

MATLAB®

The Language of Technical Computing

MATLAB

The Language of Technical Computing Julio Salvador

Contents

Development Environment [Starting and Quitting MATLAB](#)

1

[Starting MATLAB](#) **1-3**

[Starting MATLAB on Windows Platforms](#)..... **1-3**

[Starting MATLAB on UNIX Platforms](#)..... **1-3**

[Startup Directory for MATLAB](#) **1-3**

[Startup Options](#) **1-5**

[Reducing Startup Time with Toolbox Path Caching](#)..... **1-10**

[Quitting MATLAB](#) **1-15** [Running a Script](#)

[When Quitting MATLAB](#)..... **1-15** [Using the Desktop](#)

2

[Desktop Tools](#)..... **2-4** [Launch](#)

[Pad](#)..... **2-5**

[Configuring the Desktop](#)..... **2-7**

[Opening and Closing Desktop Tools](#) **2-7**

[Resizing Windows](#) **2-9**

[Moving Windows](#)..... **2-10**

[Using Predefined Desktop Configurations](#)..... **2-16**

[Common Desktop Features](#) **2-17**

[Desktop Toolbar](#)..... **2-17**

[Context Menus](#)..... **2-17**

[Keyboard Shortcuts and Accelerators](#) **2-18** [Selecting Multiple Items](#)

..... **2-19** [Using the Clipboard](#)

..... **2-20** [Accessing The MathWorks on the Web](#)

..... **2-20**

[Setting Preferences](#) **2-21** [General Preferences](#)

[for MATLAB](#) **2-23** [Running MATLAB Functions](#)

3

<u>The Command Window</u>	3-2
<u>Opening the Command Window</u>	3-2
<u>Running Functions and Entering Variables</u>	3-2
<u>Controlling Input and Output</u>	3-4
<u>Running Programs</u>	3-11
<u>Keeping a Session Log</u>	3-12
<u>Preferences for the Command Window</u>	3-13
<u>Command History</u>	3-17
<u>Viewing Functions in the Command History Window</u>	3-17
<u>Running Functions from the Command History Window</u>	3-18
<u>Copying Functions from the Command History Window</u>	3-19
<u>Getting Help</u>	
4	
<u>Types of Information</u>	4-3
<u>Using the Help Browser</u>	4-4
Changing the Size of the Help Browser	4-5
<u>Using the Help Navigator</u>	4-7
Using the Product Filter	4-7
Viewing the Contents Listing in the Help Browser.....	4-9
Finding Documentation Using the Index.....	4-11
Searching Documentation	4-13
Bookmarking Favorite Pages.....	4-15
<u>Viewing Documentation in the Display Pane</u>	4-17
Browsing to Other Pages.....	4-18
Bookmarking Pages.....	4-18
Revisiting Pages.....	4-19
Printing Pages.....	4-19
Finding Terms in Displayed Pages	4-19
Copying Information	4-19
Evaluating a Selection.....	4-19
Viewing the Page Source (HTML).....	4-20
Viewing Web Pages	4-20
<u>Preferences for the Help Browser</u>	4-21
Documentation Location – Specifying the help Directory	4-22
Product Filter – Limiting the Product Documentation	4-23
PDF Reader – Specifying Its Location.....	4-23
General – Synchronizing the Contents Pane with the Displayed Page.....	4-23
Help Fonts Preferences – Specifying Font Name, Style, and Size	4-24

Printing Documentation.....	4-26
Printing a Page from the Help Browser	4-26
Printing the PDF Version of Documentation	4-26
Using Help Functions	4-28
Viewing Function	
Reference Pages – the doc Function.....	4-29
Getting Help in the Command Window – the help Function .	4-29
Other Methods for Getting Help.....	4-31
Product-Specific Help Features.....	4-31
Running Demos	4-31
Contacting Technical Support.....	4-32
Providing Feedback.....	4-32
Getting Version and License Information	4-32
Accessing Documentation for Other Products.....	4-33
Participating in the Newsgroup for MathWorks Products.....	4-33
Operations	
5	
MATLAB Workspace.....	5-3
Workspace Browser.....	5-3
Viewing and Editing Workspace Variables	
Using the Array Editor.....	5-10
Search Path	5-14
How the Search Path Works	5-14
Viewing and Setting the Search Path	5-15
File Operations	5-20
Current Directory Field.....	5-20
Current Directory Browser.....	5-20
Viewing and Making Changes to Directories	5-22
Creating, Renaming, Copying, and Removing Directories and Files	5-23
Opening, Running, and Viewing the Content of Files.....	5-26
Finding and Replacing Content Within Files.....	5-28
Preferences for the Current Directory Browser	5-30
Importing and Exporting Data	
6	
Importing Text Data.....	6-4
Using the Import Wizard with Text Data	6-4
Using Import Functions with Text Data	6-9

Importing Numeric Text Data.....	6-11
Importing Delimited ASCII Data Files	6-12
Importing Numeric Data with Text Headers	6-13
<u>Importing Mixed Alphabetic and Numeric Data.....</u>	<u>6-14</u>
<u>Exporting ASCII Data</u>	<u>6-16</u>
Exporting Delimited ASCII Data Files	6-17
Using the diary Command to Export Data.....	6-18
<u>Importing Binary Data.....</u>	<u>6-20</u>
Using the Import Wizard with Binary Data Files.....	6-20
Using Import Functions with Binary Data	6-22
<u>Exporting Binary Data</u>	<u>6-25</u>
Exporting MATLAB Graphs in AVI Format	6-27
Working w ith HDF Data	6-29
Overview of MATLAB HDF Support.....	6-30
MATLAB HDF Function Calling Conventions.....	6-31
Importing HDF Data into the MATLAB Workspace.....	6-33
Exporting MATLAB Data in an HDF File	6-41
Including Metadata in an HDF File.....	6-47
Using the MATLAB HDF Utility API	6-49
Using Lo w-Level File I/O Functions	6-51
Opening Files	6-52
Reading Binary Data.....	6-54
Writing Binary Data	6-55
Controlling Position in a File.....	6-56
Reading Strings Line-By-Line from Text Files	6-58
Reading Formatted ASCII Data.....	6-59
Writing Formatted Text Files	6-61
Closing a File.....	6-62

Editing and Debugging M-Files

7

Starting the Editor/Debugger	7-3
Creating a New M-File in the Editor/Debugger	7-4
Opening Existing M-Files in the Editor/Debugger.....	7-5
Opening the Editor Without Starting MATLAB.....	7-6
Closing the Editor/Debugger.....	7-6
Creating and Editing M-Files w ith the Editor/Debugger	7-8
Appearance of an M-File	7-8

