaces or placed on separate lines with a continuation character ‘\ ’ appended at the end of the previous one.

---

All target filenames specified with the above “make” commands must have a suitable file extension because this determines the choice of compiler for each file. The default file extensions currently supported include:

C	.c
C++	.cxx or .cpp or .cc
FORTRAN	.for or .f

Like most standard make facilities, the above “make” commands support conditional compilation and linking of files depending on file creation time and whether the necessary dependent files currently exist. This means that recompiles will only be done for files where source is newer than the corresponding object file. If you need to force recompilation of a source module, delete the corresponding object file.

The make commands automatically create a log in your current working directory. The log filename has an extension of **.log** (for example, **makexxx.log**). Upon completion of the make, a copy of this file remains in your local directory in case you need to review the contents of the make.

If you need to customize your builds, each of the make script source files described above contains a commented section highlighting several predefined macro strings that you can modify as needed to customize the build process. Follow the instructions provided in the files.

#### **8.4.4 Undefining an LNX**

If an existing resident LNX file is relinked while Xmath is running, use the **undefine** command to terminate the current LNX process so that the new LNX is used upon the next invocation.

#### **8.4.5 Using the User-Callable Interface**

The User Callable Interface (UCI) program uses the function **XmathStart()** to invoke Xmath. Any inputs that will be used in Xmath are copied from the user program to Xmath objects using **XmathPut()**. Once all inputs are copied over to the Xmath process, any Xmath statement can be executed using **XmathExecute()** or **XmathCommand()**. Any data transferred to Xmath and altered can be retrieved using **XmathGet()** or saved to a file using **XmathSave()**. The Xmath process is terminated using **XmathStopv()**.



### 8.4.6 Building and Calling a UCI

A UCI is created in the same way as an LNX. A UCI is invoked by specifying the `-call` option to the command to start Xmath:

```
xmath -call myuci.ext
```

```
xmath -tty -call myuci.ext
```

where `ext` = `lnx` on UNIX machines and `exe` on PCs.

Any required arguments to `myuci` can be supplied at the end of the command line.

### 8.4.7 LNX Example

Example 8-6 provides an example of the LNX function `negate()`. The `negate()` function works exactly like the minus (-) operator on matrix and PDM inputs. The function returns an error if the input is a string.

Example 8-6 `negate()`

```
#include "xmathlib.h"
void negate(nlhs, lhs, nrhs, rhs)
int nlhs, nrhs;
/* lhs is a pointer to the return arguments */
/* rhs is a pointer to the input arguments */
externType **lhs, **rhs;
{
    int number_elem, i;
    et_matrix *input;
    et_pdm *in_pdm;
    double *in_data;
    switch(*rhs[0]) {
        case ETMATRIX: {
            input = (et_matrix *)rhs[0];
            in_data = input->real;
            number_elem = input->rows * input->columns;
            for(i=0; i < number_elem; i++, in_data++)
                *in_data = -(*in_data);
            lhs[0] = (externType*)input;
            break;
        }
        case ETPDM: {
            in_pdm = (et_pdm *)rhs[0];
            in_data = in_pdm->theData->real;
            number_ele = in_pdm->theData->rows*in_pdm->theData->columns;
            for(i=0; i < number_elem; i++, in_data++)
                *in_data = -(*in_data);
            lhs[0] = (externType*)in_pdm;
            break;
        }
    }
}
```

```
    }
    default:
        XmathError(ERROR_FATAL,
            "Data Type not supported in this function", 1);
    }
}

/* Define the online Help */

#define Help "No Help yet"

/* Holds the function information: */

static functionData fdata[] = {
    {"negate", negate, 1, 1, 1, 1, help},
    {0}
};

main(argc, argv)
int argc;
char **argv;
{
    int resident = 0;
    XmathMain(argc, argv, fdata, resident);
    return 0;
}
```

### 8.4.8 UCI Examples

[Example 8-7](#) is a UCI program that uses the Xmath `log()` function to calculate the logarithm of an input. This file is found in `$XMATH/src/call.c`. [Example 8-8](#) is a UCI example that uses Xmath graphics in an external C program.

#### Example 8-7 Xmath as a Computational Engine

```
#include <math.h>
#include <stdio.h>
#include "xmathlib.h"
int doMyProgram()
{
    et_matrix *x, *y;
    double *ptx, *pty;
    int n, i;
    n = 10;

    /* allocate two matrix structs */

    x = AllocateMatrix(n, 1, 1);
    y = AllocateMatrix(n, 2, 1);

    /* fill up some local data */
```

```

ptx = x->real;
pty = y->real;
for (i = 0; i < n; i++) {
    *ptx = (double)i;
    *pty++ = sin(*ptx);
    *pty++ = cos(*ptx++);
}
/* send local x and y over to Xmath as variable y.
Check for errors*/

er_string = XmathPut("y", y);
if (er_string != NULL)
    printf("ERROR: %s", er_string);

/* execute an Xmath function */

er_string = XmathExecute("y = log(abs(y));");
if (er_string != NULL)
    printf("ERROR: %s", er_string);

/* Get y back. have to delete the current y since we
* get a new one from XmathGet.*/

DeleteMatrix(y);
er_string = XmathGet("y", (externType**)&y);
if (er_string != NULL)
    printf("ERROR: %s", er_string);

/* Output the new y */

pty = y->real;
for (i = 0; i < n; i++)
    fprintf(stdout, "%g %g\n", *pty++, *pty++);
}
int main(argc, argv)
unsigned argc;
char** argv;
{
    XmathStart("");
    doMyProgram();
    XmathStop();
    return 0;
}

```

Example 8-8 Xmath as a Graphics Engine

```

#include "xmathlib.h"
#include <stdio.h>

/* Generate some test data */
double data[8] = {0.0, 1.0, 2.0, 3.0, 4.0, 3.0, 2.0, 1.0};
int number_points = 8;

int DisplayVector(vector, columns)

```

```
double *vector;
int columns;

{
    et_matrix *thedata;
    int real = 1;
    char *er_string;

    /* Convert the data array to the data type et_matrix, so
       Xmath will recognize it*/

    thedata = WrapMatrix(1, columns, vector, 0);

    /* Copy the data over to the Xmath child process */

    er_string = XmathPut("thedata", (externType*)thedata);
    if (er_string != NULL)
        printf("ERROR: %s", er_string);

    er_string = XmathExecute("plot(thedata)?");
    if (er_string != NULL)
        printf("ERROR: %s", er_string);

    /* The plot is now drawn, and the user can interact with
       the window, adding text, changing colors, etc*/
    XmathExecute("pause");
}
int main(argc, argv)
    unsigned argc;
    char** argv;
{
    /* Start the Xmath process */
    XmathStart("");

    /* Send data to be plotted */
    DisplayVector(data, number_points);5

    /* Stop the Xmath child process */
    XmathStop();
    return 0;
}
```

Any plot can be saved to a PostScript or HPGL file using the **hardcopy** command:

```
XmathExecute("hardcopy file=\mygraph\", {ps}");
```

The C escape character `\` (backslash) is necessary for the embedded Xmath string.

### 8.4.9 Calling an LNX in Background Mode

If an LNX performs a long calculation, you can invoke the LNX in background mode so that you can continue to use Xmath for other tasks while the LNX runs.

Another scenario where a background LNX is useful is where the LNX is a GUI application (see [Advanced Background LNX Function \(IPCWC\)](#) on p.335) for information on how to communicate with a background LNX).

### Example

To invoke the LNX `myfun()` in background mode, issue the following command:

```
[output] = (define myfun, {background})(1000);
```

The return value, `output`, will be “busy” during the background LNX’s execution. In this example, `1000` is the input argument to `myfun()`.

Given the above example, typing the command `WHO` (which lists variables) in the Xmath window shows that `output` is busy:

```
who
output -- busy (job #13103)
```

After the background define command for the LNX process has been entered, the process will be spawned to run in background mode and the user will have immediate control of the Xmath command area.

Upon completion of the background LNX process, notification of the process termination status appears in the Xmath log area, after you press Return.

```
[out]=(define myfun, {background}) (1000);
(job 13103) has terminated normally.
```

[Example 8-9](#) is an example of an LNX program that can run in either foreground or background mode.

Compile this sample LNX program using the steps described in [8.4.1](#), p.316. To see how to run the sample program in background mode, refer to [Advanced Background LNX Function \(IPCWC\)](#) on p.335.

#### Example 8-9 `getpi` (Runs in Foreground or Background)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "xmathlib.h"

/* This sample lnx program calculates the value of pi based on the */
/* number of randomly-generated (x,y) points that fall within the */
/* upper right quarter of the unit circle. */
/* */
```

```

/* Test using an input value between 500000 and MAXRANDOM.          */
                                                                    */

#define REAL1
#define MAXRANDOM((double) (exp(31 * log(2.0))-1)) /* (2**31) - 1 */

void getpi(nlhs, lhs, nrhs, rhs)
int nlhs, nrhs;
externType **lhs, **rhs;
{
    externType *data;
    et_matrix *arg;
    et_matrix *out;
    long      steps;
    double    x, y, r;
    double    p_i;
    char      buffer[255], *errstr;
    int       count; /* Number of random points inside unit circle */

    if (nrhs != 1) {
        /* User did not provide an integer argument. Go to Xmath's */
        /* main partition and get the variable `step_number'.      */
        errstr = XmathGet("main.step_number", &data);
        if (errstr != NULL) {
            sprintf(buffer, "Error getting main.step_number : %s",
                errstr);
            XmathError(ERROR_FATAL, buffer, 1);
            free(errstr);
            return;
        }
    }

    if (*data != ETMATRIX) {
        XmathError(ERROR_FATAL, "Usage: getpi number", 1);
        return;
    }
    arg = (et_matrix*) data;
    XmathExecute("main.pi = 0;"); /*create the result variable*/
} else {
    /* User provided an integer argument to the lnx */
    if (*rhs[0] != ETMATRIX) {
        XmathError(ERROR_FATAL, "This LNX requires a number!", 1);
        return;
    }
    arg = (et_matrix*) rhs[0];
}

srandom((int) time(0)); /* Start random number generator */

count = 0;
for (steps = 0; steps < (int) arg->real[0]; steps++) {
    /* Get x and y coordinate values between 0 and 1 */
    x = random() / MAXRANDOM;
    y = random() / MAXRANDOM;
    r = sqrt( (x * x) + (y * y) );
    if (r <= 1.0)
        count++;
}

```

```

    p_i = 4.0 * count / steps;
    fprintf(stderr, "%ld steps: p_i = %f\n", steps, p_i);

    out = AllocateMatrix(1, 1, REAL);
    nlhs = 1;
    out->real[0] = p_i;
    lhs[0] = (externType*) out;

    if (nrhs != 1) {
        XmathPut("main.pi", (externType*) out);
        DeleteAny(data);
    }
}

functionData fdata[] =
{"getpi", getpi, 0, 1, 0, 1, "Help text for getpi" }, {0} };

main(argc, argv)
int argc;
char **argv;
{
    fprintf(stderr, "Starting ...\n");
    XmathMain(argc, argv, fdata, 0);
    fprintf(stderr, "Stopping ...\n");
    return 0;
}

```

#### 8.4.10 Removing an LNX Job

When an LNX is invoked in background mode, Xmath echoes a job number (which is really its process ID) to the log area. This job number can be used as input to the **REMOVE JOB** command.

```
REMOVE JOB job_number
```

The **REMOVE JOB** command uses the specified job number to terminate the LNX.

#### 8.4.11 Building an LNX to Link a FORTRAN Routine

Xmath provides two ways to create an LNX function based on FORTRAN code. The preferred approach is to use C as described in the previous sections and then transfer control to your FORTRAN subroutine from within C. The second method is to use the special FORTRAN interface to LNX described in this section. This approach is less complete due to limitations in FORTRAN, and it is recommended only for users who don't know C.

### Calling FORTRAN from C LNX Files

There are three important points to remember when calling a FORTRAN routine from C: name linkage, argument linkage, and array ordering.

1. (UNIX Only) In C, append an underscore (`_`) to the end of the name of the FORTRAN routine you need to call. You will need to define the FORTRAN function as a void external function within your C routine. (Some architectures do not support underscores.)
2. FORTRAN expects subroutine arguments to be passed by reference (address). Here is a sample FORTRAN subroutine:

```
subroutine fort(n, a)
double precision a(n)
integer n
```

To call the above subroutine from C, you need:

```
double *a;
int n;
fort_(&n, a)
```

Here you pass the address of `n`. Note that the variable `a` is already an address.

3. FORTRAN stores two-dimensional arrays in column-major, as opposed to row-major, mode. This means that sequential elements of a FORTRAN array that comprise the columns and sequential elements of a C array run along the rows of the array.

### Creating FORTRAN LNX Files

The C interface to LNX described above is the preferred method of presenting external FORTRAN code as an Xmath function. However, for users who may not be familiar with the C language, a FORTRAN interface that does not require any C programming is also provided.

To get started using FORTRAN LNX you may want to study the file `template.f` in `$XMATH/src`. This file is an example of how to link a FORTRAN matrix-vector multiply routine into Xmath. You must supply an initialized common block named `fdata` declared as:

```
character *10 name
integer minIn, maxIn, minOut, maxOut
common /fdata/ minIn, maxIn, minOut, maxOut, name
```



The template does this by using a block data section where it initializes the common block with data statements. These parameters have the same meaning as the fields of the **functionData** structure in 8.3.1, p.304. Currently the **name** is ignored, and the name of the LNX function will be the name of the generated LNX executable file.

You must also supply a subroutine named **ftnlrx** with the calling sequence. The template (`$XMATH/src/template.f`) gives an example of a **ftnlrx** subroutine.

```

subroutine ftnlrx(thefun,
! nin, stkin, locin, cmxin, rowin, colin,
! nout, stkout, locout, cmxout, rowout, colout,
! howmuch, error)
integer thefun
integer nin,locin(nin),cmxin(nin),rowin(nin),colin(nin)
integer nout,locout(nout),cmxout(nout),rowout(nout),colout(nout)
integer howmuch, error
double precision stkin(*), stkout(howmuch)

```

The meanings of the parameters are described in [Table 8-8](#).

Table 8-8 **ftnlrx Parameters**

Parameter	Function
<b>thefun:</b>	For future expansion. Set to 1 in this version.
<b>nin</b>	The number of input arguments.
<b>stkin</b>	A “stack” of the input matrices.
<b>locin</b>	An array indicating the index in <b>stkin</b> of each input matrix. For example, input argument 2 starts at position <b>locin(2)</b> , so the (1,1) element of input argument 2 is <b>stkin(locin(2))</b> , and the (2,1) element is <b>stkin(locin(2)+1)</b> .
<b>cmxin</b>	<b>cmxin(i)</b> is 1 if input argument <b>i</b> is complex. Zero otherwise.
<b>rowin</b>	<b>rowin(i)</b> gives the number of rows of input argument <b>i</b> .
<b>colin</b>	<b>colin(i)</b> gives the number of columns of input argument <b>i</b> .
<b>nout</b>	The number of output arguments requested by the Xmath user.
<b>stkout, locout, cmxout, rowout, and colout</b>	are analogous to <b>stkin, locin, cmxin, rowin, and colin</b> , except that they pertain to the output arguments. You are responsible for setting these values completely and correctly.

Table 8-8 **ftnlx Parameters** (Continued)

Parameter	Function
<b>howmuch</b>	Indicates how much space is reserved in <b>stkout</b> . That is, you should regard <b>stkout</b> as an array declared as <b>double precision stkout(howmuch)</b> .
<b>error</b>	a user-settable error flag.  <code>if error &gt; 0 - fatal error</code> <code>if error &lt; 0 - warning</code> <code>if error == 0 - no error</code>

The typical sequence in **ftnlx** will be to:

1. Unpack the input stack (**stkin**).
2. Pass control to your desired FORTRAN subroutine.
3. Pack the output arguments in **stkout** (and set **locout**, **cmxout**, **rowout**, and **colout**).

The routines discussed in previous sections (**XmathError**, **AllocateMatrix**, etc.) are not available in FORTRAN LNX.

## 8.5 Debugging

Debugging procedures for LNXs and UCIs involve setting breakpoints and then analyzing errant behavior versus expected behavior as described in the following sections.

### 8.5.1 Debugging an LNX with *dbx* (on UNIX Systems)

1. Create an LNX called **myfun.lnx** with debug information.  
  
You can modify the **make** command itself (see [Step 4](#), p.316) by adding the debug option (for example, **UCFLAG = -g**) or by changing the appropriate user-defined flag within the makefile itself (for example, **UCFLAG = -g** or **UCCFLAG = -g**) (see [Sample makefile for Solaris Platform](#) on p.317).
2. You must indicate that you want the debugger to ignore the USR1 interprocess signal handler.

- For dbx under SunOS, create a file called `.dbxinit` with this line:  

```
ignore USR1
```
  - On HP-UX, create a file called `.xdbrc` with this line:  

```
z 16sr
```
3. Issue the Xmath **DEBUG** command:  

```
debug myfun
```
  4. Now call the function:  

```
myfun(1+jay)
```

Xmath displays the debug LNX dialog window and then pauses. The debug message dialog will have a message similar to,

```
dbx ./myfun.lnx 8134
```

where dbx is followed by the LNX function and the process ID.
  5. To start the dbx process with the LNX process attached, type or copy the above command into a UNIX shell.
  6. In dbx, set a breakpoint in `myfun()` with the command:  

```
stop in myfun
```
  7. Issue the dbx continue command by typing **cont** in the debugger.
  8. Return to Xmath and dismiss the debug LNX dialog.  
Immediately, dbx breaks at the breakpoint previously set. You can start debugging the function.
  9. When you finish debugging the function, issue the dbx **CONT** command.  
Xmath returns with the output of the LNX function.
  10. When the debug session is complete, use the dbx **DETACH** command to detach the LNX process from dbx.

For resident functions, Xmath automatically turns off debug mode for **LNXname** after it returns. If you want to debug the LNX function with another set of inputs, call **LNXname** again. This time, however, Xmath will not display the debug dialog. On the other hand, if you haven't removed the breakpoint in dbx, the LNX process will break at the same breakpoint. The function can then be debugged with the new inputs.

Specifying an LNX to be nonresident means that the LNX is automatically undefined after it finishes. Therefore, the debugging mode is forgotten. This makes MSF and LNX debug mode behavior consistent, because undefining an MSF also makes Xmath forget everything about the MSF, including the debug mode.

### 8.5.2 Debugging LNXs (on Windows systems)

To debug an LNX, use the following procedure:

1. Create an LNX called **myfun.exe** with debug information as described in [Building on a Windows System](#) on p.319:

```
makelnx (-debug) myfun.c
```

This creates an LNX called **myfun.exe**.

2. Go to Xmath Commands window and call the LNX:

```
debug myfun
```

```
myfun(1+jay)
```

3. A dialog box (**myfun.exe-Application Error**) appears with the message:

**A breakpoint has been reached. Click Cancel to go into the debugger.**

Then another dialog appears with the message:

**Break caused by hard coded breakpoint instruction.**

Click OK in this dialog.

4. Now, select Debug→Breakpoints.

A Breakpoints dialog appears.

5. In the Location area, enter **myfun**. Click Add to add the name to the breakpoints column. Click OK to dismiss the dialog.

6. Select Debug→Go from the Debug pull-down menu.

The debugger will now stop at the breakpoints you have specified.

7. When you are finished debugging, select Debug→Breakpoints. When the Breakpoints dialog appears, click Clear All to clear the breakpoints. Click OK to dismiss the dialog box. Select Debug→Go.

The LNX will run to completion.

---

➔ **NOTE:** Do not exit the debugger until the LNX runs to a completion.

---

8. To exit the debugger window, select File→Exit.

### 8.5.3 Debugging UCIs (on UNIX systems)

To debug a UCI on a UNIX system, perform the following procedure:

1. Create a UCI with debug information as described in [Building on a UNIX System](#) on p.316.

2. Debug the UCI using **dbx**:

```
xmath -call dbx uci.lnx
```

3. Now, set a breakpoint in **myfun.c** with the command:

```
stop in myfun
```

For each function you want to debug.

4. Enter **run**.

The debugger will now stop at the breakpoints you have specified.

5. When you are finished debugging, clear the breakpoints and type **cont** to let the UCI run to completion.

---

➔ **NOTE:** Do not exit the debugger until the UCI runs to a completion.

---

6. To exit the debugger, type **quit**.

### 8.5.4 Debugging UCIs (on Windows systems)

To debug a UCI on a Windows system, perform the following procedure:

1. Create a UCI with debug information as described in [Building on a Windows System](#) on p.319.

2. Debug the UCI using MSVC:

```
xmath -call msdev uci.exe
```

3. Now, select Debug→Breakpoints.

A Breakpoints dialog appears.

4. In the Location area, enter the name of the function you want to debug. Click Add to add the name to the breakpoints column. Keep doing this for all of the desired breakpoints. Click OK when you have finished.
5. Select Debug→Go.  
The debugger will now stop at the breakpoints you have specified.
6. When you are finished debugging, select Debug→Breakpoints. When the Breakpoints dialog appears, click on Clear All to clear the breakpoints. Click OK to dismiss the dialog. Select Debug→Go from the Debug pull-down menu to let the UCI run to completion.



---

**NOTE:** Do not exit the debugger until the UCI runs to a completion.

---

7. To exit the debugger, select File→Exit.

## 8.6 Advanced Topics

### 8.6.1 Handling an Aborted LNX

The following MathScript command

```
set debugonerror off
```

allows a script to resume execution after an LNX that it calls terminates abnormally. Without using this command, a script will be aborted if the LNX that it calls terminates abnormally.

For example:

```
command callsegv
  set debugonerr off      # allow this script to resume if segv() aborted
  out = []                # assuming segv() never returns a []
  out = segv()            # an LNX that terminates abnormally
  if out == []            # if segv() aborted,
    display "segv() failed."
  else
    display "segv() returned successfully."
  endif
endcommand
```

If an LNX process terminates abnormally, Xmath prints out a message similar to the following:

```
Process name has terminated abnormally (Signal #)
```

The signal number is the UNIX error code. These codes are standard on UNIX systems and are described in the file `/usr/include/sys/signal.h`.



**NOTE:** `XmathPanic` should be in your LNX or UCI program's Ctrl-C signal handler to clean up after an abnormal stop. The syntax is as follows:

```
void XmathPanic()
```

## 8.6.2 Advanced Features and Notes

**On UNIX systems only:**

- When an `XmathSave()` or `XmathLoad()` link is called, an Xmath process called `xmathsl` is invoked. To avoid this overhead, you can link with the `libxmsl.a` library in addition to `libXmath.a` (`libXmath.a` must follow `libxmsl.a` in the `link` command). You will need the standard C++ library supported for your platform for the link, typically by including `-lC` in the link command line.

That is, the standalone saveload document references the last line of this file:

```
$(CC) -o $@.lnx $(USEROBJECTS) -L$(LIBS) $(USERLIBS) -lXmath
```

which must be changed to the following:

```
$(CC) -o $@.lnx $(USEROBJECTS) -L$(LIBS) $(USERLIBS) -lxmsl  
-lXmath -lC
```

- LNX and UCI use the signal `USR1` as part of communications processes; do not modify this signal's handler.

## 8.6.3 Advanced Background LNX Function (IPCWC)

**IPCWC** allows you to communicate with a background LNX process that is also a windows client. First, a message is sent to the LNX with the specified window ID (`wid`) and the process ID (`pid`). Additional data (the arguments listed) is then sent to the LNX (formatted according to the specifiers in the format string, as applicable). The calling syntax is:

IPCWC *wid*, *pid*, *format\_string*, *arg\_list* ...

*wid* Window ID (a number).  
*pid* Process ID (a number).  
*format\_string* A string with format specifiers (as described below).  
*arg\_list* The values to be sent. You can have as many values as you like, as long as they are separated by commas and each one maps to a format specifier in **format\_string**.

- The format specifiers are codes consisting of the percent sign (%) and a character. They are:
  - %c A single character.
  - %d A decimal number.
  - %s A string.
- All non-format specifiers are sent as individual characters.
- The LNX process receives data with the calls shown in [Table 8-9](#). **XmathIPCgets** returns a malloc'ed string. Remember to free it when done.

Table 8-9 **Background LNX Functions**

Function	Description and Prototype
<b>XmathIPCgetc()</b>	<b>XmathIPCgetc()</b> returns a character from the IPC stream to the LNX process.  <code>char XmathIPCgetc()</code>
<b>XmathIPCgeti()</b>	<b>XmathIPCgeti()</b> returns an integer from the IPC stream to the LNX process.  <code>int XmathIPCgeti()</code>
<b>XmathIPCgets()</b>	<b>XmathIPCgets()</b> returns a malloc string from the IPC stream to the LNX process. Remember to free the string when you are done.  <code>char *XmathIPCgets()</code>

- `$XMATH/include/xmathlib.h` contains the definition for optional flags, such as `LNX_USE_IPC`. In a background call, **XmathReleaseIPC()** detaches an LNX. The last argument in the **XmathMain()** call sets the `LNX_USE_IPC` flag. The callback LNX function, defined in the **functionData** structure, is responsible for calling **XmathReleaseIPC()**.



### Sample IPCWC Calling Sequence

The following sample IPCWC calling sequence sends the character H followed by the number 104 to an LNX that has window ID 9999 and process ID 99:

```
ipcwc 9999, 99, "H%d", 104
```

The next step is to send the character B followed by character A, the string "Test1", and then the ID number 5 to an LNX that has window ID 9999 and process ID 99.

```
ipcwc 9999, 99, "B%c%s%d", "A", "Test1", 5
```



---

**NOTE:** To ensure proper handshaking between the client and server in sophisticated LNXs, the client program should wait for a status from the client; when the client has finished reading it should return the status via `XmathIPCputs()`. For example:

---

```
ipcwc 9999, 99, "B%c%s%d&s", "A", "Test1", 5, status
```

[Example 8-10](#) shows a pseudocode LNX example that uses some of the `XmathIPCget` call. [Example 8-11](#) is pseudo-code for a sample LNX program using IPCWC.

#### Example 8-10 Sample Usage of `ipcwc` to Communicate with a Background LNX

```
#
# action = SaveFile or LoadFile
#
Command SendAction action, file_name

wid = 9999;
pid = 99;

if (!is(action, {string}))
    error("Argument 'action' must be a string", "F")
endif

if (!is(file_name, {string}))
    error("Argument 'file_name' must be a string", "F")
endif

ipcwc wid, pid, "%c %s", stringex(action, 1, 1), file_name

endCommand
```

#### Example 8-11 Pseudo-Code for an LNX that Responds to `ipcwc`

```
#ifdef UNIX
```

```

#include <X11/Xlib.h>

/* This is how the ipcwc command actually sends to the client window */
void Send_ipcwc_window_message(Window wid)
{
    XEvent xclient;
    extern Display *dpy;
    xclient.xclient.message_type = 0;
    xclient.xclient.type = ClientMessage;
    strcpy(xclient.xclient.data.b, "XMATH");
    xclient.xclient.format = 8;
    XSendEvent(dpy, wid, 0, NoEventMask, &xclient);
    XFlush(dpy);
}

/* This is how to detect a window message sent by the ipcwc command */
/* The following is a typical X event loop */
...
XEvent event;
switch (event.type) {
case ClientMessage:
if (Is_ipcwc_window_message(&event)) {
    action = XmathIPCgetc()
    switch (action) {
        case 'S':
            savefile = XmathIPCgets()
        ...
        case 'L':
            loadfile = XmathIPCgets()
            ...
            default:
                ...
            }
    }
}
else {
    /* other ClientMessage messages */
}

int Is_ipcwc_window_message(XEvent *event)
{
    extern Display *dpy;
    XClientMessageEvent *xclient;
    Atom wmpAtom, wmdAtom;
    xclient = (XClientMessageEvent *) event;
    wmpAtom = XInternAtom( dpy, "WM_PROTOCOLS", True );
    wmdAtom = XInternAtom( dpy, "WM_DELETE_WINDOW", True );
    return ((wmpAtom == None || wmdAtom == None ||
    xclient->message_type != wmpAtom ||
    xclient->data.l[0] != wmdAtom)
    && !strcmp( "XMATH", xclient->data.b ));
}

#else

#include <windows.h>

```

```
/* This is how the ipcwc command actually sends to the client window */
void Send_ipcwc_window_message(HWND hwnd)
{
    PostMessage(hwnd, WM_USER, 0, 0);
}

/* This is how to detect a window message sent by the ipcwc command */
/* The following is a typical Windows event loop */
...
switch (message) {
    case WM_USER:
        /* ipcwc Window message detected */
}
#endif
```



# 9

## *Graphical User Interface*

This chapter introduces Xmath's fully programmable graphical user interface (PGUI or GUI).

The GUI is available on all MATRIX<sub>X</sub> platforms. The GUI allows arbitrary windows to be created and manipulated using only Xmath source code (MathScript). GUI windows might contain, for example, sliders, pushbuttons, menus, and plot areas, all of which can accept user input from the mouse. Xmath simultaneously supports user interaction in any number of newly created GUI windows, as well as through each of its standard windows.

The GUI provides a number of predefined dialogs that can be used to interact with the user. These dialogs are a collection of modal dialogs that are used by most applications. When called they suspend command execution until the user responds to the dialog. Once the user responds, the response is returned and command execution resumes.

### **9.1 Finding Out About the GUI**

Whether you are a GUI tool user or a developer, you will want to learn about the GUI, although the ultimate learning will be at different levels.

### 9.1.1 GUI Tool Users

GUI tools are simple and intuitive to use, but there are a few basic things you should know. You should run **guidemo** and look at some of the examples, especially **leadlag**, to get a feel for the features and capabilities of GUI tools. Each GUI tool has extensive Help menus describing its use. Browsing through Help messages is a good way to learn what a tool does.

### 9.1.2 GUI Developers

You might also want to develop your own GUI tools. For example, you might add a graphical user interface to an existing Xmath command script. Programming with the GUI is more difficult than writing your own Xmath commands and functions, so delay trying this until you are quite comfortable programming in Xmath and using GUI tools.

To develop your own simple tools using the GUI, we recommend that you run the GUI demos while looking at the corresponding source code, which is in **\$XMATH/demos/gui**. The next step is to read the Help entries for the GUI functions in the MATRIX<sub>X</sub> online Help. Each function has an example, consisting of an Xmath command that creates a PGUI tool. Start with **uiToolCreate()**. For this and other examples where an Xmath command is defined:

- Use a text editor to create a new Xmath command file.
- Copy the example command script into the file.
- Name the file *commandname.msc* and save it to a folder included in the lookup path.
- Execute the command by typing its name in the Xmath command area.

For example, to run the **uiToolCreate()** example, copy the entire **ex\_uiTool** command to a file named **ex\_uiTool.msc**. Save that file to a folder in your lookup path. In the Xmath command area, type **ex\_uiTool** and press Enter.

### 9.1.3 Running the GUI Demos

To see a menu of Programmable GUI examples, type **guidemo** in the Xmath command area. This displays the menu shown in [Figure 9-1](#).

Figure 9-1 Programmable GUI Examples

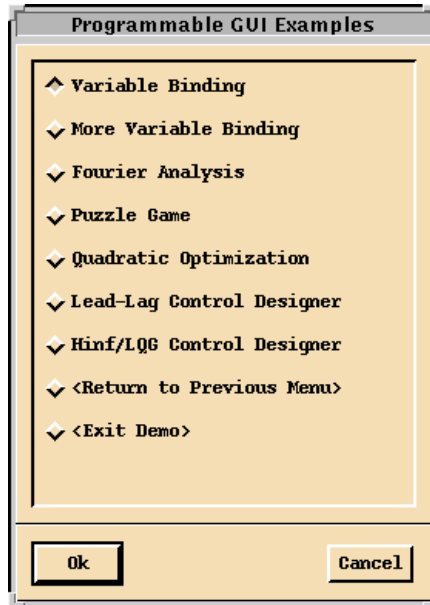


Figure 9-1 lists a number of GUI demos. You can run several demos at once.

To run a demo:

1. Select a demo (for example, Variable Binding).
2. Click OK.

In a few seconds the demo appears. (Your window manager may require you to position the window(s) generated by the demo.)

Each demo has a Help menu in its menu bar (near the upper right side of the window). The Help messages explain how to interact with the demo and what it does. It may be helpful to read the rest of this chapter before (or while) you try the demos.

3. To exit a demo, select Special→Exit or File→Exit from the individual demo window.

To see another example of a GUI implementation, type `ifilter` in the commands window command area.

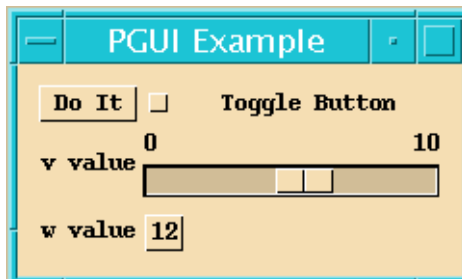
## 9.2 Interacting with a GUI Application

This section describes the mechanics of interacting with GUI windows. First, we create an example dialog and then we discuss the various kinds of GUI objects that you can place in a dialog or window and how to use them.

### 9.2.1 Creating an Example Dialog

Tools that use the GUI create windows that contain control elements such as pushbuttons, sliders, pulldown menus, plots, and lists. Some of these elements are shown in [Figure 9-2](#), the PGUI Example dialog.

Figure 9-2 PGUI Example Dialog



If you are a user only, you might want to just create the dialog without paying much attention to the individual commands that follow. If you are a developer, this is another example from which you can learn.

To create the dialog in [Figure 9-2](#), type the following in the Xmath command area:

```
t1 = uiToolCreate("guiexhelp");
mw = uiWindow(t1,{title="PGUI Example"});
tb = uiTable(mw,{height = 200, columns = 2});
void = uiButton(tb,{ text = "Do It"});
void = uiButton(tb,{ type = "toggle", text = "Toggle Button"});
void = uiLabel(tb,{ text = "v value"});
void = uiSlider(tb, {varname = "main.v", min = 0, max = 10});
void = uiVarEdit(tb, {varname = "main.w", text = "w value" });
void = uiShow(mw);
main.v = 5;
main.w = 12;
```

To kill the dialog in [Figure 9-2](#) type:

```
uiDestroy("guiexhelp")
```



## 9.2.2 Controlling GUI Objects

You can control most functions with the left mouse button. For example, you can activate a button by placing the mouse pointer anywhere on the button and clicking the left mouse button. The PGUI Example dialog has two buttons: Do It and 12.

Other objects behave as follows:

- A *toggle button* (square shaped) is either on or off. Its indicator is filled in when it is on. It can be toggled by pointing and clicking the left mouse button. The toggle button shown in [Figure 9-2](#) is off. Activating a toggle button causes some action to be performed.
- *Radio buttons* (diamond shaped) are a group of buttons with “radio” behavior. Like the station selection buttons on a radio, selecting one button automatically turns off any other button that is on.
- A *pull-down menu* is displayed by depressing and holding the left mouse button. As the mouse is dragged, the various menu selections (usually pushbuttons) are highlighted. Releasing the mouse activates the selected button.

A *cascade menu* is indicated by a small arrow to the right of the text in the button. The cascade menu is displayed by moving the mouse to the right.

- A *text entry area* behaves like the command input area in Xmath. Input is terminated by a newline character. Before you can type into a text entry area, you must focus on the area by placing the mouse pointer in the area and clicking the left mouse button. Focus is indicated by a border highlight.
- A *list* is a vertical list of items (strings) that can be selected (highlighted). Depending on the application, a list can be configured to allow various types of selection:
  - A single-selection list allows only a single line to be selected. Clicking the left mouse button selects a line.
  - A multiple-selection list allows multiple lines to be selected. The selection of a single line is toggled by clicking with the left mouse button.
  - An extended-selection list also allows multiple lines to be selected. A contiguous range of items can be selected by pressing the left mouse button, dragging the mouse, and releasing. Depressing Shift and the left mouse button selects all the items from the current item to the previous item that was selected with the left mouse button. Depressing Control and the left mouse button augments (rather than replaces) the existing

selections. This allows discontinuous ranges of items to be selected. This type of list is used in the history sorting and history column dialogs in the **leadlag** demo.

Once you select one or more items from a list, you then choose some action such as Delete or Display.

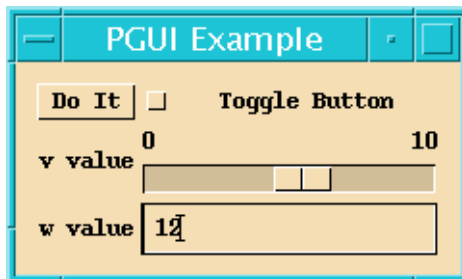
- A *dialog* is a small window that can contain a message and one or more buttons. For example, a dialog might have a single button and a message giving a warning or indicating an error.

Usually a dialog is *modal*—that is, you cannot interact with any other GUI or Xmath window until the dialog has been closed. If you find you can't interact with Xmath or other GUI windows, then look for a modal dialog that might have been accidentally covered by another window.

- *Help messages* are often listed under a Help pulldown menu at the top-right of the GUI window. The Help message appears in a new window that provides scroll bars as needed. The scroll bars are operated with the left and middle mouse buttons. The window is dismissed by selecting the Close button.
- A *variable edit box* appears in a GUI window as a button that displays some value. The value can be changed by selecting the button, whereupon a text entry area appears in place of the button. You can type a new value followed by Return. If the GUI tool doesn't like your new value, it reserves the right to change it to an acceptable value that is displayed again on the button.

The pushbutton labeled 12 shown in [Figure 9-2](#), p.344 is a variable edit box (displaying the value of the variable **w**). If you press this button, it is replaced by the “w value” text entry area as shown in [Figure 9-3](#). After entering a value from the keyboard, the text entry area is replaced by a button that contains the new value.

Figure 9-3 PGUI Example Dialog after Pressing the 12 Button



- A *slider* resembles a linear potentiometer and its value is changed by a linear motion of the handle. The position of the slider's handle represents its value. Usually the limits of the slider are shown at its ends. [Figure 9-3](#) shows a slider with minimum value 0 and maximum value 10. Its current value is about 6. You can change the value of a slider in several ways:
  - Place the mouse pointer on the handle, depress the left mouse button, and drag the handle to the desired location. Some GUI tools might do something (for example, change a plot) as you drag the handle. In other cases, nothing happens until you release the handle at the new value.
  - Click the middle button at the new value.
  - Click the left button away from the handle to increase or decrease the value a small amount. Holding the button down makes the handle steadily move towards the cursor.

Often a value is displayed with a slider and a variable edit box (for example, the **leadlag** demo). This allows the value to be changed either by dragging the slider or entering a new value via the keyboard.

- *Plots*, which can accept graphical input from the user, can also appear in GUI windows. You can use the left mouse button for graphical input, the middle for plot zooming, and the right for plot data value viewing:
    - The function of the left mouse button depends upon the particular tool and plot. Often a tool allows a curve to be grabbed and dragged by depressing the left mouse button with the cursor near the curve, dragging the mouse with the button down, and then releasing at a new position.
    - Pressing the middle mouse button anywhere in the plot creates a box containing a magnification of a small area of the plot centered at the cursor. The middle mouse button can be held down and dragged, which creates an effect similar to dragging a magnifying glass across the plot. The center of the zoomed window corresponds to the tip of the cursor.
- Pressing Control with the middle mouse button increases the size of the magnified box. Pressing Shift with the middle mouse button increases the zoom factor. Pressing Shift *and* Control with middle mouse button yields a large zoom box with a large magnification factor.
- By pointing at or near a curve or object in a plot and pressing the right mouse button, a small window appears; it identifies the curve or object and gives the coordinates and index of the nearest data value.

If you press and drag the right mouse button, the selected curve is tracked, even if another curve comes close.

Pressing Shift along with the right mouse button allows the user to get values on the piecewise line curve that interpolates the data values. In this case “index 45.7” means that the selected plot point is between the 45th and 46th curve index entries.

### 9.3 GUI Programming Overview

The Programmable GUI allows you to perform the following tasks:

- Design the layout and appearance of windows.
- Create, destroy, and manipulate these windows.
- Bind Xmath variables to various objects in the windows.
- Arrange for Xmath code to be executed when the user interacts with the windows.

These tasks are accomplished as follows:

- Windows are created, destroyed, and manipulated using a number of Xmath functions.
- Bindings between Xmath variables and sliders, pushbuttons, plotted curves, and other objects in the GUI windows are specified by setting the appropriate widget attribute.
- The execution of particular pieces of Xmath code when the user interacts with a GUI window is also specified by setting the appropriate widget attribute.

These tasks are described in more detail later in this chapter and in the MATRIX<sub>X</sub> online Help.

### 9.4 Concepts and Terminology

A single GUI application is called a tool. The components that make up a complete tool are described in the following section. Usually a user explicitly

starts a tool by sending a command (MSC) to Xmath. The MSC calls some Xmath functions that tell the GUI to create a new tool, one or more windows, and their children widgets. This is what happens when you type **guidemo**. After the MSC creates one or more initial windows, the MSC returns and Xmath is again idle. Tools can be launched in other ways. For example, an MSF, script file, or another tool can launch new tools.

Once a tool is created, it is then used as the parent of all subsequent windows created. Each window is then in turn used as the parent of each widget in that window. In this way a hierarchy of the tool is defined. As it is created, each object is given various attributes that define different aspects of appearance and behavior, including the binding hooks back to Xmath. The binding of variables to various objects on a window is a key feature of the GUI. For example, a variable can be bound to a slider in a window. Whenever the user moves the slider, the Xmath variable is updated. Similarly, whenever the Xmath variable is updated, the slider moves. Variables can also be bound to plotted curves: whenever the variable is changed, the plotted curve changes accordingly. With variable binding, you don't have to explicitly update a display; merely changing the value of the variable (reassigning it) causes all displays bound to the variable to update automatically.

A second key feature of the GUI is the Xmath callback. In itself, updating a variable when the user moves a slider isn't useful. Every time the user interacts with a window (that is, moves a slider or selects a pushbutton), you can specify certain Xmath code to be executed through an Xmath callback. An Xmath callback simply means that the tool's MSC is called with arguments that describe what the user just did. Based on these arguments, the MSC can take whatever action is required.

The GUI is event driven. Normally, Xmath is idle. When the user does something to a GUI window, variables, if any, are updated, and Xmath callback(s), if any, are executed. Once the Xmath callbacks finish (that is, the MSC returns), Xmath is again idle, waiting for a new event.

### 9.4.1 Conceptual Example

A conceptual example can show how these features work together to form a simple tool. Suppose we have some Xmath code **compute<sub>y</sub>.msc** that computes some value  $y$  given some parameter value  $x$ . Our tool arranges for the variable  $y$  to be bound to a read-only slider and the variable  $x$  to be bound to an interactive slider. The tool arranges for the Xmath code **compute<sub>y</sub>.msc** to be executed when the interactive slider is released.

When our tool is invoked, a window containing the sliders appears. When the user moves and releases the interactive slider, the variable  $x$  is first updated (assigned its new value) and then the Xmath code is executed (using the new value of  $x$ ). The Xmath code assigns a new value to the variable  $y$ . Since  $y$  is bound to the read-only slider, the read-only slider changes to reflect the new value of  $y$ .

It is interesting to compare the original Xmath code with the tool described above from the user's point of view. The user interacts with the original code by repeatedly typing commands into the Xmath Command window such as  $x=3.2$  followed by **compute $y$**  followed by  $y$ , which prints the new value of  $y$  to the Xmath log area. Thus, user input and output are via the Xmath command and log areas, respectively, and both are alphanumeric in form.

In contrast, the user interacts with the tool described above by simply grabbing and moving the interactive slider. After it is released, the new value of  $y$  is displayed on the read-only slider. Thus, user input and output are via the sliders in the tool's window and graphical in form. In effect, we have implemented a completely graphical interface for our original Xmath code **compute $y$ .msc**. In fact, once the graphical tool is running, we can iconify all of the standard Xmath windows, and someone completely unfamiliar with Xmath can use the code **compute $y$**  through the slider and bargraph.

## 9.4.2 Anatomy of a GUI Tool

It is possible to type commands directly in the Xmath Command window that instruct the GUI to create a tool and windows. Usually, however, a GUI tool consists of MathScript Command files (MSCs), MathScript Function files (MSFs), and a Help file (.hlp):

- An MSC contains the code for starting the GUI tool and all the code for the Xmath callbacks. An Xmath callback simply calls the MSC with particular arguments, and the MSC takes the corresponding action based on these arguments. If the tool is smaller, the MSC may also contain all the widget creation code as well. A large tool can consist of multiple MSCs. Usually though all the tool callbacks are in one MSC. The tool's MSC filename is the tool name followed by the extension, **.msc**.
- MSFs are often used if the tool is quite large. An MSF can help organize and group widget creation code to a particular window or functionality. The MSC can call an MSF at the appropriate time to create portions of the tools GUI as needed.

- A Help file contains one or more Help messages or strings. The tool's Help file is the tool name followed by the extension, `.hlp`.

These files are described in more detail in the following sections. See the GUI demos in `$XMATH/demos/gui` for examples of each of these files. Each of the demos is implemented as an MSC script, possibly an MSF script, and an ASCII file that contains the Help message text and global plot options. You can develop and debug GUI applications rapidly with Xmath's interactive environment and debugger.

### 9.4.3 MSC File

The tool's MSC is declared with three arguments:

```
command MSC_name {fragname, widgetname, instance}
```

When an Xmath callback occurs, the MSC is called with two strings (**fragname** and **widgetname**), and an integer (**instance**). The string **fragname** is the name of the Xmath code fragment to execute. The string **widgetname** is the name of the widget that caused the callback (usually this will be ignored, unless a single Xmath code fragment needs to handle user input into different widgets). Finally, the instance number uniquely identifies multiple instances of the same window. For example, if two identical windows are instantiated (see the `MATRIXx` online Help *uiWindow* topic) and the user selects a pushbutton on each window, one Xmath callback will have **instance = 1**, and the other will have **instance = 2**.

Usually each Xmath code fragment is executed using **goto**, so each Xmath code fragment name is written as a **goto** label. Also, when the MSC is invoked with no arguments, it is often convenient (but not necessary) to arrange that the tool itself be launched. Therefore, a template MSC appears as follows:

```
Command MSC_name {fragName, widgetName, instance}

if (exist(fragName))
    goto *fragName;
else
    # start tool
    [CODE LAUNCH TOOL GOES HERE]
    return;
endif

<ButtonPressed>          #executed when fragname == "ButtonPressed"
    [CODE TO EXECUTE WHEN BUTTON IS PRESSED]
    return;

<SliderMoved>           #executed when fragname == "SliderMoved"
    [CODE TO EXECUTE WHEN SLIDER IS MOVED]
    return;
```

```
<DoQuit>                # executed when fragname == "DoQuit"
    [CODE TO QUIT TOOL]
    return;

endCommand
```

### 9.4.4 Help File

The tool's Help file is where the tool's Help messages are stored. Each Help message (or Help fragment) is preceded by a name or label. The name is used to refer to the particular Help fragment. The order of the Help fragments in the Help file is not important. The Help file can be quite large if necessary; fragments are read only when needed.

Each Help fragment has the form:

```
<helpFragName>
    This is the Help text that will be displayed.
    The Help text can contain many lines. The indent of
    the initial line is stripped from all lines.
    # comment lines (lines starting with '#') are ignored,
    # although an embedded '#' will not be treated
    # specially. Use '\#' at the start of a line if
    # you need a '#' at the start of a non-comment line.
```

The Help fragment name **helpFragName** is any string of your choice. The indent of the initial line of the Help fragment will be stripped off all the lines in the Help fragment when the Help fragment is displayed. This assists in the legibility of the Help file.

One Help fragment can be included inside another with an **include** directive:

```
<helpFrag1>
    Note that:
    !#include <helpFrag1>
        That's all folks!

<helpFrag2>
    This Help text contains two
    lines.
```

The extra indent of the include line is applied to the entire included fragment, so the above is equivalent to:

```
<helpFrag1>
    Note that:
        This Help text contains two
        lines.
    That's all folks.
```



The include facility is useful for grouping Help messages on specific topics into a single large *overview* message. For an example, see the help file for the leadlag demo (`$XMATH/demos/gui/leadlag.hlp`).

The Help fragment name can be followed by an optional title:

```
<helpFragName> Help Dialog title
  This is the Help text that will be displayed.
  The Help text can contain many lines.
```

Depending on the windowing system you use, the title should be displayed in the top border of the Help window.

The Help file is really a database of strings accessed by name. The Help file can be used to store strings or string arrays that a tool needs. Long options to the `uiPlo()` function, for example, can be placed in the Help file. This feature is shown in the binding1 demo (`$XMATH/demos/gui/binding1.hlp`).

## 9.5 Xmath GUI Functions

The Xmath GUI functions are categorized as follows:

- `uiToolCreate()`—creates a function for a tool.
- `uiWindow()`—creates a function for a top-level window of a tool.
- `uiPanel()`, `uiTab()`, and `uiTable()`—create container regions in a window or other container.
- `uiMenu()` and `uiMenuItem()`—create menu bars, pulldown menus, popup menus, and menu items.
- `uiButton()`, `uiComboBox()`, `uiList()`, `uiSeparator()`, `uiSlider()`, `uiText()`, `uiVarChoice()`, `uiVarEdit()`, `uiVarView()`, and `uiLabel()`—are controls for windows and containers for user interaction and displaying data.
- `uiPlotArea()`—creates a special control for displaying two-dimensional graphical plots.
- `uiDestroy()`, `uiExist()`, and `uiHandle()`—are PGUI object operations for checking existence, handle/name conversions and generic destruction.
- `uiHide()` and `uiShow()`—display and hide a PGUI object.

- **uiGetValue()** and **uiSetValue()**—get or set a PGUI object's resources.
- **uiFlush()**—forces the update of the objects displayed.
- **uiTimer()**—invokes an Xmath callback after a given amount of time has elapsed.
- **uiPlot()** and **uiPlotGet()**—are commands for generating two-dimensional plots in a **uiPlotArea** and getting user's input to the plot.
- **uiFileSelection()**, **uiMessage()**, and **uiPrompt()**—are predefined dialogs for selecting files, displaying messages, and prompting the user for input.
- **uiWindowDeiconify()** and **uiWindowIconify()**—deiconify and iconify a window.
- **uiWindowLower()** and **uiWindowRaise()**—lower and raise a window.

For more information on PGUI functions, see the MATRIX<sub>X</sub> online Help topic *MathScript Programming, Programmable GUI*.

## 9.6 Tutorial

In this section we discuss two tools: the pushbutton and the calculator examples. These tools perform trivial functions; the point is not their purpose but their operation.

### 9.6.1 Pushbutton

[Example 9-1](#) shows the **ex1.msc** file, located in the **\$XMATH/demos/gui** directory.

#### Example 9-1 Pushbutton Creation

```
command ex1 {fragname, widgetname, instance}

alias T "ex1"

if( exist(fragname) )
    goto *fragname;
else
    t1 = uiToolCreate("ex1");
    wn = uiWindow(t1, {name = "win", title = "Tutorial"});
    void = uiButton(wn, {
```

```

        text = "Press This Button",
        xmath = "ButtonPress"});
    main.count = 1;
    void = uiShow(wn);
    return;
endif

<ButtonPress>
    main.count = main.count + 1;
    display sprintf("Button Press count: %d", main.count);
    if(main.count >= 5)
        void=uiDestroy("ex1");
    endif;
    return;

endcommand

```

When the user types

```
ex1
```

The `ex1` tool window appears.

Let's investigate the steps that produced this window. When the user types `ex1`, the MSC `ex1.msc` is invoked with no arguments. Therefore, the `if` conditional

```
if( exist(fragname) )
```

fails, and the `else` clause is executed. The statement

```
t1 = uiToolCreate("ex1");
```

creates the new tool `ex1`. (If the tool already existed, this step would first destroy the tool and all its windows before creating a new tool.) The value returned and stored in `t1` is the tool's handle. All GUI creation routines return an object handle that is used when creating the tool's windows; the object handle can also be used to reference the tool for other GUI functions, such as `uiExist()` and `uiDestroy()`. In addition to the handle, some operations on a tool can also be invoked with the tool's name.

The next statement:

```
wn = uiWindow(t1,{name = "win", title = "Tutorial"});
```

actually creates a new window. The keyword arguments provide attribute information about a widget, the window in this case. You can provide a widget name—`win` in this case—so that you can reference the widget by its name instead of its handle.

```
void = uiButton(wn, {text = "Press This Button",  
                 xmath = "ButtonPress"});
```

creates a single button as a child of the window. Since no **type** keyword is specified, the default type “**button**” is created. The **text** keyword specifies the text to appear on the button face and the **xmath** keyword designates the callback fragment to execute when the button is pressed.

At this point, the window is still not visible. The call to **uiShow()** makes the window appear when desired. In this case, note that the call to **uiShow()** takes the handle returned from the **uiWindow()** call. The call to **uiShow()** could just as well appear as follows, which uses the name passed into the **uiWindow()** function.:

```
uiShow("ex1", "PushBWin")
```

While handles are slightly more efficient at times, they are less convenient. Therefore, both methods are provided.

Finally, the MSC initializes a global variable that is used to count button presses and then returns:

```
main.count = 1;  
return;
```

When the user clicks the button, the button checks to see if it has a value for its **xmath** attribute. In this case it is set to “**ButtonPress**” so the button will invoke the tool’s MSC as:

```
ex1 "ButtonPress", "Push"
```

We use the term *Xmath callback* to describe the calling of the tool’s MSC in this way. The first argument is the argument value set as the **xmath** attribute, the second argument is the pushbutton’s name, and the third argument is the instance number of the window, which will always be 1 (unless we create multiple instances of the same window). Therefore, when the user activates the button, it is equivalent to typing:

```
ex1 "ButtonPress", "Push"
```

When called with these arguments, the MSC executes the code

```
main.count = main.count + 1;  
display sprintf("Button press count: %d", main.count);  
if(main.count >= 5)  
    void = uiDestroy("ex1");  
endif;  
return;
```

This increments the count variable and displays a message in Xmath's log area. If the button has been pressed five times, you can see the following messages in the log area:

```
Button press count: 2
Button press count: 3
Button press count: 4
Button press count: 5
```

Then the tool is destroyed, which causes the window to disappear.

We use a global variable (**main.count**) so that its value is maintained between calls to the tool's MSC. Local variables in an MSC disappear when the MSC returns. You might notice that most GUI tools create their own partitions for storing all their global variables.

## 9.6.2 Calculator

[Example 9-2](#) shows the `ex2.msc` file, located in the `$XMATH/demos/gui` directory.

Example 9-2 **Calculator**

```
command ex2 {fragname, widgetname, instance}

alias T "ex2"

if( exist(fragname) )
    goto *fragname;
else
    t1=uiToolCreate("ex2")
    wn=uiWindow(t1,{name = "win", title="Tutorial"});
    tb=uiTable(wn,{columns = 2});
    void = uiLabel(tb,{text = "Operand 1"});
    void = uiSlider(tb,{varname = "op1",
                    xmath="NewOperand",
                    xmathdrag="NewOperand", flags = "hdm",
                    min = -1, max = 1})
    void =uiVarChoice(tb,{text="Operation",
                      xmath="NewOperation",
                      varname= "operation", flags = "H",
                      items=["plus", "minus", "times"],
                      values=[1,2,3]});
    void = uiLabel(tb,{text="Operand 2"})
    void = uiSlider(tb,{varname = "op2",
                    xmath="NewOperand",
                    xmathdrag="NewOperand", flags = "hdm",
                    min = -1, max = 1})
    void = uiSeparator(tb,{colspan = 2});
    void = uiLabel(tb,{text="Result"})
    void = uiSlider(tb,{varname = "result",
```

```
        xmath="NewOperand",
        xmathdrag="NewOperand", flags = "bhdms",
        readonly, min = -2, max = 2})
void = uiSeparator(tb,{colspan = 2});
void = uiButton(tb,{text = "Quit",
        col = 1, xmath = "DoQuit"});
main.op1=0;
main.op2=0;
main.operation=1;
ex2 "NewOperation";
void = uiShow(wm);
return;
endif

<NewOperation>
<NewOperand>
    if( main.operation == 1 )
        main.result = main.op1 + main.op2;
    elseif ( main.operation == 2 )
        main.result = main.op1 - main.op2;
    elseif ( main.operation == 3 )
        main.result = main.op1 * main.op2;
    endif
    return;

<DoQuit>
uiDestroy("ex2");
return;

endcommand
```

When the user types

`ex2`

a window showing a selectable operation between two operands appears. This window is created using the same steps as the previous tutorial. However, it uses a few more widgets, the first of which is the `uiTable()`. A table is used for laying out a number of other objects in regular rows and columns.