Navigating in an M-File	7-11
Saving M-Files	7-15
Running M-Files from the Editor/Debugger.....	7-15
Printing an M-File.....	7-16
Closing M-Files	7-16
Debugging M-Files.....	7-17
Types of Errors.....	7-17
Finding Errors	7-17
Debugging Example – The Collatz Problem.....	7-18
Using Debugging Features.....	7-22
Preferences for the Editor/Debugger	7-35
General Preferences for the Editor/Debugger	7-36
Font & Colors Preferences for the Editor/Debugger.....	7-37
Display Preferences for the Editor/Debugger.....	7-38
Keyboard and Indenting Preferences for the Editor/Debugger	7-40
Printing Preferences for the Editor/Debugger.....	7-42

Improving M-File Performance– the Profiler

8

What Is Profiling?	8-3
Using the Profiler	8-4
The profile Function.....	8-4
An Example Using the Profiler	8-6
Viewing Profiler Results	8-7
Viewing Profile Reports.....	8-7
Profile Plot	8-12
Saving Profile Reports.....	8-13

Interfacing with Source Control Systems

9

Process of Interfacing to an SCS.....	9-3
Viewing or Selecting the Source Control System.....	9-4 Function Alternative for Viewing the SCS.....
Setting Up the Source Control System.....	9-6
For SourceSafe Only – Mirroring MATLAB Hierarchy.....	9-6
For ClearCase on UNIX Only – Set a View and Check Out a Directory.....	9-6

Checking Files into the Source Control System..... 9-7 Function Alternative for Checking In Files..... 9-8

Checking Files Out of the SCS..... 9-9

Function Alternative for Checking Out Files..... 9-10

Undoing the Check-Out..... 9-10

Using Notebook

10

Notebook Basics 10-3

Creating an M-Book 10-3

Entering MATLAB Commands in an M-Book..... 10-6

Protecting the Integrity of Your Workspace..... 10-6

Ensuring Data Consistency..... 10-7

Defining MATLAB Commands as Input Cells 10-8 Defining Cell

Groups..... 10-8 Defining Autoinit Input

Cells..... 10-10 Defining Calc Zones

..... 10-10 Converting an Input Cell to Text

..... 10-11

Evaluating MATLAB Commands 10-12 Evaluating Cell

Groups..... 10-13

Evaluating a Range of Input Cells 10-14

Evaluating a Calc Zone..... 10-14

Evaluating an Entire M-Book 10-15

Using a Loop to Evaluate Input Cells Repeatedly..... 10-15

Converting Output Cells to Text 10-16

Deleting Output Cells 10-17

Printing and Formatting an M-Book..... 10-18

Printing an M-Book..... 10-18

Modifying Styles in the M-Book Template..... 10-18

Choosing Loose or Compact Format..... 10-19

Controlling Numeric Output Format 10-20

Controlling Graphic Output..... 10-20

Configuring Notebook 10-24

Notebook Command Reference 10-26

Bring MATLAB to Front Command..... 10-26

Define Autoinit Cell Command 10-26

Define Calc Zone Command 10-27

Define Input Cell Command 10-27

Evaluate Calc Zone Command	10-28
Evaluate Cell Command	10-28
Evaluate Loop Command.....	10-29
Evaluate M-Book Command	10-30
Group Cells Command	10-30
Hide Cell Markers Command.....	10-31
Notebook Options Command.....	10-31
Purge Output Cells Command	10-31
Toggle Graph Output for Cell Command	10-31
Undefine Cells Command	10-32
Ungroup Cells Command	10-32

Mathematics Matrices and Linear Algebra

11

Function Summary	11-3
Matrices in MATLAB	11-5
Creation	11-5
Addition and Subtraction	11-7
Vector Productsand Transpose.....	11-7
Matrix Multiplication.....	11-9
The Identity Matrix.....	11-11
The Kronecker Tensor Product	11-11
Vector and Matrix Norms	11-12
Solving Linear Systems of Equations	11-13
Overview	11-13
Square Systems.....	11-15
Overdetermined Systems	11-15
Underdetermined Systems.....	11-18
Inverses and Determinants	11-21
Overview	11-21
Pseudoinverses.....	11-22
Cholesky, LU, and QR Factorizations.....	11-25
Cholesky Factorization.....	11-25
LU Factorization	11-26
QR Factorization	11-28
Matrix P o w ers and Exponentials	11-32
Eigenvalues	11-35
Singular Value Decomposition	11-39
Polynomials and Interpolation	

12

P olynomials.....	12-3
Polynomial Function Summary.....	12-3
Representing Polynomials.....	12-4
Polynomial Roots.....	12-4
Characteristic Polynomials	12-5
Polynomial Evaluation.....	12-5
Convolution and Deconvolution.....	12-6
Polynomial Derivatives	12-6
Polynomial Curve Fitting.....	12-7
Partial Fraction Expansion.....	12-8
 I nterpolation	 12-10
Interpolation Function Summary	12-10
One-Dimensional Interpolation.....	12-11
Two-Dimensional Interpolation.....	12-13
Comparing Interpolation Methods.....	12-14
Interpolation and Multidimensional Arrays	12-16
Triangulation and Interpolation of Scattered Data.....	12-19
Tessellation and Interpolation of Scattered Data in Higher Dimensions.....	12-27
 S elected Bibliography	 12-38 D ata A nalysis a nd S tatistics

13

C olumn-Oriented Data Sets.....	13-4
 B asic Data Analysis Functions.....	 13-8 F unction S ummary
.....	13-8 Covariance and Correlation
Coefficients.....	13-11 Finite
Differences.....	13-12
 D ata P reprocessing.....	 13-14
Missing Values.....	13-14
Removing Outliers.....	13-15
 R egression and Curve Fitting	 13-17
Polynomial Regression	13-18
Linear-in-the-Parameters Regression.....	13-19
Multiple Regression	13-21
 C ase S tudy: C urve F itting	 13-22
Polynomial Fit.....	13-22
Analyzing Residuals.....	13-24

Exponential Fit.....	13-27
Error Bounds.....	13-30
The Basic Fitting Interface	13-31
Difference Equations and Filtering	13-40

Fourier Analysis and the Fast Fourier Transform (FFT) 13-43	
Function Summary	13-43
Introduction.....	13-44
Magnitude and Phase of Transformed Data	13-49
FFT Length Versus Speed.....	13-50