```
tb = uiTable(wm,{columns=2});
```

The keyword `columns` does two things:

- It sets the number of columns the table will have
- It specifies that the table will fill rows first

Widgets will be added across the table, one per column.

```
void = uiLabel(tb,{text = "Operand 1"});
```

creates a label containing text string, "Operand 1."

```
void = uiSlider(tb,{varname = "op1", xmath="NewOperand",
                 xmathdrag="NewOperand", flags = "hdm", min = -1, max = 1})
```

creates a slider bound to the Xmath variable **op1**. Each time the user sets the slider to a new value, and each time the slider is dragged, the variable **op1** is updated and the Xmath callback **NewOperand** is called. (Similarly, if the variable **op1** is set to a new value, the slider moves to the corresponding position.) The flag **hdm** specifies that the slider is horizontal. The Xmath variable is updated as the user drags the slider, and the minimum and maximum of -1 and 1 are enforced (even if the Xmath variable is set by the programmer to a value outside this interval).

```
void = uiVarChoice(tb,{text="Operation",xmath="NewOperation",
varname = "operation", flags="H", items=["plus", "minus",
"times"], values=[1,2,3]});
```

creates two entries in the table:

- A label containing the text “Operation”
- A box containing three radio buttons with the choices “**plus**”, “**minus**”, and “**times**” bound to the Xmath variable **operation**.

When the user selects one of these choices, the values 1, 2, and 3, respectively, are assigned to the variable **operation**. (Similarly, if the variable is set to one of these values, the corresponding radio button is set.)

Whenever the user selects a new toggle button, the Xmath callback **NewOperation** is called.

```
void = uiLabel(tb,{text="Operand 2"})
```

creates a label containing the text string, “Operand 2.”

```
void = uiSlider(tb,{varname = "op2", xmath="NewOperand",
xmathdrag = "NewOperand", flags = "hdm",
min = -1, max = 1})
```

creates a slider bound to the Xmath variable **op2**. This slider is otherwise the same as the first.

```
void = uiSeparator(tb,{colspan = 2});
```

draws a horizontal line in the row. The **colspan = 2** keyword expression causes it to occupy both columns in the table.

```
void = uiLabel(tb,{text="Result"})
```

creates a label containing the text “Result.”

```
void      = uiSlider(tb,{varname = "result", xmath="NewOperand",  
xmathdrag = "NewOperand", flags = "bhdms",  
          readonly, min = -2, max = 2})
```

creates a read-only slider bound to the Xmath variable **result**. The user cannot drag this slider because of the **readonly** keyword, but, whenever the Xmath variable **result** is set to a new value, the slider changes accordingly. The limits of this slider are -2 and 2.

When the user sets the Operand 1 slider to a new value, the variable **op1** is set to the new value, and the Xmath callback **NewOperand** is called. If the new value is, for example, 0.75, these operations are identical to the user typing the statements:

```
main.op1 = 0.75;  
ex2 "NewOperand", "dontcare",1;
```

(The actual widgetname argument will be different, but this isn't relevant to the discussion.)

This callback causes the following Xmath code to be executed:

```
<NewOperation>  
<NewOperand>  
  if( main.operation == 1 )  
    main.result = main.op1 + main.op2;  
  elseif ( main.operation == 2 )  
    main.result = main.op1 - main.op2;  
  elseif ( main.operation == 3 )  
    main.result = main.op1 * main.op2;  
  endif  
  return;
```

Based on the operation, the new result is computed. Since the variable **main.result** is bound to the bottom slider, the new value is automatically displayed when the variable is assigned. Similarly, when the user changes the operation, the same Xmath code is called to compute the new result.

For additional examples and descriptions, see the MATRIX<sub>x</sub> online Help.



## 9.7 Translating Version 5.X GUI Files to Version 6.X PGUI Files

This section describes the two utilities for translating Version 5.X GUI files to Version 6.X PGUI files, instructions on executing these scripts, details on using the translator, and some minor limitations.

### 9.7.1 Overview

Due to the significant changes in the Xmath Programmable GUI (PGUI) syntax in MATRIX<sub>X</sub> Version 6.X, the **to60pgui** utility has been created to facilitate the transition of old graphical tools to the new syntax (see [9.4.2 Anatomy of a GUI Tool](#), p.350). This utility consists of a pair of Perl scripts that convert the resource and MSC or MSF files from Version 5.X syntax to the Version 6.X syntax.



---

**NOTE:** MSC and MSF files are translated in place. Make sure you have a backup.

---

### 9.7.2 Execution

The easiest way to execute these Perl scripts is to copy them to a working directory. Ensure that Perl is in your path and copy the tool to be translated to that working directory. Then execute the main script with the following command:

```
perl to60pgui.pl Mytool mytool.msc
```

where *Mytool* is the X resource file used by *mytool.msc*. A resource file is a collection of resource settings that describe the appearance of the windows,



---

**NOTE:** The tool could also be an MSF in which case you provide the appropriate name and extension.

---

This script modifies the original *mytool.msc* file (make sure you have a backup) and creates a new file named *mytool\_build.msf* from the resource file. You can then compare the files and make any needed modifications. After that you should be able to run the MSC as before.

The Perl script **restopgui.pl** converts the X resource file, **Mytool**, to an MSF file in the new format. To run the resource translator **restopgui.pl** independently, use the following syntax:

```
perl restopgui.pl Mytool
```

where *Mytool* is the X resource file.

This script creates a new MSF file with the name *mytool\_build.msf*.

### 9.7.3 Details

The **MSC** translator scans through an **MSC** file and changes any Version 5.X **GuiFunction()** to a Version 6.X **uiFunction()** with the exception of **GuiShellCreate()** and **GuiDialogCreate()**. These functions have no counterpart in the Version 6.X PGUI because there is no need to create a shell separately from creating a window. However, in the old GUI these calls caused the window and its children to be created, so they are not just omitted from the new file. Instead they are changed to a call to an **MSF** file that is generated from the tool's X resource file. This call to the new **MSF** file has the same result as calling **GuiShellCreate()** or **GuiDialogCreate()** in that the window specified and its children are created.

For example, in our **Fourier** tool a **GuiShellCreate()** call such as:

```
GuiShellCreate("fourier", "MainWin", "Fourier Tool", "fourier tool");
```

becomes

```
Fourier_build("fourier", "MAINWIN", "Fourier Tool", "fourier tool");
```

Notice that the second argument is converted to uppercase because the second argument in the **fourier\_build.msf** file is used as the fragment label. The third and fourth arguments are optional as they are in **GuiShellCreate()**. See [Limitations](#).

The resource file conversion results in a new **MSF** file named *resourcefile\_build.msf*. For example, **Fourier** becomes **fourier\_build.msf**. The resource file conversion is the biggest task of the translator. It takes all of the X resource specifications and creates a hierarchy of Xmath calls to build the desired user interface. The commands in the resulting **MSF** file are grouped by window and indented to show the hierarchy. Fragment labels separate the code associated with each window so each window can be created as needed.

### 9.7.4 Limitations

The PGUI translators have some minor limitations because some features are not supported by PGUI or X resource settings need human intervention to be properly assigned. More specifically, X resources set in a global sense, such as a Motif class of widget, are not handled. Also, X resources set to affect all children of a certain widget are not handled. Examples of these are:

```
*MyTool*background: red
```

and

```
*MyTool*MainWin*background: redvisible
```

Specific X resources, such as the following, are not supported:

```
*MyTool*MyText.marginwidth: 4
```

In general, anything that can't automatically be translated is set as a comment using **uiSetValue**. The generic comments appear in the beginning of the **MSF** file and the more specific ones appear after the creation of the widget in question.

Within an **MSC**, calls to **GuiSetValue()** are not translated if they are of the form

```
GuiSetValue(T, "resource block");
```

where **resource block** is one or more X resource settings to be applied to the resource database. For both **GuiSetValue()** and **GuiGetValue()**, if the **resource block** is not known, then the command is not translated.

For additional help with any PGUI translation, contact **MATRIX<sub>X</sub>Customer Support** as described in Chapter 2 of the *MATRIX<sub>X</sub> Getting Started Guide*.



# A

## *X Windows and Motif*

This appendix introduces X Windows and the Motif window manager. If you are not using Motif, much of the Motif material will still be useful to you, as Xmath uses Motif

The material in this appendix gives general information that allows an inexperienced X Windows or Motif user to use Xmath and X at a novice level. However, it is not a replacement for X Windows documentation or documentation appropriate to your window manager.

### **A.1 X Window System**

What is X Windows? “The *X Window System*, commonly referred to as X, is a network-based graphics window system that was developed at MIT in 1984.”<sup>1</sup>

Xmath can be used with any window manager that runs as a layer over X Windows. X is largely transparent from Xmath; usually you only notice it while logging in or out.

Your X installation can be very complicated. If you are unfamiliar with X, you should consult the documentation or ask your system manager about your installation.

---

1. Quercia, Valerie & O’Reilly, Tim, *The Definitive Guides to the X Window System, Volume 3: X Window System User’s Guide* (O’Reilly & Associates, Inc., 1988, 1989), p 5.

You should know the answers to the following questions:

- Does your installation have an autostart procedure for X Windows, or must you start X Windows manually?
- If you start manually, what is the command to initialize X Windows at your site? (Usually it is `xinit`.)
- Are X Windows, window manager, and Xmath available locally, or must you access them across a network? If you are getting these applications from a remote source, what special instructions apply?

### A.1.1 Starting X

You should see your system manager to verify the correct way to start X at your installation. The normal procedure for starting X is as follows:

1. Log in at the system prompt.
2. Type `xinit`.

### A.1.2 X Terminology

This section defines some general terms this manual uses to direct your interaction with Xmath windows and menus. For comprehensive information, consult X Windows documentation or man pages.

#### Software Terms

Creating an environment for Xmath requires several types of software that are usually transparent to you. They are mentioned briefly here so that you have a point of reference if you see these terms in error messages, default files, etc.

The lowest-level software is the operating system—currently we assume UNIX. On top of that you must have X Windows and a window manager (such as Motif).

X Windows is a windowing system. A windowing system allows many processes to exist simultaneously, each running in a different window. X keeps track of input and output data for all windows.

A *window manager* is a *client* (an application) that describes how a window looks and allows you to manipulate windows (move, resize, stack, etc.). Xmath could be

run without a window manager, but there would be no borders, the windows couldn't be moved or resized, and so on.

The final element is the *server*. In this context, the server communicates X graphics instructions to the screen.

In general, X tells how a window is drawn, a window manager defines its appearance and activity, and the server implements these instructions on your graphics display.

### Mouse Terms

A list of the mouse conventions is on [p. 13](#). Some common mouse instructions are:

**click** — Press and quickly release a mouse button. If click is used without a button designation, MB1 is assumed. For example, “click the root window.”

**double-click** — Two clicks in quick succession. Double-click without a button designation assumes MB1.

**drag** — Hold down a mouse button while moving the mouse. This action is used for movement and resizing. Release the button when the desired result is obtained. Drag assumes MB1.

**press, push** — “Press” or “push” can be used interchangeably with “click MB1.” These terms are often used for buttons. For example, “press the Lock button.”

## A.2 Motif Window Manager

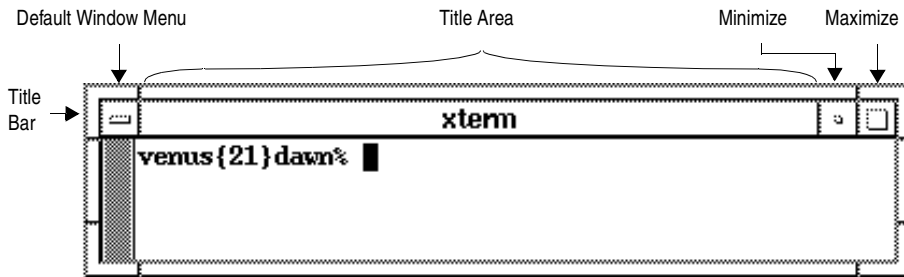
As discussed earlier, a window manager allows you to manipulate windows. In theory, you should be able to use any window manager that is compatible with X11 (see the *System Administrator's Guide* for your operating system; this documents UNIX window managers under which Xmath has been tested). The window manager creates frames and is also responsible for any window functions in X Windows and Xmath. The graphics in this manual use standard Motif frames.

## A.2.1 Motif Frame Components

The frame is basic to all windows. [Figure A-1](#) shows an xterm and labels each feature in the frame.

- The rectangular button shown in [Figure A-1](#) activates the Default Window Menu. This button has seven selections to change the window's appearance or position. These are discussed in [Default Window Menu](#).

Figure A-1 **Window Frame**



- The title area displays the name of the window. Click on this area (not including the buttons), then drag to move the window.
- The title bar is the title area and the buttons.
- The Minimize button turns the window into an icon (a small manageable graphic). To minimize (or *iconify*) a window, click MB1 on the Minimize button. The resulting icon has the same name as the window. To bring the window back, click MB1 on the icon.
- The Maximize button enlarges a window to fill the screen (it is not the opposite of Minimize). To maximize a window, click MB1 on the Maximize button. To return the window to its former size, click the Maximize button again.

### **Default Window Menu**

To view this menu, click MB1 over the rectangle. You can see the selection box move as you run the mouse up and down the menu. Click MB1 to select. If an item is not available, it will be grayed out. If you don't want to make a selection, move the mouse off the menu and release the mouse click in a neutral area (the root window, for example).

- Restore returns a minimized or maximized window to its original state.



- Move enables you to position a window anywhere you wish. The cursor will attach to the center of the window.
- Size resizes the window.
- Minimize reduces a window to an icon.
- Maximize enlarges a window so that it fills the entire screen.
- Lower puts the current window in back of any window(s) sharing the same space.
- Close terminates the client. In Xmath it is preferable to use File→Quit for the Xmath Commands window or File→Close Window from other Xmath windows, rather than closing from the Default Window Menu.

For instructions on making menu selections using keystrokes, see [A.2.4 Using Menus Without the Mouse](#), p.371.

## Frame Buttons

### Minimize

Reduces the window to an icon; double-click or select Restore from the Default Window Menu to return it to original size. Has the same effect as Minimize on the Default Window Menu.








### Maximize

Enlarges the window to fill the entire screen; click the Maximize button again, or select Restore from the Default Window Menu to restore it to original size. Has the same effect as Maximize on the Default Window Menu.

## Window Operations

These operations can be accomplished without using the Default Window Menu. Move the pointer over a window's frame. Notice the changes in the pointer. The pointer symbols are shown in [Figure A-2](#).

Figure A-2 **Pointer Symbols**

-  A large X is visible when the pointer is over the root window.
-  The "I-beam" appears when the pointer is over an area that accepts text.
-  For resize, the pointer changes to a symbol appropriate to the selection.
-  If an edge is selected, the chosen edge is displayed.
-  If a corner is selected, the chosen corner is displayed.
-  For movement, changes to a fleur.
-  For menus, points opposite direction from selection arrow.

Resize and move require you to *drag*:

**resize** — To change the size in one direction, place the cursor over an edge, then drag to the desired dimension. To simultaneously change two dimensions, place the cursor over a corner, then drag to new size.

**move** — To move a window, place the cursor in the title area, then drag to the desired location.

A window is raised (brought to the front) whenever you click on its Title area. The only way to lower a window (send it to the back) is from the Default Window menu.

## A.2.2 **Mouse Focus and the Pointer**

When you move the mouse, the pointer moves on the display. Pointer position governs *input focus* (that is, where keyboard input appears). That is, the pointer determines the active window.

In Motif (and most other window managers) there are two ways to give input focus. For simplicity, let's describe them as *point* and *point-and-click*. The *point* method means that input is directed to the window under the pointer. For *point-and-click*, you must position the cursor over the target window, and then click before you have focus.

There are several pointer symbols to become familiar with. The pointer symbol changes according to the context, as described in [Figure A-2](#).

Consult your window manager documentation or see your system manager for more information.

### A.2.3 Copying and Pasting with Motif

You may find it easier to cut and paste available text (such as pathnames, etc.) instead of retyping. Your machine's selecting, copying, and pasting methods are valid for both X Windows and Xmath.

The standard Motif method is:

1. Point to the desired text and drag until everything you want appears in reverse video (is highlighted). Avoid highlighting extra characters.
2. Point to the destination and click MB2.

As an alternative to dragging, use one of the following three mouse-click selection sequences. To select a word, point anywhere within the desired word and double-click. To select a line, point anywhere on the line and click three times. To select all text in an Xmath window, point and click four times.

These click sequences are often used in the Xmath Commands window to copy text from the log area and paste it into the command area.

### A.2.4 Using Menus Without the Mouse

The Motif window manager makes it possible to use Xmath menus via the keyboard. To make a menu selection you normally place the pointer over the menu, drag down MB1, and release when the desired selection is highlighted. If you look at the Commands window menu bar, you will see that the first character of each pull-down is underlined.

1. To invoke a menu, make sure the proper window has focus.

Press the Meta key (see [Table 1-5](#), p.15 for equivalents across platforms), followed by the character underlined in the menu bar. For example, press Meta-e to invoke the Edit menu. Note that although the underlined letter is capitalized, only lowercase letters will work (this is a Motif limitation). Use the up and down arrow keys to travel up and down the available options. Hit Return to invoke an option.

2. Once you are in the menu bar, use the left and right arrow keys to move along the menu bar.
3. If a submenu is available, an arrow points to the right at the end of the entry. Cursor up or down to the submenu and press the right arrow key to pop up the submenu. To go to a top-level menu, keep pressing the left or right arrow keys.
4. Press Esc or F10 to dismiss the latest menu.

The above option works with any Xmath window.

### **A.2.5 Using a Motif File Selection Dialog**

Figure A-3 shows a typical dialog that uses the Motif file selection dialog. Most dialogs have the same fields, but some actions may not require all fields. The instructions below show UNIX file paths.

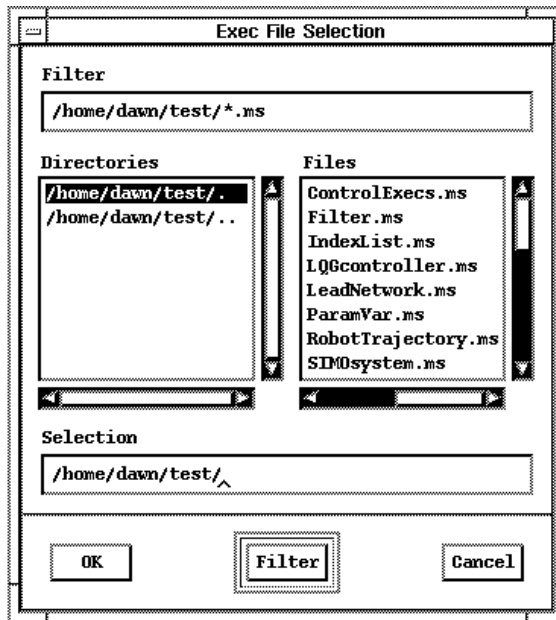
1. The first step is usually to make a selection from the Directories (the column on the left): either click on a selection and press Filter, or double-click on the selection. You may need to use the scroll bars to bring the name in view.

Alternatively, type in the Filter field to alter the search parameters. However, you must have a file specification, even if it is only the wildcard \*. To start the search, either press Return or push the Filter button at the bottom of the dialog.

Files meeting the filter criteria are displayed in the Files field. To search for the same parameters in another place, double-click on a new entry in the Directories field. Note that both columns can be scrolled to view long names.

2. To make a selection from the Files field, either click on an entry and press Return, or click on an entry and push OK, or double-click on the file. Note that the selected file will be displayed in the Selection field.

Figure A-3 Exec File Selection Dialog



A

## A.3 Changing Resource Parameters

If you want to change Xmath's appearance (color, etc.), you must be familiar with how X Windows works. The way Xmath looks is defined in a resource file, which contains hundreds of settings. Xmath looks for a resource file that isolates resources that directly affect Xmath (see [Table A-1](#)); this file often resides in your home directory and is read when X windows is started. (If you are on a network, you may not have a copy in your home directory.)

Table A-1 Resource File Default Location

	ISI Top Level	Resource File
UNIX	\$SIHOME	\$XMATH/etc/Xmath

The `$XMATH/etc/Xmath` file contains default settings for Xmath that you may want to change or override. Do not modify the `$XMATH/etc/Xmath` file. Instead, use a local version in your home directory to specify any changes. Your version only needs to contain settings that *differ* from the defaults. [Example 9-3](#) shows a sample Xmath file.

Example 9-3 **Sample Xmath File**