Function Functions

14

Function Summary	14-3
Representing Functions in MATLAB	14-4
Plotting Mathematical Functions	14-6
Minimizing Functions and Finding Zeros	14-9
Minimizing Functions of One Variable	14-9
Minimizing Functions of Several Variables	14-10
Setting Minimization Options.....	14-11
Finding Zeros of Functions.....	14-12
Tips	14-15
Troubleshooting.....	14-15
Converting Your Optimization Code to MATLAB Version 5 Syntax.....	14-16
Numerical Integration (Quadrature).....	14-19
Example: Computing the Length of a Curve.....	14-19
Example: Double Integration	14-20

Differential Equations

15

Initial Value Problems for ODEs and DAEs	15-3
ODE Function Summary.....	15-3
Introduction to Initial Value ODE Problems.....	15-6
Initial Value Problem Solvers	15-6
Representing ODE Problems	15-10
Improving ODE Solver Performance.....	15-16
Examples: Applying the ODE Initial Value Problem Solvers	15-31
Questions and Answers, and Troubleshooting	15-50

Boundary Value Problems for ODEs	15-57
BVP Function Summary	15-57
Introduction to Boundary Value ODE Problems.....	15-59
Boundary Value Problem Solver	15-60
Representing BVP Problems.....	15-63
Using Continuation to Make a Good Initial Guess	15-68
Improving BVP Solver Performance	15-75
P artial Differential Equations	15-82
PDE Function Summary	15-82
Introduction to PDE Problems	15-83
MATLAB Partial Differential Equation Solver.....	15-84
Representing PDE Problems.....	15-88
Improving PDE Solver Performance	15-93
Example: Electrodynamics Problem.....	15-94
Selected Bibliography	15-99
Sparse Matrices	

16

Function Summary	16-3
Introduction	16-6
Sparse Matrix Storage	16-6
General Storage Information	16-7
Creating Sparse Matrices.....	16-7
Importing Sparse Matrices from Outside MATLAB	16-12
Viewing Sparse Matrices.....	16-13
Information About Nonzero Elements	16-13
Viewing Sparse Matrices Graphically.....	16-15
The find Function and Sparse Matrices.....	16-16
Example: Adjacency Matrices and Graphs	16-17
Introduction to Adjacency Matrices	16-17
Graphing Using Adjacency Matrices.....	16-18
The Bucky Ball.....	16-18
An Airflow Model	16-23
Sparse Matrix Operations	16-25
Computational Considerations.....	16-25
Standard Mathematical Operations.....	16-25
Permutation and Reordering.....	16-26
Factorization	16-30
Simultaneous Linear Equations.....	16-36
Eigenvalues and Singular Values	16-39

Selected Bibliography **16-42**
Programming and Data Types M-File Programming

17

MATLAB Programming: A Quick Start.....	17-3
Kinds of M-Files.....	17-3
What's in an M-File?	17-4
Providing Help for Your Programs.....	17-4
Creating M-Files: Accessing Text Editors.....	17-5
Scripts	17-7 Simple Script Example.....
	17-7
Functions	17-8
Simple Function Example.....	17-8
Basic Parts of a Function M-File.....	17-9
Function Names.....	17-11
How Functions Work.....	17-12
Checking the Number of Function Arguments.....	17-14
Passing Variable Numbers of Arguments	17-16
Local and Global Variables.....	17-19
Persistent Variables.....	17-20
Special Values.....	17-21
Data Types	17-22
Key words.....	17-25
Operators.....	17-26
Arithmetic Operators	17-26
Relational Operators	17-28
Logical Operators.....	17-29
Operator Precedence	17-32
Flow Control.....	17-35
if, else, and elseif	17-35
switch	17-37
while.....	17-39
for.....	17-40
continue.....	17-41
break	17-41
try ... catch	17-42
return	17-42
Subfunctions	17-43
Private Functions.....	17-45

Subscripting and Indexing	17-46
Subscripting	17-46
Advanced Indexing.....	17-51
String Evaluation	17-54
eval.....	17-54
feval.....	17-54
Command/Function Duality.....	17-56
Empty Matrices.....	17-57
Operating on an Empty Matrix.....	17-57
Using Empty Matrices with If or While	17-58
Errors and Warnings.....	17-59
Error Handling with eval and lasterr	17-59
Displaying Error and Warning Messages	17-60
Dates and Times	17-62
Date Formats	17-62
Current Date and Time	17-67
Obtaining User Input	17-69
Prompting for Keyboard Input.....	17-69
Pausing During Execution	17-69
Shell Escape Functions	17-70
Optimizing MATLAB Code	17-71
Vectorizing Loops	17-71
Preallocating Arrays	17-73
Making Efficient Use of Memory.....	17-74

Character Arrays (Strings)

18

Character Arrays	18-5
Creating Character Arrays	18-5
Creating Two-Dimensional Character Arrays.....	18-6
Converting Characters to Numeric Values	18-7
Cell Arrays of Strings	18-8 Converting to a Cell Array of Strings.....
String Comparisons	18-10
Comparing Strings For Equality.....	18-10
Comparing for Equality Using Operators	18-11

Categorizing Characters Within a String	18-12	
Searching and Replacing	18-13	
String/Numeric Conversion	18-15	Array/String Conversion
	18-16	Multidimensional Arrays

19

Multidimensional Arrays	19-3	
Creating Multidimensional Arrays	19-4	
Accessing Multidimensional Array Properties.....	19-8	
Indexing	19-9	
Reshaping.....	19-10	
Permuting Array Dimensions	19-12	
Computing w ith Multidimensional Arrays	19-14	
Operating on Vectors.....	19-14	
Operating Element-by-Element.....	19-14	
Operating on Planes and Matrices.....	19-15	
Organizing Data in Multidimensional Arrays	19-16	
Multidimensional Cell Arrays.....	19-18	Multidimensional Structure Arrays.....
	19-19	Applying Functions to Multidimensional Structure Arrays .
	19-20	Structures and Cell Arrays

20

Structures	20-4
Building Structure Arrays.....	20-5
Accessing Data in Structure Arrays.....	20-7
Finding the size of Structure Arrays.....	20-10
Adding Fields to Structures	20-10
Deleting Fields from Structures.....	20-10
Applying Functions and Operators	20-10
Writing Functions to Operate on Structures.....	20-11
Organizing Data in Structure Arrays	20-12
Nesting Structures.....	20-17
Cell Arrays.....	20-19
Creating Cell Arrays	20-20
Obtaining Data from Cell Arrays.....	20-23
Deleting Cells.....	20-24
Reshaping Cell Arrays.....	20-25
Replacing Lists of Variables with Cell Arrays	20-25
Applying Functions and Operators	20-27