```
!A local Xmath file must reside in your home directory. This file
!changes window sizes and has them appear staggered on the right side
!of the screen:
!-----
!  COMMAND WINDOW
*main.geometry:      535x695-5+85
*main*log.rows:      32
*main*command.rows:  7

!  GRAPHICS WINDOW
*graphicsW.geometry: 545x450-13+93

!  HELP WINDOW
*helpW.geometry:     -21+101
*helpW*text.rows:    40

!  DEBUGGER WINDOW
*debuggerW.geometry: -29+109
*debuggerW*text.rows: 30
```

[Example 9-3](#) deals with window dimension and placement only, but other common changes might be changes to the key bindings or window colors. To make your own Xmath file, follow this procedure:

1. Copy Xmath from its default location (see [Table A-1](#)) to your home directory.
2. Use a text editor to alter the local Xmath file.

If you are changing the key bindings to a style other than emacs, the settings will be commented out with exclamation marks (!); make sure these are removed in your personal file.

3. After making changes, *delete all unchanged portions*.

Your changes will be implemented the next time you invoke Xmath. When you start Xmath, the Xmath file in the default installation location is read first, followed by the Xmath file in your home directory. (This is why duplications should be deleted; startup will be slower if they exist.)

### A.3.1 Remapping Your Keyboard

Because the keyboards in use with X workstations vary so much between platforms, vendors, and countries, you may at some point wish to change the key bindings Xmath uses.

#### Changing Key Bindings in X

A machine-specific key code is associated with each key on your keyboard. Within the X Window system, you can use *keysyms* (key symbols) to make this machine-specific code produce whatever key-binding code you need. To get the complete list of key codes for all the keys on your keyboard, type:

```
xmodmap -pk
```

`xmodmap` gives output similar to that shown in [Example 9-4](#).

UNIX: If `xmodmap` is not in your path, see your system administrator.

#### Example 9-4 Sample KeySym Output

There are 2 KeySyms per KeyCode; KeyCodes range from 8 to 132.

KeyCode	KeySym	(Keysym) ...
Value	Value	(Name) ...
:	:	:
61	0x0051	(Q)
62	0x0057	(W)
63	0x0045	(E)
64	0x0052	(R)
65	0x0054	(T)
66	0x0059	(Y)
67	0x0055	(U)
68	0x0049	(I)
69	0x004f	(O)
70	0x0050	(P)
71	0x005b (bracketleft)	0x007b (braceleft)
72	0x005d (bracketright)	0x007d (braceright)
:	:	:

The **KeyCode** value in the first column is machine-specific and cannot be changed. However, you can change the **Keysym** value globally (so that the key's function is changed in all applications) or locally.

For example, some keyboards do not include the `[]{}` characters, which are used widely within Xmath. On the SunOS and Solaris platforms, if you want to bind

the `[` characters to the F1 key and the `]` characters to the F2 key, go to the command line and type:

```
xmodmap -e "keycode 12 = bracketleft braceleft"  
xmodmap -e "keycode 13 = bracketright braceright"
```

This means F1 will type “[” and Shift-F1 will type “{”, etc. Note that you should modmap to keys you do not use, rather than to alphanumeric or punctuation keys. Also, the key code values may be different on different platforms. For example, on the HP platform, F1 is key code 16 and F2 is key code 24.

These `xmodmap` settings will be lost when you log out, so if you want them to be a standard part of your environment, save the settings to a file and call this file up as part of your `.login` file.

### Changing an Xmath Key Binding

To get the list of all key bindings local to Xmath, look at the default `Xmath` file. There you can see that the emacs style keyboard translation settings are the default.

Note that there are two translations: `XmTextField.translations` and `*XmText.translations`. Text field translations are active in dialog boxes (where all input is appended on a single line) such as the Load dialog. Text translations, (the longer list) are active in multiline environments such as the commands window command area and the debugger edit area. For this reason a key may have different assignments. For example, look at the assignments for `Key<Home>`. In the Text field translation it is set to `beginning-of-line`. In the text translation it is set to `beginning-of-file`.

1. Create a file called `Xmath` in your home directory (this can be the same file discussed in [Table A-1](#), p.373).
2. Go to the operating system and use `xmodmap` ([p.375](#)) to identify the `KeySym` name for the key you are rebinding.
3. Locate the key binding to assign to the chosen keycode.
4. To change a binding, put a line of the following form in your personal Xmath file:

```
*defaultBinding:yourKeyBinding:<Key>yourKeySymName
```

5. Save your Xmath file and restart Xmath to see the change.



### A.3.2 Sizing and Placing Windows

X Windows uses a geometry option to size and place windows. The standard geometry string is:

$$width \times height \pm xoffset \pm yoffset.$$

As shown in [Example 9-3](#), there are full geometry strings for the Commands window and the Graphics window (dimensions are specified in pixels). You can size and place the Graphics window in one step, because it is a *single* window.

It takes two steps, however, to do the same for the Commands window, (which handles text in three areas). For it, you specify the number of character columns (the default is 80), then specify the number of rows of text you want to see in each area. Xmath builds a window that reflects your changes. Consequently the *width*  $\times$  *height* dimensions are unknown. This means the dimensions shown in the example may not necessarily work on your machine, because the true dimension of a window is affected by factors Xmath does not control, such as border width and shadowing settings specified for your window manager. If the sizes do not agree with what X Windows knows, it may use the defaults instead; or you may see that the size was changed as you desired, but the window is not in the place you want it.

To find out the true window size, restart Xmath with the new settings. When the window appears, go to an xterm and type **xwininfo**.

When you get a crosshairs cursor, click on the window you need to control, and make note of the width and height dimensions.

Note that in the sample on [p.374](#) only the placement dimensions need to be supplied for windows other than the Commands window; your window manager, may, however, require a full geometry. See your X Window documentation for a full description of this process. Edit your **Xmath** file so that it contains the correct dimensions for the windows you want to move. Save your file and restart Xmath.



# B

## *Xmath HP-GL Driver*

Xmath supports Hewlett-Packard Graphics Language (HP-GL) hardcopy devices. You can choose to either print to a file (that is, save the output in a file), or print to a printer. To write an HP-GL file, go to the graphics window and select File→Save (to print to a file) or File→Print (to print to an output device), or use the **hpgl** keyword in the **HARDCOPY** command.



---

**NOTE:** The HPGL driver does not support hidden surfaces. For 3-dimensional plots, you must remove the surfaces by suppressing the **face** keyword (**!face** or **face=0**).

---

### ***B.1 Supported Devices***

All devices supporting the HP-GL language (for example, HP plotters models HP7550A, 7470, 7475, 7580, 7585, and 7586) should be able to plot the **.hp** file created by Xmath. The following plotters have been tested: HP7440A, HP7575, and the ENCAD SP2800 plotter.

## B.2 Setting the Aspect Ratio

Xmath assumes a paper size of 8.5 by 11 inches on the HP7440A, corresponding to a plotting area of 25 by 18.1 cm. The aspect ratio of the hardcopy output might change if you use a different plotter or paper size. You can use the Print Scale options in the Print dialog to change the aspect ratio of the plot.

## B.3 Color Pen Specifications

Xmath expects the following color pens to be in the specified stalls in the pen carousel, as indicated in [Table B-1](#).

Table B-1 **Color Pen Specifications**

Pen Number	Expected Color	Pen Number	Expected Color
1	black	5	red
2	blue	6	magenta
3	green	7	yellow
4	cyan	8	digitizing sight

Xmath attempts to map plot colors to these eight colors.

# C

## *Xmath for MATLAB Users*

Xmath is a numerical problem-solving application similar to MATLAB™ and other numerical software. While many of the constructs for storing and manipulating data are similar to MATLAB, you will find that Xmath extends both the amount of information stored with a given object and the number of actions a command or function can take, depending on the type of data passed. The Xmath work environment retains the configurable nature you are accustomed to in MATLAB, but syntax changes have been made to make Xmath more consistent, intuitive, and flexible.

This appendix describes changed features, explains the motivation for changes, and in general helps smooth your transition from MATLAB to Xmath. [C.1 Syntactic Differences](#), p.382 describes basic changes in the punctuation and syntax used in the software. [C.2 Object Differences](#), p.386 describes objects that were represented as vectors or matrices in MATLAB but are represented as full-fledged data types in Xmath. [C.3 Interpretation Differences](#), p.388 describes differences that affect environment settings, data representations, and programming issues. [C.4 Comparison of Frequently Used Commands](#), p.398 provides a comparison between Xmath and MATLAB of frequently used commands. Moreover, tables illustrating equivalent expressions in MATLAB and Xmath appear throughout this appendix.

## C.1 Syntactic Differences

This section details Xmath features that have the same functionality as MATLAB features, but are invoked in a slightly different way.

### C.1.1 Continuation

If a MATLAB function cannot fit onto a single line, it can be split over multiple lines with two adjacent periods to signal a continuation.

In Xmath, a continuation is seldom needed; if an unmatched parenthesis or brace exists, or the line ends in a comma, Xmath assumes that the expression will continue. Aside from this, the Xmath command area can take a line of nearly infinite length ( $2^{31}-1$ ). Most users break their instructions for readability rather than necessity. Xmath uses an ellipsis (...) when an explicit continuation is required. Because strings must be complete on a line, they are the most frequent candidates for continuation. [Table C-1](#) shows examples of command continuation in MATLAB and Xmath.

Table C-1 **Command Continuation Examples**

MATLAB	Xmath
<pre>plot(1:10,.. 'b') title('An Easy Plot')</pre>	<pre>plot (1:10, {!grid, title="An Easy Plot"}) plot(x, {title="A very"+... " long string"})</pre>

### C.1.2 Output Display

In MATLAB, variables are by default displayed to the MATLAB Command window as soon as they are created; output is suppressed if a semicolon is placed at the end of the expression that generated the variable.

Xmath's default display mode behaves similarly. This mode can be explicitly set with the command **set display on**.

Alternatively, you can specify **set display off**. In display-off mode, any variable created with an expression containing an equality sign is *not* displayed to the Xmath Commands window log area. For example,

```
A=sin(pi)
```

does not generate any output in the commands window log area if the display is Off. If you want to display a value as soon as it is created, place a question mark (?) at the end of the expression. If you want to see the value of a previously-created variable, type its name; because the name is not an expression (does not contain an equality sign), its value is displayed. [Table C-2](#) shows examples of output display in MATLAB and Xmath.

Table C-2 **Output Display Examples**

MATLAB	Xmath (set display on)	Xmath (set display off)	output?
<code>A = SIN(PI);</code>	<code>A = sin(pi);</code>	<code>A = sin(pi)</code>	No
<code>A = SIN(PI)</code>	<code>A = sin(pi)</code>	<code>A = sin(pi)?</code>	Yes
<code>A</code>	<code>A</code>	<code>A</code>	Yes

### C.1.3 Matrix Punctuation

Matrices are created and entered in the same basic manner, with one important difference: all matrix elements in Xmath must be separated by commas, as shown in [Table C-3](#), whereas commas are optional in MATLAB.

Table C-3 **Matrix Punctuation Examples**

MATLAB	Xmath
<code>A = [1 -1 2;-4 3 12]</code> or <code>A = [1,-1,2;-4,3,12]</code>	<code>A = [1,-1,2;-4,3,12]</code>

When you specify matrix elements separated only by spaces, it is unclear whether the element specification [1 -1] represents two separate numbers or the single number 0 (the result of the arithmetic operation 1 - 1 = 0). Because matrix elements in Xmath must be explicitly delineated by commas, the value of a given element is always clear both to you and to the Xmath interpreter. You still use semicolons and new lines to mark the end of a matrix row.

### C.1.4 String Punctuation

To avoid confusion with the transpose operator, Xmath uses double quotation marks rather than the single quotation marks used in MATLAB. [Table C-4](#) illustrates.

Table C-4 String Punctuation Examples

MATLAB	Xmath
<code>str = 'This is a string'</code>	<code>str = "This is a string"</code>

The treatment of string variables is discussed in more detail later in this appendix.

### C.1.5 Logical Not

In MATLAB the operator denoting a logical **not** is a tilde (~); in Xmath it is an exclamation point (!). To express an inequality relation in Xmath, use <> (the greater-than and less-than signs); in MATLAB ~= (tilde-equality sign) denotes inequality. [Table C-5](#) shows the logical **not** operators for MATLAB and Xmath.

Table C-5 Logical Not Operators

MATLAB	Xmath
<code>if ~(A &gt; 0)   disp('A is negative') end</code>	<code>if !(A &gt; 0)   display "A is negative" endif</code>
<code>A ~= B</code>	<code>A &lt;&gt; B</code>

### C.1.6 Comments

The single-line comment symbol has been changed from % in MATLAB to # in Xmath. Unlike MATLAB, Xmath supports block comments, which are delineated with #{ at the beginning and }# at the end. Instead of beginning each line of a section of comments with #, you can place the #{ marker at the beginning of the first comment line and the }# marker at the end of the last comment line. [Table C-6](#) shows comment examples for MATLAB and Xmath.

Table C-6 Comment Examples

MATLAB	Xmath
<code>% This is a comment.</code>	<code># This is a comment.</code>
<code>% Should you feel the need % to describe what you have % written at greater length % you have to comment each % line individually in MATLAB.</code>	<code>{# This is a block comment. Anything inside the markers is interpreted as a comment. Most programming languages support this construct.}#</code>



### C.1.7 Function Names

Xmath tends to preserve the full names of functions performing a given operation. Where the Hessenberg-decomposition and random-value generation functions in MATLAB are **HESS()** and **RAND()**, respectively, the Xmath equivalents are **hessenberg()** and **random()**. These names are less cryptic and more descriptive to the new user.

For your convenience, however, Xmath also recognizes a function called using only the first four letters of its name, or as many more as needed to specify the function uniquely. For example, you can call **random()** as **rand()**, but would need to use **polyn()** to distinguish **polynomial()** from **polyfit()**.

In addition, you can take advantage of Xmath's **alias** command to alias lengthy function names or command statements to shorter ones of your choosing. For example:

```
alias sdon set display on
```

(See [C.3.13 Useful Aliases](#), p.397 for a listing of aliases that you might want to have predefined in a startup file.)

### C.1.8 RAND, ONES, ZEROS, and EYE

Another syntax change concerns the matrix-building functions **RAND()**, **ONES()**, **ZEROS()**, and **EYE()**. These functions operate in one of two ways depending on the type of input provided. They either create a random, ones, or identity matrix of the same size as the input, or a matrix of the dimensions specified in the input. This causes some ambiguity when the function argument is a scalar—should the output matrix also be a scalar, or should it be a square matrix whose dimensions have the same value as the scalar?

When these functions are used with one argument in Xmath, the output matrix always has the same dimensions as the input object. [Table C-7](#) illustrates.

Table C-7 **Examples With RAND**

MATLAB	Xmath
<b>RAND(1)</b>	<b>random(4) # (a 1x1 matrix)</b>
<b>RAND(4)</b>	<b>random(4,4) # (a 4x4 matrix)</b>
<b>RAND(2,3)</b>	<b>random(2,3) # (a 2x3 matrix)</b>

### C.1.9 IF, FOR, and WHILE

In executable files, MathScript functions, commands, the IF...END, FOR...END, and WHILE...END loops, and conditional structures have been modified slightly. Conditional statements starting with **If** in Xmath should be closed with **endIf**, rather than END. (Because functions and commands are case-insensitive, any capitalization scheme will work with these constructs.) Similarly, Xmath **For** and **While** loops terminate with **endFor** and **endWhile**, making it much easier for a user reading MathScript to decipher which ending statements close which loops. [Table C-8](#) shows examples of conditional statements in MATLAB and Xmath.

Table C-8 **Conditional Statement Examples**

MATLAB	Xmath
FOR variable=vector DO, commands; END	For variable=vector commands endFor
WHILE expression DO, commands; END	While expression commands endWhile
IF relation1 THEN, commands; ELSEIF relation2 THEN commands ELSE, commands; END	If relation1 commands elseif relation2 commands else commands endIf

### C.1.10 Pure Imaginary Number

The variable representation of the pure imaginary number (the square root of -1) is **jay** in Xmath, following engineering standards, as opposed to **i** in MATLAB.

## C.2 Object Differences

Several objects that were represented as vectors or matrices in MATLAB are represented as full-fledged data types in Xmath.

### C.2.1 Strings

Xmath's real character strings can be manipulated more easily than strings implemented in MATLAB (which are essentially vectors of ASCII values). For example, in Xmath you can append one string to another one of any length using the + operator. You can also create matrices where elements are all strings of differing sizes. (This is, in fact, a handy way to create a table where text entries are neatly aligned.)

Table C-9 **String Examples**

MATLAB	Xmath
<code>STR='A string'</code>	<code>str = "A string"</code>

### C.2.2 Polynomials

In Xmath, polynomial coefficients and roots are stored as one of two types of polynomial objects instead of vectors. When you create a polynomial, both the roots and the coefficients of that polynomial are stored internally for use in future computations for greatest efficiency and accuracy. [Table C-10](#) gives examples of polynomial creation in MATLAB and Xmath.

Table C-10 **Polynomial Examples**

MATLAB	Xmath
<code>% Creating a polynomial by % listing its coefficients: CP = [1 4 4]</code>	<code># Creating a polynomial by # listing its coefficients: cp = makepoly([1,4,4])</code>
<code>% Creating a polynomial by % listing its roots: RP = POLY([-2 -2])</code>	<code># Creating a polynomial by # listing its roots: rp = polynomial([-2,-2])</code>

### C.2.3 Dynamic Systems

Xmath stores dynamic systems as single objects containing all state-space or numerator/denominator information, as well as any sampling rate information. In MATLAB you need to keep track of different commands for building different types of systems. In Xmath, everything is grouped in the system object. A brief comparison of these representations is shown in [Table C-11](#).

Table C-11 Dynamic Systems Examples

MATLAB	Xmath
<pre>% For statespace systems sys = ss(A, B, C, D);  % For transfer function sys = tf(num, dem);</pre>	<pre># Creating a system from # matrices A, B, C, and D: sys = system(A,B,C,D)  # Same command for transfer fn sys = system(num, dem);</pre>

The `system()`, `makepoly()`, and `polynomial()` functions are far more flexible, and can encompass more information, than their MATLAB equivalents. See the MATRIX<sub>X</sub> online Help for a complete reference on these functions.

## C.3 Interpretation Differences

The differences described in this section are by-products of Xmath's more complete user environment. In general, these are conceptual changes that involve learning new terms rather than word-for-word syntax changes.

### C.3.1 Environment Commands

Xmath has a highly customizable user environment. Many environment settings in Xmath replace functionalities that existed as individual commands in MATLAB. These include creating session and command diaries, changing display format in the commands window, setting random number distribution and generator seeds and more. In Xmath, settings are treated as parameters that are changed with the **SET** command; each parameter is a keyword. Help for the **SET** command describes many new capabilities not included in MATLAB. Read the MATRIX<sub>X</sub> online Help to understand the full range of settings available. A setting remains in its current mode until it is explicitly changed. To see the status of a particular environmental setting, you can use **SHOW**. For example:

```
show echo      #(default is off)
set echo on
```

The settings discussed below map closely to MATLAB capabilities you are probably familiar with.

### Creating Diaries

Once a diary file has been created, it collects input from your Xmath or MATLAB session until it is closed. The presence or absence of a diary is thus a mode of operation. The **DIARY()** function used to start a diary session in MATLAB has been replaced with the **set sessiondiary** and **set commanddiary** syntax shown in [Table C-12](#).

Table C-12 **Creating Diaries**

MATLAB	Xmath
<pre>% Creating diaries DIARY 'stamen' DIARY off % MATLAB can't % keep a % command diary</pre>	<pre># Creating diaries set sessionDiary="sdname" remove sessionDiary  set commandDiary = "cdname" remove commandDiary</pre>

### Random Seeds and Distribution

The MATLAB **RAND()** function is ambiguous because it returns an output like a standard function when called with purely numeric input, but also takes string input and uses it to set the distribution mode and initial seed. In these cases there is no logical function output. Xmath's handling of these functionalities through the **SET** command is more consistent, as shown in [Table C-13](#). The Xmath **random()** function always returns purely numeric output.

Table C-13 **Random Seeds and Distribution Examples**

MATLAB	Xmath
<pre>% Setting random % number seed randn('SEED',100)</pre>	<pre># Setting random # number seed set seed 100</pre>
<pre>% Setting distribution randn('NORMAL')</pre>	<pre># Setting distribution set distribution normal</pre>

The default seed is **0** and the default distribution is **uniform**.

### Number Formatting

The Xmath equivalent to the MATLAB commands **SHORT**, **SHORT E**, **LONG**, **LONG E**, **HEX**, **BANK**, **COMPACT**, **LOOSE** and **RAT** is **set format formatname**. An

advantage of the Xmath syntax is that it allows a wider range of formatting options without the need to add a new command each time! [Table C-14](#) gives an example.

Table C-14 **Number Formatting Examples**

MATLAB	Xmath
<pre>% Set number format long % or format long</pre>	<pre># Set number format set format long</pre>

The Xmath format names are: **compact** (the default format), **engineering**, **fixed**, **long**, **longe**, **scientific**, **short**, and **shorte**. Note that the format can also be set interactively via the Options→Format menu option in the Commands window.

Note that **fixed** is slightly different in that you must set *two* parameters; you must specify the format name **fixed**, and the precision:

```
set precision 4;set format fixed
```

The precision is the number of characters allowed. Remember that both settings remain the same until you reset them; if you use the above settings and then set another format, the **precision** will still be 4 the next time you **SET format** to **fixed**.

### C.3.2 User-Defined Functions and Commands

While MATLAB allows you to define optional arguments to a user-defined function or command, delineating them with single quotation marks, Xmath offers related but much richer ways to extend the user input to a MathScript function or command.

In Xmath, optional arguments and keywords are specified following the required argument list when the function is declared.

- Keywords must be delineated with curly braces {}. They can take any values and be specified in any order, but the name of the keyword must always be used so that the Xmath interpreter knows which keyword is being sent. If you are writing your own function or command using keywords, you should provide default values for any keywords where values are not user-supplied. (See [3.5.1 Command and Function Calling Syntax](#), p.85 starting on [p.85](#) for more on function syntax.)
- Optional arguments can be specified by their value or variable name alone, and are assigned to the optional variables in the order that they are listed.

When a function is called with optional arguments, they are listed directly after the required arguments and are not enclosed in curly braces.

In both MATLAB and Xmath, you can define functions and commands that override existing functions and commands, including intrinsic ones. In Xmath, you can place the function or command in the search path or use the **DEFINE** command to determine which one you want to use (see [6.1.7 Using User-Defined MSFs and MSCs](#), p.230); in MATLAB, a user-defined function has priority over a function supplied by MATLAB.

### C.3.3 `plot()`

In Xmath, `plot()` is a function that returns an output variable (a graphics object, as discussed in [4.2 Using the `plot\(\)` Function](#), p.108). This variable can be subsequently replotted to regenerate a plot, kept to form a background or template for subsequent plots, and augmented via interactive changes to the graphics.

MATLAB option strings are replaced in Xmath by `plot()` keywords. Referring to the online Help will give you a good idea of the scope of `plot()` parameters that you can set in Xmath, but [Table C-15](#) illustrates briefly.

Table C-15 **Plot Examples**

MATLAB	Xmath
<code>PLOT(1:10, 'b')</code>	<code>plot(1:10,{line_color=4}) # or plot(1:10,{line_color="Blue"})</code>

### C.3.4 Transpose Operators

The transpose operator is interpreted differently in Xmath. MATLAB offers only one transpose operator, the apostrophe (`'`). When used with a complex matrix, the transpose operator performs a Hermitian, or complex-conjugate transpose.

Xmath offers two transpose operators:

- The Xmath apostrophe operator (`'`) performs a regular transpose, leaving complex values untouched.
- The Xmath complex-conjugate transpose operator is the asterisk-apostrophe (`*'`).

For purely real matrices these two transpose operators perform the same function.

Table C-16 illustrates Xmath and MATLAB equivalents.

Table C-16 **Transpose Operator Examples**

MATLAB	Xmath
$A = \begin{bmatrix} 1+i & 1+2*i \\ 3-6*i & 2+9*i \end{bmatrix}$	$A = [1+jay, 1+2*jay; 3-6*jay, 2+9*jay]$
A'	A*
CONJ(A')	A'

### C.3.5 Convolve

The CONV() function, which performs polynomial and vector convolution in MATLAB, has been replaced by the **convolve()** function and the \* operator in Xmath. **convolve()** is equivalent to CONV() when used on two vectors or two polynomial objects; however, the \* operator performs exactly the same operation on polynomials as **convolve()** does and is easier to use.

### C.3.6 Series and Parallel

The MATLAB functions SERIES() and PARALLEL() have been replaced by the Xmath operators \* and + respectively, when these operators are used with dynamic systems. You will find the online Help and [Using Operators with Dynamic Systems](#) on p.210 useful for a quick but thorough overview of the extended role operators play in Xmath.

### C.3.7 Simulation

The MATLAB continuous- and discrete-time simulation primitives LSIM and DLSIM have been replaced with the **system\*PDM** construct. (The parameter-dependent matrix [PDM] is a highly useful data type unique to Xmath. It allows you to store multiple sets of matrix information (input values) that are dependent on a parameter [time]. For a complete explanation of PDMs, see [5.4 Parameter-Dependent Matrix \(PDM\)](#), p.187.) This construction finds the system response to the input values contained at each point in the PDM. The syntax inherits from terminology frequently used in the linear systems field:  $Y = H*U$ , where U represents system input, H represents the mathematical model of the system's dynamics, and Y is the output of the system. This is a brief description; for more information, see [5.5.5 Time Response](#), p.214.



Note that where MATLAB generally offers two separate functions for discrete- and continuous-time system representations, Xmath only offers one. This is because sampling-rate information (which is by default zero, thus describing a continuous system) is stored with the system object itself. For example, the Xmath function `bode()`, which encompasses all the functionality of the MATLAB functions `BODE()` and `DBODE()`, automatically checks whether your system is continuous or discrete and then performs the appropriate operations in either case. You can write similarly flexible MathScript functions.

### C.3.8 Eval (Executable Strings)

Xmath offers a facility (similar to the MATLAB `EVAL()` function) that allows you to create strings containing valid Xmath commands and then execute the contents of the strings. It can be used for creating macros or customizing functions. You can create strings directly or append them using the `+` operator, then use Xmath's `EXECUTE` command. The only constraint is that the string must form a complete Xmath statement by itself and be terminated by a semicolon or question mark to indicate its end. [Table C-17](#) illustrates.

Table C-17 Executable String Examples

MATLAB	Xmath
<pre>x = pi s = 'y = sin(x);' eval(s)</pre>	<pre>x = pi; s = "y = sin(x)?" execute s</pre>

As mentioned in [C.1.4 String Punctuation](#), p.383, MATLAB does not allow string concatenation. In Xmath, the `+` operator is overloaded to perform string concatenation. In addition, numbers can be converted to strings using the `string()` function.

### C.3.9 Executable Files

Executable files (often referred to as script files in Xmath) function similar to `script.m` files in MATLAB. A small change is that the names of these files must terminate with the extension `.ms` in Xmath. The syntax to execute files is slightly different as well, as shown in [Table C-18](#), for an executable file called `testexec.ms` in Xmath and `testexec.m` in MATLAB.

Table C-18 Executable Filename Examples

MATLAB	Xmath
TESTEXEC	execute file="testexec"

### C.3.10 Finding Files

Xmath has the ability to use files that are not in the working directory. It does this in a more flexible manner than that employed in MATLAB. In MATLAB, the **MATLABPATH** environment symbol defining the accessible directories is generally set up before you start your MATLAB session. The **MATLABPATH** could be changed during a session using the **MATLABPATH** command, but it had to be completely changed at once. In Xmath you can alter the directory search path at any time during your Xmath session, and you can add or remove paths separately, without having to redefine the entire path each time a modification is desired.

**SET path** is used to specify a list of directories that Xmath will automatically search to find MathScript functions and commands (MSFs and MSCs). You can use the corresponding **REMOVE path** command to remove paths you no longer want or need.

If you write an MSF in one of the directories in the path, you can call it immediately from within Xmath. When you call a function you have written, Xmath searches your current directory and all the directories in your path until it finds a function file where name matches the function you called. Upon finding the file, Xmath compiles it to a low-level operational code and it runs immediately. If a function or command file is not in a directory listed as one of your path directories, you need to define it explicitly and specify the directory where it resides.

[Table C-19](#) compares these facilities in Xmath and MATLAB (the operating system commands shown are for a version of MATLAB running under a UNIX operating system).

Table C-19 Examples of Finding Files

MATLAB	Xmath
!pwd	show directory
!cd /home/new	set directory = "/home/new"
!echo \$MATLABPATH % or matlabpath	show path

Table C-19 **Examples of Finding Files** (Continued)

MATLAB	Xmath
<code>matlabpath("~/me/myfuns")</code>	<code>set path = "~/me/myfuns"</code>
(no analogous feature)	<code>remove path 2</code>

In MATLAB, the exclamation-point notation (!) can be used to send out an operating-system command and display its output. Xmath offers an analogous `oscmd` function, as shown in [Table C-20](#).

Table C-20 **Operating System Command Examples**

MATLAB	Xmath
<code>!ls -l</code>	<code>oscmd("ls -l")</code>

### C.3.11 Debugging Files (on UNIX systems)

MATLAB's debugging facility consists primarily of keyboard commands. Xmath provides an interactive debugger for MathScript files. It can be used in either of two modes:

- The Xmath debugger is automatically invoked when you try to run a function containing a syntax error. The offending statement is highlighted. You can fix the mistake, save the file, and rerun the function, all from the debugger window.
- The second debugging mode is useful when you have written a function and want to halt execution at some point to examine variable values. To do this, type **debug functionName** in the Xmath Commands window. The debugger will then appear when you call the function, allowing you to step through any portion of the MSF one statement at the time, or to set breakpoints and jump to them. You can use the Commands window to look at local variable values or evaluate expressions.

For more on the debugger window, see [6.4 Using the Xmath Debugger](#), p.245.

### C.3.12 Save and Load

The commands for saving and loading data also differ somewhat. MATLAB offers flags that enable you to save data in either a MATLAB-written binary format or a short or long ASCII format. Xmath's **SAVE** command has a number of keywords

associated with it to determine what type of format to use to save the data. Xmath's **LOAD** command can be used to load in data saved by Xmath or MATRIX<sub>x</sub> (FSAVE-format data). [Table C-21](#) compares the commands.

Table C-21 **Save and Load Examples**

MATLAB	Xmath
SAVE 'filename' VAR1 VAR2...	save var1 var2... file = "filename" {keyword} # The keyword is optional and # may be set for binary, ASCII, # or MATRIXx formatted saves
LOAD 'filename'	load "filename"
No equivalent feature	load a b "filename"

In Xmath, as in MATLAB, if a list of variables to be saved or loaded is omitted, all variables are saved or loaded. Xmath data files terminate with the suffix **.xmd** (MATLAB uses the **.mat** suffix). MATLAB always loads all the data stored in a **.mat** file; Xmath can load either all or part of the data stored in a **.xmd** file.

Xmath's **load** command cannot directly load MATLAB data; however, as described in [Chapter 8](#), you can create a linked executable (LNX) that can. **\$XMATH/src/matload.c** is a sample LNX that loads MATRIX<sub>x</sub> 3.X format (which is similar to older MATLAB formats) into Xmath data objects. This file is commented to assist you in making any changes. To make a local copy of **matload.c** in the Xmath log area and create a local copy, go to the command area and type:

```
copyfile "$XMATH/src/matload.c"
```

### **Loading In External Data (read)**

Loading in data generated by external programs other than Xmath, MATRIX<sub>x</sub>, and MATLAB is also possible. If you have data written to a non-Xmath file by another program and you know the size and type of the data in the file, you can use the **read()** function to read from the data file into an Xmath matrix variable. The input arguments you pass to **read()** describe how large the matrix should be, the format of the data in the external file, and how many bytes of data (if any) you choose to skip before reading data into the target variable. This allows you to create data files that are easily readable by a variety of programs, not necessarily just Xmath. This function is described in more detail in the Xmath online Help,

and the `$XMATH/demos` directory contains sample files that you can use to test `read()`.

### Writing Data to an External File (`print`, `fprintf`)

In addition to the Xmath-formatted `SAVE` command, Xmath provides two other functions that are useful for writing data to external files: `print()` and `fprintf()`.

`print()` writes any Xmath data object to an external file you specify. The data is written exactly as it appears when displayed in the Commands window log area.

`fprintf()` converts scalar numeric values to a string representation, then writes them to the file you specify. A wide range of format specifiers (identical to the ones used for the C-language `fprintf()` function) can be used to specify field width, zero-padding, tabs, and new lines, among other formatting options.

### C.3.13 Useful Aliases

You may want to define the following aliases in a `startup.ms` file so that you can use familiar names for the following Xmath commands. Some examples follow.