Organizing Data in Cell Arrays	20-27
Nesting Cell Arrays.....	20-29
Converting Between Cell and Numeric Arrays.....	20-30
Cell Arrays of Structures.....	20-31

Function Handles

21

Benefits of Using Function Handles	21-3
A Simple Function Handle.....	21-5
Constructing a Function Handle	21-7
Maximum Length of a Function Name	21-7
Evaluating a Function Through Its Handle	21-9
Function Evaluation and Overloading	21-9
Examples of Function Handle Evaluation.....	21-10
Displaying Function Handle Information	21-13
Fields Returned by the Functions Command.....	21-14
Types of Function Handles.....	21-17
Function Handle Operations.....	21-21
Converting Function Handles to Function Names.....	21-21
Converting Function Names to Function Handles.....	21-22
Testing for Data Type.....	21-23
Testing for Equality	21-23
Saving and Loading Function Handles	21-25
Handling Error Conditions.....	21-26
Handles to Nonexistent Functions.....	21-26
Including Path In the Function Handle Constructor	21-26
Evaluating a Nonscalar Function Handle.....	21-27
Historical Note - Evaluating Function Names	21-28
MATLAB Classes and Objects	

22

Classes and Objects: An Overview	22-3
Features of Object-Oriented Programming	22-3
MATLAB Data Class Hierarchy.....	22-4
Creating Objects.....	22-4
Invoking Methods on Objects	22-5
Private Methods.....	22-6
Helper Functions.....	22-6

Debugging Class Methods.....	22-6
Setting Up Class Directories	22-7
Data Structure	22-8
Tips for C++ and Java Programmers	22-8
Designing User Classes in MATLAB	22-9
Canonical Class	22-9
Method.....	22-10
Examples of Constructor Methods	22-11
Identifying Objects Outside the Class	22-11
Directory.....	22-11
The display Method.....	22-12
Accessing Object Data	22-13
The set and get Methods	22-13
Indexed Reference Using subsref and subsasgn.....	22-14
Handling Subscripted Reference.....	22-14
Handling Subscripted Assignment.....	22-17
Object Indexing Within Methods	22-17
Defining end Indexing for an Object	22-18
Indexing an Object with Another Object.....	22-19
Converter Methods.....	22-20
Overloading Operators and Functions	22-21
Overloading Operators.....	22-21
Overloading Functions	22-23
Example - A P olynomial Class	22-24
Polynom Data Structure	22-24
Polynom Methods.....	22-24
The Polynom Constructor Method	22-24
Converter Methods for the Polynom Class.....	22-25
The Polynom display Method	22-28
The Polynom subsref Method.....	22-28
Overloading Arithmetic Operators for polynom.....	22-29
Overloading Functions for the Polynom Class	22-31
Listing Class Methods.....	22-33
Building on Other Classes	22-35
Simple Inheritance.....	22-35
Multiple Inheritance	22-36
Aggregation.....	22-37
Example - Assets and Asset Subclasses.....	22-38
Inheritance Model for the Asset Class	22-38
Asset Class Design	22-39
Other Asset Methods	22-39
The Asset Constructor Method.....	22-39
The Asset get Method.....	22-41
The Asset set Method.....	22-41

The Asset subsref Method	22-42
The Asset subsasgn Method.....	22-43
The Asset display Method	22-44
The Asset fieldcount Method.....	22-45
Designing the Stock Class.....	22-45
The Stock Constructor Method.....	22-46
The Stock get Method.....	22-48
The Stock set Method.....	22-48
The Stock subsref Method.....	22-49
The Stock subsasgn Method.....	22-51
The Stock display Method	22-52
 Example - The Portfolio Container.....	22-54
Designing the Portfolio Class.....	22-54
The Portfolio Constructor Method.....	22-55
The Portfolio display Method	22-56
The Portfolio pie3 Method.....	22-57
Creating a Portfolio	22-58
 Saving and Loading Objects	22-60 Modifying Objects
During Save or Load.....	22-60
 Example - Defining saveobj and loadobj for P ortfolio . .	22-61 Summary of Code Changes
.....22-61 The saveobj Method	
.....22-62 The loadobj	
Method.....	22-62 Changing the Portfolio Constructor
.....22-63 The Portfolio subsref	
Method.....	22-64
 Object Precedence	22-65 Specifying
Precedence of User-Defined Classes	22-66
 Ho w MATLAB Determines Which Method to Call	22-67
Selecting a Method.....	22-67
Querying Which Method MATLAB Will Call.....	22-70
 External Interfaces and the MATLAB API	
 A	
 Finding the Documentation in Online Help	A-2
Reference Documentation	A-4 xxii <i>Contents</i>
 1	
Starting and Quitting MATLAB	
 Starting MATLAB	1-3

Starting MATLAB on Windows Platforms.....	1-3
Starting MATLAB on UNIX Platforms.....	1-3
Startup Directory for MATLAB	1-3
Startup Options.....	1-5
Reducing Startup Time with Toolbox Path Caching	1-10
Quitting MATLAB.....	1-15
Running a Script When Quitting MATLAB.....	1-15

These sections describe how to start and quit MATLAB, including options associated with starting and quitting.

- “Starting MATLAB on Windows Platforms” on page 1-3
- “Starting MATLAB on UNIX Platforms” on page 1-3
- “Startup Directory for MATLAB” on page 1-3
- “Startup Options” on page 1-5
- “Reducing Startup Time with Toolbox Path Caching” on page 1-10
- “Quitting MATLAB” on page 1-15
- “Running a Script When Quitting MATLAB” on page 1-15

Starting MATLAB

Instructions for starting MATLAB depend on your platform. For a list of supported platforms, see the system requirements in the *Installation Guide* for your platform or the Products section of the MathWorks Web site, <http://www.mathworks.com>.

Starting MATLAB on Windows Platforms

To start MATLAB on a Microsoft Windows platform, double-click the MATLAB shortcut icon  on your Windows desktop. The shortcut was automatically

created by the installer in the installation directory.

If you start MATLAB from a DOS window, type `matlab` at the DOS prompt.

After starting MATLAB, the MATLAB desktop opens – see Chapter 2, “Using the Desktop.”

If you use a virus scanner, your settings may slow down MATLAB startup. For example, if you use McAfee VirusScan, set the options to scan program files only.

Starting MATLAB on UNIX Platforms

To start MATLAB on a UNIX platform, type `matlab` at the operating system prompt.

After starting MATLAB, the MATLAB desktop opens – see Chapter 2, “Using the Desktop.” On UNIX platforms, if the `DISPLAY` environment variable is not set or is invalid, the desktop will not display. On some UNIX platforms, the desktop is not supported – see the *R 12 Release Notes* for details.

Startup Directory for MATLAB

The initial current directory in MATLAB depends on your platform and installation. You can start MATLAB in a different directory.

Startup Directory on Windows Platforms

On Windows platforms, when you installed MATLAB, the default startup directory was set to \$matlabroot\work, where \$matlabroot is the directory where MATLAB files are installed.

Startup Directory on UNIX Platforms

On UNIX platforms, the initial current directory is the directory you are in on your UNIX file system when you invoke MATLAB.

Changing the Startup Directory

You can start MATLAB in a different directory from the default. The directory you specify will be the current working directory when MATLAB starts. To do so:

1 Create a startup.m file – see “Using the Startup File for MATLAB, startup.m” on page 1-5.

2 In the startup.m file, include the cd function to change to the new directory.

3 Put the startup.m file in the current startup directory.

For Windows Platforms Only. For Windows platforms, you can follow these steps to change the startup directory:

1 Right-click on the MATLAB shortcut icon and select **Properties** from the context menu.

The **Properties** dialog box for matlab.exe opens to the **Shortcut** page.

2 Enter the new startup directory in the **Start in** field and click **OK**.

The next time you start MATLAB using that shortcut icon, the current directory will be the one you specified in step 2.

You can make multiple shortcuts to start MATLAB, each with its own startup directory, and each startup directory having different startup options.

Startup Options

You can define startup options for MATLAB, which instruct MATLAB to perform certain operations upon startup. There are two ways to specify startup options for MATLAB:

- “Using the Startup File for MATLAB, startup.m” on page 1-5

- “Adding Startup Options for Windows Platforms” on page 1-5 or “Adding Startup Options for UNIX Platforms” on page 1-7

Using the Startup File for MATLAB, startup.m

At startup, MATLAB automatically executes the master M-file matlabrc.m and, if it exists, startup.m. The file matlabrc.m, which lives in the local

directory, is reserved for use by The MathWorks, and on multiuser systems, by your system manager.

The file startup.m is for you to use to specify startup options. You can modify the default search path, predefine variables in your workspace, or define Handle Graphics[®] defaults. For example, creating a startup.m file with the line

```
addpath \home\me\mytools
```

```
cd \home\me\mytools
```

adds \home\me\mytools to your default search path and makes that directory the current directory

upon startup.

On Windows platforms, place the startup.m file in
\$matlabroot\toolbox\local, where \$matlabroot is the directory in which MATLAB is installed.

On UNIX workstations, place the startup.m file in the directory named matlab off of your home directory, for example, ~/matlab.

Adding Startup Options for Windows Platforms

You can add selected startup options (also called command flags) to the target path for your Windows shortcut for MATLAB, or include the option if you start MATLAB from a DOS window. To do so:

1 Right-click on the MATLAB shortcut icon

and select **Properties** from the context menu.

The **Properties** dialog box for matlab.exe opens to the **Shortcut** panel.

2 In the **Target** field, after the target path matlab.exe, add one or more of the allowable startup options listed here.

Option Description

/automation Start MATLAB as an automation server,

minimized and without the MATLAB splash screen. For more information, see “Client/Server Applications” in the *External Interfaces Guide*.

/c licensefile

/logfile logfilename

/minimize

/nosplash

/r M_file

/regserver

/unregserver SetLM_LICENSE_FILE to licensefile. It can have the form port@host.

Automatically write output from MATLAB to the specified log file.

Start MATLAB minimized and without the MATLAB splash screen.

Start MATLAB without displaying the MATLAB splash screen.

Automatically run the specified M-file

immediately after MATLAB starts. This is also referred to as calling MATLAB in batch mode.

Modify the Windows registry with the

appropriate ActiveX entries for MATLAB. For more information, see “Client/Server Applications” in the *External Interfaces Guide*.

Modify the Windows registry to remove the ActiveX entries for MATLAB. Use this option to reset the registry. For more information, see “Client/Server Applications” in the *External Interfaces Guide*.

3 Click OK.

Example –Setting the Startup Options to Automatically Run an M-File. To start MATLAB and automatically run the filerresults.m, use this target path for your Windows shortcut.

D:\matlabr12\bin\win32\matlab.exe /r results

Startup Options If You Run MATLAB from a DOS Window. If you run MATLAB from a DOS window, include the startup options listed in the preceding table after the matlab startup function.

For example, to start MATLAB and automatically run the filerresults.m , type matlab /r results

Adding Startup Options for UNIX Platforms

Include startup options (also called command flags) after thematlab startup function. The startup options for UNIX are listed in the following table

Option Description

-arch
-arch/ext
-c licensefile
-Ddebugger [options]
-debug
-display Xserver
-ext
-h or-help

Run MATLAB assuming architecturearch.

Run the version of MATLAB with the extensionext if it exists, assuming architecture arch.

SetLM_LICENSE_FILE tolicensefile. It can have the formport@host.

Start MATLAB with the specified debugger. Turn on MATLAB internal debugging. Send X commands toXserver.

Run the version of MATLAB with the extensionext, if it exists.

Displays startup options.

Option Description (Continued)

-mwvisual visualid

-n

-nodesktop

Specify the default X visual to use for figure windows.

Print environment variables only.

Start MATLAB without bringing up the MATLAB desktop. Use this option to run without an X-window, for example, in VT100 mode, or in batch processing mode. Note that if you pipe to MATLAB using the> constructor, the nodesktop option is used automatically.

With nodesktop, you can still use most development environment tools by starting them with a

function. Specifically use:

- edit to open the Editor/Debugger
- helpbrowser to open the Help browser
- filebrowser to open the Current Directory browser
- workspace to open the Workspace browser
- openvar to open the Array Editor

You cannot use the LaunchPad and the Command History window in-nodesktop mode.

Don't use nodesktop to provide a command line interface, if you prefer that over the desktop tools. Instead, select **Vie w -> Desktop Layout -> Command Windo w Only**.

Option Description (Continued)

-nojvm Start MATLAB without loading the Java VM. This minimizes memory usage and improves initial startup speed. Withnojvm, you cannot use the desktop, nor any of the tools that require Java. The restrictions are the same as those described under UNIX Platform Limitations in the *R 12 Release Notes*.

-nosplash Start MATLAB without displaying the splash screen during startup.

For example, to start MATLAB without the splash screen, type matlab -nosplash

Reducing Startup Time w ith Toolbox Path Caching

If you run MATLAB from a network server, you can significantly reduce your startup time by using the MATLAB toolbox path cache. The toolbox path cache stores search path information on all toolbox directories under the MATLAB root directory. During startup, MATLAB obtains this information from the cache rather than by reading it from the remote file system.