```
alias    clear    delete
alias    ss2tf    numden
alias    tf2ss    abcd
```

The MATLAB function names `lyap()` and `conv()` invoke their Xmath counterparts `lyapunov()` and `convolve()`, but the Xmath functions have a different set or order of inputs. Along these lines, Xmath has both a `rootlocus()` and `rlocus()` function. `rlocus()` is the one analogous to MATLAB `rootlocus()`. To create an alias enter the following:

```
alias    rlocus    rootlocus
```

Xmath versions do not necessarily take exactly the same inputs in exactly the same order as their MATLAB namesakes. When in doubt, refer to the Xmath online Help.

You are, of course, not limited to these aliases. Xmath commands and functions tend to be as descriptive as possible without being excessively long. As you acquire expertise with Xmath, you will probably want to alias other frequently

used commands as well. To obtain a list of all the aliases currently set up in an Xmath session, just type **alias** on a line by itself in the command area.

Note that aliases can cause some problems; for example, if you have **clear** defined as an alias for **delete**, you will not be able to use **clear** as a keyword in a function. We recommend that you use aliases to speed your transition from MATLAB to Xmath, and then learn the Xmath syntax as you go along.

## C.4 Comparison of Frequently Used Commands

Table C-22 summarizes some of the most frequently used Xmath and corresponding MATLAB commands. Both Xmath and MATLAB commands are case insensitive.

Table C-22 **Xmath and MATLAB Summary of Frequently Used Commands**

Xmath Command or Operator	MATLAB Command or Operator	Description
<b>cond(A)</b>	<b>cond(A)</b>	Finds the condition number.
<b>convolve or *</b>	<b>conv</b>	Performs polynomial and vector convolution.
<b>cos(x)</b>	<b>cos(x)</b>	Calculates the trigonometric <b>cos</b> function.
<b>bode</b>	<b>bode or dbode</b>	The Xmath function <b>bode</b> checks whether your system is continuous or discrete and then performs the appropriate operation.
<b>det(A)</b>	<b>det(A)</b>	Finds the determinant.
<b>eig(A)</b>	<b>eig(A)</b>	Computes eigenvalues and eigenvectors for real and complex square matrices.
<b>execute</b>	<b>eval</b>	Xmath and MATLAB versions perform similar functions (see <a href="#">3.9 MathScript Batch Files</a> , p.97).

Table C-22 Xmath and MATLAB Summary of Frequently Used Commands (Continued)

Xmath Command or Operator	MATLAB Command or Operator	Description
execute file	<i>file.m</i>	Executable files are similar (see <a href="#">3.9 MathScript Batch Files</a> , p.97).  <b>NOTE:</b> In MATLAB, execution of a script can be done directly from the script's name. In Xmath, execution must be done with the <b>execute( )</b> function. (This prevents ambiguous code or accidental execution.)
exp(x)	exp(x)	Computes the exponent of (x).  <b>NOTE:</b> For matrix exponentiation, MATLAB requires the format <b>expm(A)</b> .
eye(A)	eye(A)	Generates the identity matrix.
GuiPlotGet	ginput	Get the current pointer selection and coordinates.
hessenberg	hess	Converts a matrix to Hessenberg form.
hilbert(n)	hilb(n)	Creates a Hilbert ill-conditioned matrix.
hilberttransform	<b>No MATLAB equivalent</b>	{Apparently this command is available but not documented.}
inv(A)	inv(A)	Finds the inverse matrix.
load	load	Xmath and MATLAB versions perform similar functions (see <a href="#">3.7 Saving and Loading Data</a> , p.88).
log(x)	log(x)	Computes natural logarithm.
makepoly	<b>No corresponding command</b>	Create a polynomial from its coefficients. MATLAB reformats polynomials by using the corresponding vector with its coefficients.

Table C-22 Xmath and MATLAB Summary of Frequently Used Commands (Continued)

Xmath Command or Operator	MATLAB Command or Operator	Description
<b>norm</b>	<b>norm</b>	Calculates the norm of a vector, matrix, or PDM (Xmath only).
<b>No corresponding command?</b>	<b>null</b>	The <b>NULL(A)</b> command in MATLAB calculates an orthonormal basis for the null space of A.
<b>ones</b>	<b>ones</b>	Xmath and MATLAB have some minor syntactical differences (see MATRIX <sub>X</sub> online Help).
<b>ortho</b>	<b>orth</b>	Used as <b>ortho(A)</b> to give the orthonormal basis for A.
<b>pinv</b>	<b>pinv</b>	Used as <b>pinv(A)</b> to give the pseudoinverse for A.
<b>plot(0:10)</b>	<b>plot(0:10)</b>	The basic plot command is the same, but the keyword syntax is different.
<b>polyfit</b>	<b>polyfit</b>	Both commands fit a polynomial, but the Xmath command uses a PDM as input. (PDMs are not supported in MATLAB.)
<b>polynomial</b>	<b>poly</b>	Create a polynomial from its roots.
<b>polyval</b>	<b>polyval</b>	Evaluates a polynomial.
<b>No corresponding command</b>	<b>quad8</b> <b>quad -dblquad</b>	Estimates an integral numerically.
<b>random</b>	<b>rand</b>	Generates random numbers or matrices.
<b>residue(sys)</b>	<b>residue(b,a)</b>	Expands a partial fraction.
<b>roots(p)</b>	<b>roots(p)</b>	Returns the roots of a polynomial.
<b>round</b>	<b>round</b>	Round matrix values to the nearest integer.
<b>rref</b>	<b>rref</b>	Transforms a matrix into reduced echelon form.



Table C-22 Xmath and MATLAB Summary of Frequently Used Commands (Continued)

Xmath Command or Operator	MATLAB Command or Operator	Description
save	save	Xmath and MATLAB versions perform similar functions (see <a href="#">3.7 Saving and Loading Data</a> , p.88).
schur	schur	Calculates the Schur factorization.
set format name	format name short, short e, long, long e, hex, bank, compact, loose, rat	Xmath format names are <b>compact (the default), engineering, fixed, long, longe, scientific, short, and shorte</b> .
set seed num	rand('seed', num)	Setting the random number seed. For MATLAB 5, <b>rand('State', j)</b> gives the jth state. <b>rand('State', s)</b> makes the actual state equal to s ( <b>state = s</b> ).
sin(x)	sin(x)	Calculates the trigonometric <b>sin( )</b> function.
sqrt(x)	sqrt(x)	Calculates the square root of x.
zeros	zeros	Generates a matrix of zeros.
./, *, ...	./, *, ...	Point preceding operator means elementwise operation.
' and *'	.' and '	Transpose operators. The operators on the left (' and '.') are for regular transpose and The operators on the right are for complex-conjugate transpose.
+	parallel	The overloaded Xmath + operator performs the same function as the MATLAB <b>parallel( )</b> function.
*	series	The overloaded Xmath * operator performs the same function as the MATLAB <b>series( )</b> function.

Table C-22 Xmath and MATLAB Summary of Frequently Used Commands (Continued)

Xmath Command or Operator	MATLAB Command or Operator	Description
# ... # or # # . . . #	%	Indicates a comment. MATRIX <sub>χ</sub> supports block comments that span multiple lines. The second # is only needed for a multiple-line comment.
"string"	'string'	MATRIX <sub>χ</sub> uses double quotes to avoid confusion with the transpose operator.
A <> B	A~=B	Logical NOT EQUAL operators.
!A	~A	Logical NOT operators.
x = A\b	x = A\b	Computes the least squares approximation (Ax = b).

# D

## *Xmath to Mathematica Interface*

This appendix describes how to set up and use the Xmath to Mathematica Interface.

### ***D.1 Overview***

Mathematica is a powerful symbolic manipulation program from Wolfram Research, Inc. (WRI). It performs operations such as differentiation and integration symbolically, achieving exact general solutions to many problems. This capability can be coupled with Xmath's powerful numerical analysis and design capabilities, resulting in a very strong joint analysis tool.

Xmath was developed with an open architecture, which simplifies communication with other programs and processes. The interface between Xmath and Mathematica is based on Xmath's LNX (link external) capability and Mathematica's Mathlink facility. When Mathematica is first invoked from Xmath, Mathematica's Mathlink facility establishes a process running Mathematica, and maintains a link to that process for all subsequent calls from the same Xmath session, allowing Xmath the use of intermediate variables in Mathematica. Furthermore, the Mathlink facility allows Mathematica to be invoked on a different computer than the Xmath host; this is completely transparent to the user.

When a valid Mathematica command is entered in the Commands window command area, a separate Mathematica process computes the answer (commands that produce graphics should never be used). The resulting text output that

would normally appear in Mathematica is converted to an Xmath string object that is displayed in the Commands window log area. If the answer is a numeric matrix, it can be passed directly to Xmath. Matrices can also be passed from Xmath to Mathematica. All Mathematica warnings and other messages are transmitted to Xmath and displayed in the Commands window message area.

For a more detailed explanation of the LNX process, see [Chapter 8](#). The source for the Xmath to Mathematica interface can be found in `$XMATH/src/mathlink.c`.

## D.2 Setup

These instructions apply to setting up Xmath, and the Xmath to Mathematica interface on an Xmath host. If you encounter problems related to Mathematica functionality, contact Wolfram Research, Inc. Their website is <http://www.wri.com>, their e-mail address is [support@wolfram.com](mailto:support@wolfram.com), and their Technical Support phone number in the USA is 217-398-6500.

To use the Xmath to Mathematica interface the following conditions must be met.

- Mathematica must be installed and accessible to you, the Xmath user. The Mathematica version must be 3.0 or higher.
- Only UNIX versions of Xmath and Mathematica are supported.
- Your UNIX execution path must include the path to your Mathematica installation directory. For example,

```
set path = ($path /home/Mathematica/Executables/SPARC)
```

where the above path points to the Mathematica installation at your site.

Because the Mathematica interface LNX must be linked with the local Mathlink libraries on your target system, Integrated Systems cannot deliver an executable interface. However, we have provided all of the necessary routines to quickly create an executable interface LNX.

To allow all users access to the Xmath to Mathematica interface, a system administrator must perform the steps in [D.2.1 Setting Up the Xmath to Mathematica Interface for All Users](#), p.405. Users who do not have system privileges can perform the steps in [Creating a Local LNX \(Single User\)](#) to create a local LNX.

### D.2.1 Setting Up the Xmath to Mathematica Interface for All Users

These instructions assume the `$SIHOME` environment variable was properly set at installation time to the path to the root MATRIX<sub>X</sub> Product Family installation.

1. Change directory to `$SIHOME/platform/xmath/src`.
2. Edit the `mma.mk` file as follows:
  - a. Define the `XMATH` variable to be `$(SIHOME)/platform/xmath`.
  - b. Replace `PATH_TO_libML.a` with the path to the Mathlink libraries (for example:

```
*/Mathematica/AddOns/MathLink/DevKits/SPARC/CompilerAddOns
```

3. To create `mmalnx.lnx`, run the makefile as follows:

```
make -f mma.mk
```

4. Copy `mmalnx.lnx` to `$SIHOME/platform/xmath/modules/mathematica`.

All Xmath users will now be able to use the Xmath to Mathematica interface.

### D.2.2 Creating a Local LNX (Single User)

Although it is preferable to have a system administrator set up the Xmath to Mathematica interface, a user with no root privileges can set up an LNX for his personal use. To use the Mathematica interface without modifying the Xmath source directories, the user can copy `$SIHOME/platform/xmath/src/mma.mk` and `mathlink.c` to a local directory, and then perform steps 2 and 3 described in [D.2.1 Setting Up the Xmath to Mathematica Interface for All Users](#), p.405. However, before using the interface, the user must tell Xmath not to look in the modules directory for the LNX. To do this type the following Xmath commands in the Xmath Commands window command area:

```
undefine mma
define mma {directory="path_to_lnx"}
```

For future usability, these lines can be placed in your personal Xmath `startup.ms` file, along with a `set path` command that points to the location of your local `mmalnx.lnx` file (so that if you start Xmath from another directory you will still be able to use the LNX). For example:

```
set path="path_to_lnx"
```

See [3.9.3 startup.ms \(on UNIX systems\)](#), p.98 for more on **startup.ms**, and [Search Paths](#) on p.231.

## D.3 Syntax

Xmath provides three functions, which perform the following tasks:

1. Send a command to Mathematica (a Mathematica session is started if one does not exist):

```
mma("valid_Mathematica_cmd")
```

**mma** is actually an accepted abbreviation for **mmaexecute**.

2. Transfer a matrix from Xmath to Mathematica:

```
mmaput("mma_matrix_name", xmath_matrix_name)
```

3. Transfer a matrix from Mathematica to Xmath:

```
xmath_matrix_name = mmaget("mma_matrix_name")
```

Note that you can assign the output of a Mathematica command to a Mathematica variable and an Xmath variable in one step:

```
xmath_var = mmaget("var=Mathematica_numerics_cmd")
```

4. Close the Mathematica session:

```
mma("quit")
```

The lowercase string **"quit"** causes the LNX to close the Mathematica process. However, the LNX stays resident and active. If you issue another Mathematica command, the existing LNX will restart Mathematica. When you exit Xmath, the LNX will be killed.

### D.3.1 Passing Xmath Data to Mathematica

You can pass scalars, vectors, or matrices from Xmath to Mathematica. These forms all qualify as matrix objects in Xmath.

Mathematica assumes all incoming values are matrices and places them in nested Lists. For example, the Xmath matrix  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$  is represented as `{{1,2},{3,4}}` when passed to Mathematica, and the Xmath scalar 7.2 is represented as `{{7.2}}`.

Xmath vectors should always be passed to Mathematica as *row* vectors. (If a column vector is passed, the resulting nested list will not be as readily useful.) After a row vector is passed to Mathematica, it can be extracted from a nested List to a single List using `x=x[[1]]`. Scalars can be extracted from a nested List to a true scalar using `s=s[[1,1]]`.

### D.3.2 Passing Mathematica Data to Xmath

Lists or nested Lists can be passed to Xmath from Mathematica. The Lists can only contain numerical data, never symbolic data. In the case of nested Lists, the component List lengths must be equal so that Xmath can convert the List to a matrix.

A Mathematica symbolic matrix can be converted to a numerical equivalent using the command `x=N[x]`, and the result can then be passed to Xmath. For example,

```
mma("x = Table[EllipticK[i], {i, 0, 2/3, 1/6}]" )
x=mmaget("N[x]" )
```

To pass a scalar to Xmath it must first be placed in a List of length one. This can be done using the command `a={a}`.

## D.4 Examples

The following Xmath inputs and Mathematica responses demonstrate how data is passed between the applications.

When we ask for the Mathematica version, Xmath receives a string:

```
mma("$Version" )
```

```
ans (a string) = SPARC 3.0 (April 26, 1997)
```

The following call to Mathematica asks for a numeric result with a precision of 40.

```
str=mma("N[EulerGamma,40]")
```

```
str (a string) = 0.5772156649015328606065120900824024310422
```

We can convert this simple string to a number in Xmath and compare the displays. First, we set the format to **longe**, the longest output Xmath can display. Then we can convert the string to a scalar with **makematrix**:

```
set format longe  
s=makem(str)
```

```
s (a scalar) = 5.772156649015329e-01
```

Symbolic output (strings containing superscripts or a mixture of text and numbers) can be viewed in the Xmath Commands Window log area, but not used as Xmath inputs:

```
mma("Integrate[x^2 Sin[x]^2,x]")
```

```
ans (a string) =  
      3      2  
4 x  - 6 x Cos[2 x] + 3 Sin[2 x] - 6 x  Sin[2 x]  
-----  
                        24
```

Create a matrix in Xmath and send it to Mathematica:

```
set format compact  
m=[pi,42,0;7,tiny,6;17,huge,.02]
```

```
m (a square matrix) =
```

```
  3.14159      42      0  
  7      2.22507e-308  6  
 17      1.79769e+308  0.02
```

```
mmaput("m",m)
```

You can use Mathematica functions to manipulate the matrix and pass numeric versions of the matrix manipulations back to Xmath:

```
mRev=mmaget("mRev=N[Reverse[m],9]");  
mRot=mmaget("mRot=N[RotateLeft[m,2],9]");
```



Display the matrices (in compact form):

**mRev?**

```
17          Inf          0.02
 7          2.22507e-308  6
3.14159     42           0
```

**mRot?**

mRot (a square matrix) =