The toolbox path cache is used only during the startup of your MATLAB session. It is especially useful if you define your MATLAB search path to include many toolbox directories. It takes considerable time to acquire all of this information by scanning directories in the remote file system. Reading it from a pre-generated cache however, is significantly faster. If you have a short toolbox path, there is less benefit to using the cache, but it does still provide a time savings.

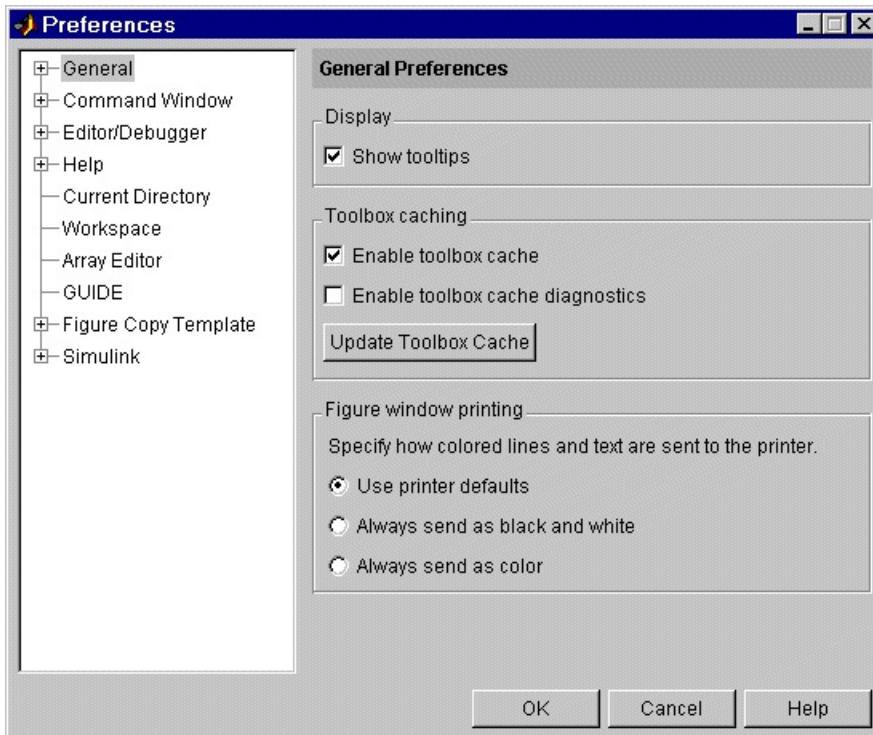
If you run MATLAB on a local disk, where your files are not served from a remote system, then the cache may provide no noticeable reduction in startup time. In this case, you may want to leave the toolbox path cache disabled.

Selecting Toolbox Path Cache Preferences

To generate and enable or disable the toolbox path cache, use MATLAB preferences.

1 From the MATLAB desktop, select **Preferences** from the **File** menu.

The **Preferences** dialog box appears. **2** In the left pane of the dialog box, click **General** to display the **General Preferences** panel in the right pane.



3 To use the cache or to regenerate the cache, select **Enable toolbox cache** under **Toolbox caching**.

With **Enable toolbox cache** selected, MATLAB displays summary information during startup, sends notification when it loads the toolbox directories from the cache, and displays a warning if the toolbox path cache cannot be found.

4 For additional information, select **Enable toolbox cache diagnostics**.

With **Enable toolbox cache diagnostics** selected, MATLAB displays additional information at startup.

MATLAB also provides warnings whenever a toolbox directory is added to the path from the remote file system at startup rather than from the cache. This occurs if the cache has not been kept up to date with changes in the toolbox directories.

5 To generate a new copy of the cache, select **Update Toolbox Cache**. See “Generating the Toolbox Path Cache” on page 1-12 for more information.

To use this, you need write access to the directory that holds the cache file. The button will be grayed out if you don’t have write access.

6 Select **OK**.

Generating the Toolbox Path Cache

The toolbox path cache is in a MAT-file in the toolbox/local directory on the system that serves files for MATLAB. When you first install MATLAB on this system, you or your system administrator needs to generate the cache. (MATLAB does not ship with a prebuilt cache). You also need to regenerate the cache file whenever toolbox directories are added or removed so that the cache does not hold out-dated path information. To do so, select **Update Toolbox Cache** as described in “Selecting Toolbox Path Cache Preferences” on page 1-10.

Function Equivalent. To update the cache, type the following in the Command Window.
rehash toolboxcache

Enabling Use of the Cache

When MATLAB is first installed, the toolbox path cache feature is disabled. In order to make use of the cache on a distributed file system:

- 1 The system administrator enables caching and generates the initial toolbox path cache on the system that serves files for MATLAB. This is explained in “Generating the Toolbox Path Cache” on page 1-12.
- 2 Each user that intends to use toolbox directory caching enables the cache on their own system. On a nondistributed system, the system user performs both steps. To enable toolbox path caching, follow the instructions under “Selecting Toolbox Path Cache Preferences” on page 1-10.

You can disable the use of this feature either on a user-by-user basis or on a global basis. To disable it on a per-user basis, uncheck **Enable toolbox cache** in the **Preferences** dialog box. To disable it on a global basis, remove the file toolbox_cache.mat from the toolbox/local directory.

Updating the Cache

If you make changes to your toolbox directories, your toolbox path cache file can become out of date. Depending upon the reason for this, you may or may not receive a warning that your cache needs to be updated. This section explains when you need to update your toolbox path cache and how to avoid problems caused by an out-dated cache file.

Updating Cache Following a Product Install or Update. If you install a new toolbox or an update from The MathWorks, it is likely that the information stored in the toolbox path cache no longer accurately reflects your toolbox directories. As part of the installation process, the MATLAB installer marks the cache as being invalid.

When your cache is marked invalid, MATLAB ceases to use cache during startup, loading path information by accessing the directories through the file system instead. MATLAB issues a message at startup to warn you that your toolbox path cache is out-dated and is being ignored. To resume use of the cache, you need to regenerate it using the procedure described in “Generating the Toolbox Path Cache” on page 1-12.

Updating Cache Following Changes to Toolbox Directories. If you make changes to the MATLAB toolbox directories (for example, by adding or deleting files), the path information in the toolbox path cache file becomes out of date. When you start up your next MATLAB session, this out-dated path information will be loaded into memory from the cache. MATLAB does not issue a warning when this occurs.

If you choose to make changes that affect the toolbox path, you must regenerate the cache file using the procedure described in “Generating the Toolbox Path Cache” on page 1-12.

Caution The MathWorks strongly recommends that you do not do development work in the toolbox area when toolbox path caching is enabled. If do you use the toolbox area for this purpose and neglect to regenerate the cache afterwards, MATLAB will use an inaccurate record

of your toolbox directories in subsequent MATLAB sessions. As a result, MATLAB will be unable to locate new files that you have added.

Quitting MATLAB

To quit MATLAB at any time, do one of the following:

- Select the close box  in the MATLAB desktop.
- Select **Exit MATLAB** from the desktop **File** menu.
- Typequit at the Command Window prompt.

MATLAB closes immediately, without issuing a warning. If you want to see a warning, use the finishdlg.m script as described in the next paragraph.

Running a Script When Quitting MATLAB

When MATLAB quits, it runs the script finish.m, if finish.m exists in the current directory or anywhere on the MATLAB search path. You create the file finish.m. It contains functions to run when MATLAB terminates, such as saving the workspace or displaying a confirmation dialog box. There are two sample files in \$matlabroot\toolbox\local that you can use as the basis for your own finish.m file:

- finishsav.m— Includes a save function so the workspace is saved to a MAT-file when MATLAB quits.
- finishdlg.m— Displays a confirmation dialog box that allows you to cancel quitting.

2

Using the Desktop

Desktop Tools	2-4	Launch Pad
.....	2-5	

Configuring the Desktop 2-7

Opening and Closing Desktop Tools.....	2-7
Resizing Windows	2-9
Moving Windows.....	2-10
Using Predefined Desktop Configurations	2-16

Common Desktop Features..... 2-17

Desktop Toolbar	2-17
Context Menus	2-17
Keyboard Shortcuts and Accelerators	2-18
Selecting Multiple Items	2-19
Using the Clipboard	2-20
Accessing The MathWorks on the Web	2-20

Setting Preferences..... 2-21

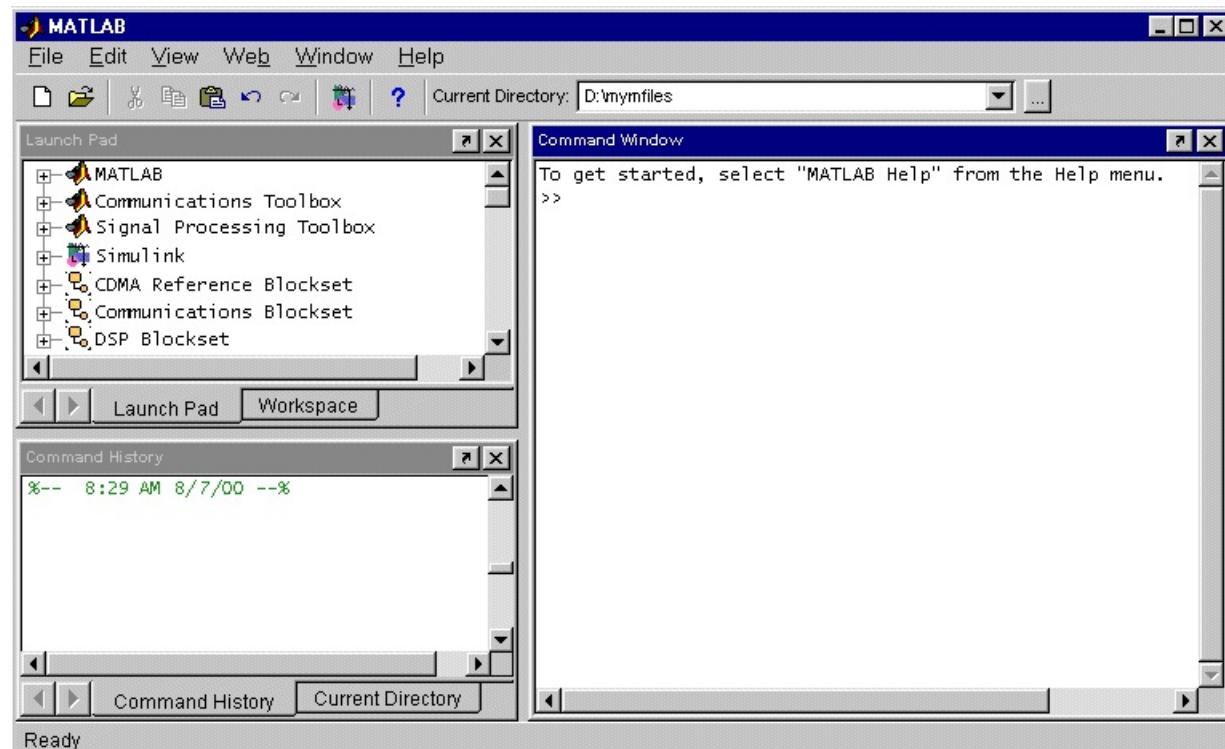
General Preferences for MATLAB	2-23
When you start MATLAB, the MATLAB desktop appears, containing tools (graphical user	

interfaces) for managing files, variables, and applications associated with MATLAB. Think of the desktop as your instrument panel for MATLAB. The main things you need to know about the desktop are:

- “Desktop Tools” on page 2-4 – All of the tools managed by the desktop.
- “Configuring the Desktop” on page 2-7 – Arranging the tools in the desktop.
- “Common Desktop Features” on page 2-17 – Features you can use in the tools, such as context menus.

The first time MATLAB starts, the desktop appears as shown in the following illustration, although your Launch Pad may contain different entries.

Expand to view Get help Enter documentation, demos, MATLAB and tools for your products functions View or change current directory
Click to move window Close window outside of desktop



View or use previously run functions Use tabs to go to Workspace browser
to resize windows or Current Directory browser Drag the separator bar

Desktop Tools

The following tools are managed by the MATLAB desktop, although not all of them appear by default when you first start. If you prefer a command line interface, you can use functions to perform most of the features found in the MATLAB desktop tools. Instructions for using these function equivalents are provided with the documentation for each tool.

- Command Window – Run MATLAB functions.

- Command History – View a log of the functions you entered in the Command Window, copy them, and execute them.
- Launch Pad – Run tools and access documentation for all of your MathWorks products.
- Current Directory Browser – View MATLAB files and related files, and perform file operations such as open, and find content.
- Help Browser – View and search the documentation for the full family of MATLAB products.
- Workspace Browser – View and make changes to the contents of the workspace.
- Array Editor – View array contents in a table format and edit the values.
- Editor/Debugger – Create, edit, and debug M-files (files containing MATLAB functions).