```
3.14159     42           0
17          Inf          0.02
 7          2.22507e-308  6
```

For more examples, execute the files `$XMATH/demos/mathematica/mma.ms` and `$XMATH/demos/mathematica/elliptic.ms`.



# Index

## Symbols

- 93  
- 94  
! 91, 93, 94, 107  
% 97  
& 91, 93, 94  
\* 93, 94, 97  
\*\* 93, 94  
\*' 93, 94  
+ 93, 94  
+ operator 72  
, 191  
.\* 93, 94  
.\*\* 93, 94  
.\*. 93, 94  
... 103  
./ 93, 94  
./.. 93, 94  
.\ 93  
.^ 93  
. 93, 94  
/ 93, 94  
: (variable arguments) 272  
: in regular vector specifier 73  
: index operator 94  
; 102, 113  
< 91, 93, 94  
<= 91, 94  
<> 91, 94, 284  
= 94  
== 91, 94  
=== 278  
> 91, 93, 94  
>= 91, 94  
? 113, 117  
@, @str, @int, @:l, @str:l, @str:p, @@, @@:p 44  
[] 72, 73, 94  
\ 93  
^ 93  
# 103  
#{ }# 103  
( ) 94  
{ } 94, 106  
| 91, 93, 94  
' 73, 93, 94

## A

**abcd** 233  
abort Xmath (Ctrl-\, UNIX) 30  
advanced topics 272  
**alias** 106, 116  
**AllocateList** 302  
**AllocateMatrix** 298  
**AllocateNull** 303  
**AllocateStringMatrix** 299

**ans** 50, 99  
apostrophe operator (") 391  
**argn** 272  
**argv** 272  
arrays, C vs. FORTRAN 328  
**ascii** 236  
assignment statement 89  
asterisk-apostrophe ("\*") 391  
**autocompile** 113  
axes 148

## B

background LNX 294, 324  
    function assignment syntax 276  
    terminating 327  
batch file 117  
    creating 120  
    running 120  
**beep** 263  
behavior  
    GUI objects 345  
binary operators 92  
breakpoint  
    remove 269  
    set 266, 268  
    show 269  
breakpoints 268  
button  
    radio 345  
    toggle 345

## C

C language  
    arrays vs. FORTRAN 328  
    LNX function file format 295, 296  
    resident functions 331  
cascade menu 345  
channels 214, 225, 226  
**char** 236  
**check** 233, 259

class  
    hierarchy 188  
    variables 282  
        computed 282  
        optional 282  
        required 282  
clear  
    logarea 38  
click 174, 367  
client 366  
colon (:)  
    in regular vector specifier 73  
    index operator 94, 194, 195, 272  
color map 137  
    current 137  
    using your own 138  
**colorind** function 126  
column vector 73, 196  
comma (,) 191  
command  
    area 38, 49  
        editing text 40  
        recall 43  
    declaration 245  
    diary 121  
    intrinsic 58, 105  
    syntax rules 105, 106  
    using 58  
Commands window 35  
**comment** 50, 97  
    add via dialog 100  
    partition 97  
    variable 97  
**commentof** 97  
comments 50  
    multiple lines (#{ }#) 103  
    single lines (#) 103  
comparator 92  
complex  
    conjugate transpose (\*) 93  
    matrix 190  
    number 190  
concatenate 191, 235, 239  
concatenation  
    lists 239

- matrix 191
- PDM 221
- strings 235
- connection
  - parallel 231
  - series 231
- continuing Xmath command lines 103
- contour plot 168
- convert using **check** 233
- copy** 62
- copying and pasting text
  - in Motif 371
  - within Xmath 371
- Ctrl-\ 30
- Ctrl-Break 30
- Ctrl-c 30
- cursor 370

## D

- data structure
  - et\_list** 302
  - et\_matrix** 297
  - et\_null** 303
  - et\_pdm** 299, 300
  - et\_string** 298
- data type
  - list 302
  - matrix 297
  - null 303
  - PDM 299
  - string 298
- dbx 330
- debug** 266
  - off** 269, 270
  - set** 268
- debugger window 86, 265
  - debugging mode 266
  - setting breakpoints 268
  - setting watchpoints 269
- debugging an LNX 330
- declaration line 244, 245, 247
- default values 248
- DEFINE** 252
- #define** 306
- defTimeRange** 234
- DeleteAny** 303
- DeleteList** 302
- DeleteMatrix** 298
- DeleteNull** 303
- DeletePDM** 302
- DeleteString** 299
- delsubstr** 237
- demo
  - debugger 87
  - graphics 70
  - guidemo 88
  - leadlag 341
- diagonal** 199
- diagonal matrix 199
- dialog 346
  - modal 346
- diary 122
  - command 121
  - session 122
- directory
  - pathnames, specifying in Xmath Command
    - area 38
  - set 28, 108
  - show 108
- directory** 276
- discretize** 233
- DISPLAY** 236
- display** environment variable 113
- distribution, random 114
- domain 81, 207, 209
- domain** 218
- double-click 367
- drag 174, 367
- dynamic system 227, 387
  - indexing 231
  - operators 230
  - size 230
  - state-space form 227, 228
  - transfer function form 228

## E

### echo

set 121  
show 118

editing text in Xmath 40

ellipsis 103, 382

environment variable

expanding in script files 115  
set 112  
XMATH\_PRINT 118, 119

eps 98

ERASE 127, 171

erase

logarea 38

err (permanent variable) 98

error

codes 335  
handling, LNX 304, 307

error 260, 263

et\_list 302

et\_matrix 297

et\_null 303

et\_pdm 299

et\_string 298

EVAL 393

executable string 117

execute 117, 122

exist 258

exponentiation 93

expression 90

extended-selection list 345

eye 199, 385

## F

face\_color 159

face\_style 159

fg\_color 158

file selection dialog 39, 372

filenames, specifying in Xmath Command area 38

find 75, 264

find variable 101

fonts 152

for 104, 256, 386

format, numerical display 114

FORTRAN 328

FORTRAN LNX 327

fprintf 111

ftnlx 329

functions

intrinsic 105  
MIMO 107  
nonresident 332  
resident 331  
syntax rules 105  
using 57  
void 248, 249

## G

G <MCurrPage `327> 341

general simulation 84

get 115

path 116

getchoice 261

getline 261

goto 258

graph object 130

bind to variable 131, 179

graphical user interface. See *GUI*

Graphics window 60

graphics window 123, 171

colors 138

grids 150

grip 34

GUI

Help menus 342

objects 345

tools

developing your own 342

using 341

guidemo 88

**H**

**HARDCOPY** 127  
 hardcopy (graphics) 180  
**HARDCOPY** command 181  
 Help  
   messages 346  
   window 45  
   **xmath**  
     Windows 30  
 help  
   **xmath**  
     UNIX 29  
**hessenberg** 200  
 Hessenberg matrix 200  
 history. See *recall*  
**huge** 98

**I**

icon bar 175  
 iconify 368  
 identity matrix 199  
 if 104, 257, 386  
 improper transfer function 228  
**impulse** 234  
 independent parameter 209  
 index  
   list 194, 238, 239, 264  
   operator 288  
**index** 237, 264  
 indexing  
   dynamic systems 231  
   functions 264  
   matrices 74, 193  
   PDMs 83  
**Inf** 98  
**initial** 234  
 initializer function 280  
 input names, extracting 233  
 interrupt  
   Ctrl-Break 30  
   Ctrl-c 30  
 intrinsic functions 105

ISIHOME 25

**J**

Jay 98, 386

**K**

**keep** 62  
 key bindings  
   changing 43  
   default 41  
 keyboard, remapping 375  
 keywords 106, 131  
   assigning default values 248  
 Kronecker product 93

**L**

label 258  
 leadlag demo 341  
**length** 235  
**licenseinfo** 32  
 line  
   feed 40, 49, 190  
   styles 140  
   widths 140  
 list 345  
   extended-selection 345  
   multiple-selection 345  
   object 84, 238  
   single-selection 345  
**list** 238  
 LNX  
   background  
     function assignment syntax 276  
     mode 294, 324  
     terminating 327  
     windows client, communicating with 335  
   building and calling 316  
   C function format 295, 296

data type. See *data type*  
debugging with dbx 330, 332  
definition 293  
FORTRAN 327, 328  
function  
    communicating with Xmath 304  
    sample 314, 321  
functions 303  
    AllocateList 302  
    AllocateMatrix 298  
    AllocateNull 303  
    AllocateStringMatrix 299  
    DeleteAny 303  
    DeleteList 302  
    DeleteMatrix 298  
    DeleteNull 303  
    DeletePDM 302  
    DeleteString 299  
    WrapMatrix 298  
    WrapPDM 302  
    WrapString 299  
    WrapStringMatrix 299  
    XmathIPCgetc 336  
    XmathIPCgeti 336  
    XmathIPCgets 336  
    XmathLoad 312  
    XmathSave 311, 312  
handling aborted 334  
include file, required 297  
interfacing Xmath with Mathematica 403,  
    405–406  
limitations on passing variables 292  
loading MATLAB data 396  
**makefile** 316  
nonresident functions 332  
program, sample 294, 325  
prototype 295, 296  
resident functions 331  
speeding execution for MSOs 289  
string data type, converting to 299  
UCI comparison 293  
undefining 320  
user function structure 294, 296  
USR1 signal handler 330  
utility 294

    version compatibility 297  
**load** 32, 109, 395  
log area 37  
    clear 38  
logical 91  
logspaced vector 73, 197  
loop 104  
    **for** 104, 256  
    **if** 104  
    **if** 257  
    **while** 104, 256, 257  
lower triangular matrix 201

## M

**makecontinuous** 233  
**makefile** 316, 319  
    for an LNX program 316, 319  
    template 316, 319  
**makematrix** 222, 236  
    converts strings to numbers 261  
**makepoly** 76, 205  
markers 141  
Mathematica to Xmath Interface 403  
MathScript 85, 89  
    files 117  
        batch 117  
        execute 117  
        format 246  
    function. See *MSF*  
    object. See *MSO*  
    programming 248  
    punctuation 102  
    scoping rules 249  
    search paths 251  
MATLAB  
    data  
        using LNX to load 396  
    to Xmath translator  
        aliases 397  
        syntax difference 382  
**matload.c** 396  
matrix 190  
    building 72



- brackets 190
  - commas 190
  - line feed 190
  - semicolons 190
  - concatenation 191
  - data type 297
  - diagonal 199
  - Hessenberg 200
  - identity 199
  - indexing 74, 193
  - operators 191, 192
  - punctuation 190
  - square 197
  - string 71
  - symmetric 198
  - Toeplitz 200
  - triangular 201
  - MATRIX<sub>X</sub>
    - block diagram 343, 344, 346
  - MATRIX<sub>X</sub> online Help 24
  - menu
    - bar 34
    - cascade 345
    - pulldown 345
    - selection from keyboard 371
  - message area 44
  - Meta key 35
  - MIMO, definition and representation 78
  - mma** 406
  - mmaget** 406
  - mmaput** 406
  - modal dialog 346
  - mouse
    - click 367
    - double-click 367
    - drag 367
    - instructions 367
    - press, push 367
    - selecting text
      - by clicking 41, 371
      - by dragging 41, 371
  - move 157
    - graphic objects 174
    - window 369, 370
  - MSC 244
    - building 246
    - command declaration 245
    - example 255
    - file format (figure) 247
    - inputs 244
    - inputs (syntax) 245
    - scoping rules 249
    - user-interface functions 260
    - variable arguments 272
  - MSF 243, 244
    - building 246
    - calling syntax 105
    - file format (figure) 246
    - function declaration 244
    - Help 244
    - inputs 244
    - optional block comment 244
    - scoping rules 249
    - user-interface functions 260
    - variable arguments 272
  - MSO 279
    - defining 281
    - index operators 287
    - initializer function 282
    - member entities 292
    - object instantiation 280
    - operator overloading 285
    - scoping (nested objects) 283
    - speeding execution with LNXs 289
    - type declaration 284
  - multiple-selection list 345
- ## N
- names** 218, 233
  - names, specifying directory pathnames and filenames 38
  - naming rules 90
  - NaN** 98
  - negation operator (!) 107
  - new partition** 95
  - nomenclature 23
  - nonresident 305
  - null** 98

**numden** 233

## O

**ones** 385

operators 92

and PDMs 223

indexing 74

matrix 192

precedence 94

with dynamic systems 230

with polynomials 76

optional arguments, assigning default values 248

**oscmd** 55, 108

expanding path names 116

output

keywords 248

names, extracting 233

## P

**PARALLEL** 392

parallel connection 231

parameter-dependent matrix. See *PDM*

parentheses 194

partition 95, 97

changing via variables window 100

delete 53, 96

handling 52

lock 101

name 90

new 52

set 52, 114

show 53, 95

size 100

viewing variables 53, 100

partition, definition 51

pastings selected text in Motif 371

path 251

adding (**set path**) 251

overriding (**define**) 252

removing (**remove path**) 251

**set** 251

specifying 38

viewing (**show path**) 251

path name

expanding in script files 115

pathnames 115

**pause** 262

PDM 207

allocate for LNX or UCI 300

channel 214, 225

concatenation 221

convert to matrix 222

creating 210, 211

dimensions 215

domain 209

extracting 218

independent parameter 209

indexing 215, 218

substitution 220

modifying 220

names 209, 218

extracting 218

operators 223

using with functions 225

**pdm** 210

PDM, definition 81

**pdmplot** function 127

**period** 233

permanent variables 98

**pi** 98

plot 347

and mouse buttons 347

**plot** 58, 59, 123, 124, 128, 131

complex data 128

copy 145

datestamp 182

drawing tools 176

edit graphics window 181

font sizes 182

icon bar 182

interactive tools 175

keep 145

timestamp 181

toolbar 175

zoom 177

plot keywords

- animate 155
- axes 148
- axisfix 148
- bg\_color 158
- colors 137
- contour 168
- defaults 132
- edge 159
- face 158
- face\_color 159
- face\_style 159
- fg\_color 158
- grid 150
- hold 161
- increments 150
- keep 146
- keepssubplot 146
- labels 135
- legend 136, 182
- light 160, 181
- line 140
- marker 141
- move 157
- polar 169
- position 157
- projection 156, 181
- reset 161, 181
- rotate 156
- rows and columns 143
- scale 156
- strip 165
- text 152
- tic labels 150
- tics 149
- titles 135
- zero lines 148

**plot2d** function 59, 123, 125

plotting commands 126

plotting functions 123

- comparative analysis 125
- special purpose 126

plus (+) operator 235, 239, 288

point (verb) 370

polar plot 169

polynomial 76, 205, 387

- addition 77
- default variable 76
- indexing 77
- multiplication 76
- operators 206

**polynomial** 76, 205

**polyval** 77

position 157

power, raise to 93

precision (**set format fixed**) 114

press 367

**print** 55, 111

print a graphics file 180

**PRINTER** 26, 180

proper transfer function 228

pull-down menu 345

punctuation, MathScript 102

push 367

## Q

**qplot** function 127

question mark (?) 113

**quit** 31

- in batch .ms files 120

## R

radio button 345

raise to a power 93

random

- distribution (**set**) 114
- seed (**set seed**) 114

**random** 385, 389

**read** 112

recall

- @ sequences 43
- ctrl sequences 42, 43

regular vector 73, 196

**remove** 115

- break** 269

- commanddiary 122
- path** 251
- sessiondiary 122
- watch** 270
- resident
  - function 305, 331
  - process 305
- resize window 369, 370
- resizing window 34
- restore 368
- roots** 77
- rotate 156

## S

- sample period, extract with **period** 233
- save
  - all 37
- save** 31, 54, 108, 395
  - PDMs as matrices 223
  - simulation data 223
- save.xmd** 31, 109
- scalar 201
- scale 156
- scoping (in scripts) 249
- scroll bars 34
- search path 251
- selecting
  - object
    - by clicking 60
    - by Shift-clicking 60
  - text
    - by clicking 371
    - by dragging 371
- semicolon (;) 102, 113, 191
- SERIES** 392
- series connection 231
- set** 112, 113
  - autocompile 253
  - break 113
  - buffering 113
  - commanddiary 113, 121
  - debugonerror 113
  - directory 28, 108, 113, 115

- display 113
- echo 38, 114, 118, 121
- format 114
- logarea 37
- partition 52, 114
- path 114
- path** 251
- pause 114, 262
- seed 114
- sessiondiary 114, 122
- timestamp 114
- uiupdate 115
- watch 115
- Shift-Enter 51
- Shift-Return 51
- show** 115
  - break 269
  - echo 118
  - logarea 38
  - partition(s) 95
  - path 251
  - path** 251
  - seed 114
  - watch** 269
- simulation, general 84
- single-selection list 345
- SISO, definition and representation 78
- slider 347
- square matrix 197
- start Xmath 22, 28
- startup.ms** 118, 119
- state names, extracting 233
- statement 89
- state-space system 78, 79, 227, 228
  - decompose with **abcd** 233
- step** 234
- string 71, 235, 387
  - breaking across lines 72
  - concatenation 235
  - converting numbers 236
  - data type 298
  - executable 117
  - indexing 237
  - matrix 235
  - plus (+) operator 235

- size of 235
- special characters 236
- stringex** 237
- strip plots 165
- symmetric matrix 198
- sys\*u** (time domain sim) 228
- system** 78, 229, 230
- system.** See *dynamic system*

## T

- target directory 108
- template.f** 328
- text 40
  - entry area 345
- tics 149
- time response 234
- timestamp 114
- tiny** 98
- Toeplitz matrix 200
- toolbar 175
- transfer function 78, 228
  - converted to state space before decomposition 233
- transpose (') 73, 93
- transpose, complex conjugate (\*) 93
- triangular matrix 201
- tril** 201
- triu** 201

## U

- UCI 296, 320
  - building and calling 316
  - cleanup after termination with **-clean** 29, 30
  - functions 303
  - include file, required 297
  - LNX comparison 293
  - start with **-call** 29, 30, 294, 321
  - Xmath
    - computational engine 322
    - graphics engine 322

- XmathExecute** 308
- XmathGet** 309
- XmathPanic** 335
- XmathPut** 309
- XmathStart** 314
- XmathStop** 314
- uiPlot** function 59, 123, 124
- uiPlotArea** function 127
- uiPlotGet** function 127
- unalias** 116
- unary operator 92, 94
- UNDEFINE** 253, 320
- upper triangular matrix 201
- user interface functions 260
- user-callable interface. See *UCI*
- USR1 signal handler 330, 335

## V

- variable 49, 50
  - comment 97
  - creating 49
  - edit box 346
  - environment, changing 112
  - find 101
  - load 101
  - lock 99, 101
  - name includes partition name 95
  - naming 90
  - permanent 98
  - print to file (**print**) 55, 111
  - save 101
  - show 53
  - size 100
  - temporary (**ans**) 99
  - type 100
  - using wildcards with 97
  - viewing 53
- Variables window 54, 99
- vector 195
  - creating 72
  - expand with [] 73, 94
  - logspaced 73, 197
  - regular 73, 196

reversing 73  
transpose (') 73, 93  
void function  
calling 249  
declaration 248

## W

watchpoint 269  
**whatis** 105, 252  
**while** 104, 256, 257, 386  
**WHO** 325  
**who** 96  
wildcard 97  
    asterisk 97  
    colon 194  
    percent 97  
window 368  
    close 369  
    default window menu 368  
    frame 368  
    iconify 368, 369  
    lower 369  
    manager 366, 367  
    maximize 368, 369  
    minimize 368, 369  
    move 369, 370  
    raise 370  
    resize 369, 370  
    restore 369  
    Xmath 34  
        Commands 35  
        debugger 265  
        Graphics 123, 171  
        resizing 34  
        Variables 99  
working directory 108  
**WrapMatrix** 298  
**WrapPDM** 302  
**WrapString** 299  
**WrapStringMatrix** 299

## X

X Windows 366  
    starting 366  
**XMATH** 25, 373  
Xmath  
    abort (Ctrl-\, UNIX) 30  
    cut and paste 41  
    debugger, exiting 333, 334  
    default key bindings 41  
    editing text 40  
    file 374  
    interrupt  
        Ctrl-Break 30  
        Ctrl-c 30  
    Mathematica interface 403, 405  
    quitting 31  
    running across the network 28  
    starting 28  
        displaying to a local host 28  
        from a remote host (-host) 28  
        with UCI 294, 321  
    syntax differences from MATLAB 382  
    tty version 29  
**xmath** command  
    help  
        UNIX 29  
        Windows 30  
**XMATH\_PRINT** 26, 118, 119, 180  
**XMATH\_STARTUP** 25, 118, 119  
**XmathError** 304, 307  
**XmathExecute** 308  
**XmathGet** 304, 309  
**xmathlib.h** 297  
**XmathLNX.h** 304, 307  
**XmathLoad** 312  
**XmathMain** 293, 303, 304  
**XmathPanic** 335  
**XmathPut** 309  
**XmathSave** 311  
**XmathStart** 304, 314  
**XmathStop** 304, 314  
**xmodmap** 375

## **Z**

zero lines [148](#)

zeros [385](#)