Other MATLAB tools and windows, such as figure windows, are not managed by the desktop.

Desktop Tools

Launch Pad

MATLAB's Launch Pad provides easy access to tools, demos, and documentation for all of your MathWorks products. To open it, select **Launch Pad** from the **View** menu in the MATLAB desktop. All the products installed on your system are listed.

Sample of listings in Launch Pad – you'll see listings for all products installed on your system.

Help – double-click to go directly to documentation for the product.

Demos – double-click to display the demo launcher for the product.

Tools – double-click to open the tool.

Product Page – double-click to view the latest product information on the MathWorks Website.

Click + to show the listing for a product.



To display the listings for a product, click the + to the left of the product. To collapse the listings, click the - to the left of the product.

To open one of the listings, double-click it, or right-click and select **Open** from the context menu. The action depends on the listing you selected, as described in the following table.

Icon Description of Action When Opened

 Documentation roadmap page for that product opens in the Help browser.

Demo launcher opens, with the demo for that product selected.



Selected tool opens.

 Product Page, which contains the latest product information on the MathWorks Web site, opens in your Web browser.

Refreshing the Launch Pad

The Launch Pad includes entries for all products found on the MATLAB search path when the MATLAB session was started. If you change the search path after the start of a session, such as by adding a toolbox directory, the Launch Pad is not automatically updated. Right-click in the Launch Pad and select **Refresh** from the context menu to update the Launch Pad so it reflects all products on the current search path.

Adding Your Own Entries to the Launch Pad

You can add your own entries to the Launch Pad by creating aninfo.xml file. To see an example, select one of the existing entries in the Launch Pad,

right-click, and select **Edit Source** from the context menu. The info.xml file for that product appears. The line for the tool you selected appears highlighted.

Create a similar info.xml file for your own application and put it in a folder that is on the search path. Right-click in the Launch Pad and select **Refresh** from the context menu to update the Launch Pad so it includes your entries.

Configuring the Desktop

You can modify the desktop configuration to best meet your needs. Close tools you don't use, open those you do, resize, and reposition them. Configure the MATLAB desktop by:

- “Opening and Closing Desktop Tools” on page 2-7
- “Resizing Windows” on page 2-9
- “Moving Windows” on page 2-10
- “Using Predefined Desktop Configurations” on page 2-16

When you end a session, MATLAB saves its desktop configuration. The next time you start MATLAB, the desktop is restored the way you left it.

Opening and Closing Desktop Tools

As part of configuring the MATLAB desktop so that it best meets your needs, you can use the following features:

- “Opening Desktop Tools” on page 2-7 – Open only those tools you use.
- “Going to Documents in Desktop Tools” on page 2-8 – Go directly to opened M-files, figures, and more.
- “Closing Desktop Tools” on page 2-9 – Close those tools you don’t use.

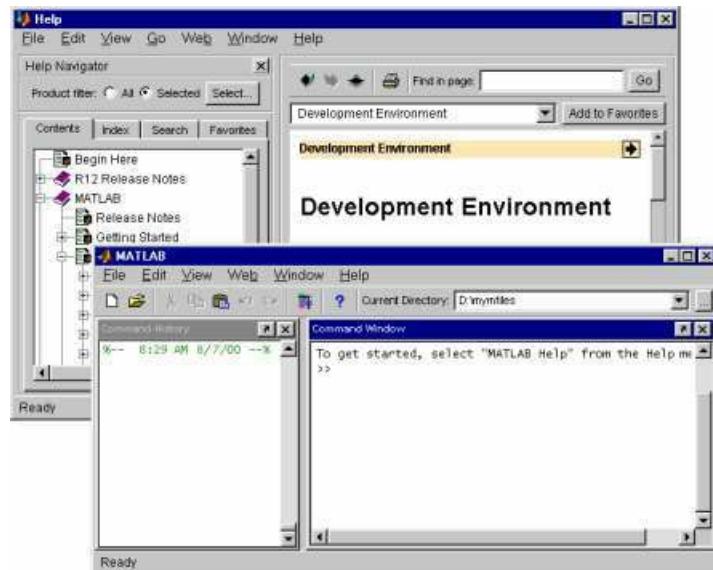
Opening Desktop Tools

To open a tool from the desktop, select the tool from the **Vie w** menu or double-click it in the list of tools displayed in the Launch Pad for MATLAB. The tool opens in the location it occupied the last time it was open.

There are a few tools controlled by the desktop that you don't open from the **Vie w** menu or Launch Pad:

- Array Editor – Open it by double-clicking a variable in the Workspace Browser.
- Editor/Debugger – Open it by creating a new M-file or opening an existing M-file. For instructions, see “Starting the Editor/Debugger” on page 7-3.

Another way to open a tool is using a function. For example, helpbrowser opens the Help browser. These functions are documented with each tool. The following example shows how the MATLAB desktop might look with the Command Window, Command History, and Help browser open.



Going to Documents in Desktop Tools

The **Windo w** menu displays all open Editor/Debugger documents, variables in the Array Editor, and figure windows. Select an entry in the **Windo w** menu to go directly to that window or tabbed document. Select **Close All** to close all items listed in the **Windo w** menu.

For example, the **Windo w** menu in the following illustration shows three documents open in the Array Editor and two documents open in the Editor/ Debugger. Selecting variance.m, for example, makes the Editor/Debugger window with the file variance.m become the active window.



CloseAll closes all items listed in the **Window** menu.

Three variables are open in the Array Editor: x, y, and m.

Two files are open in the Editor/Debugger: sqsum.m and variance.m.

Click on an item to go directly to that window.

Closing Desktop Tools

To close a desktop tool, do one of the following:

- Select the item in the **Vie w** menu (the item becomes unchecked).
- Click the close box  in the window's title bar.
- Select **Close** from the **File** menu to close the current window. The window closes.

Resizing Windows

To resize windows in the MATLAB desktop, use the separator bar, which is the bar between two windows:

1 Move the cursor onto the separator bar.

The cursor assumes a different shape. On Windows platforms, it is a double-headed arrow .

On UNIX, it is an arrow with a bar. 2 Drag the separator bar to change the sizes of the windows. Drag separator bar to resize windows in the desktop



To resize the

MATLAB desktop itself or windows for MATLAB tools outside of the desktop, drag any edge or corner of the window.

Moving Windows

There are three basic ways to move MATLAB desktop windows:

- “Moving Windows Within the MATLAB Desktop” on page 2-10
- “Moving Windows Out of the MATLAB Desktop” on page 2-12 and “Moving Windows Into the MATLAB Desktop” on page 2-13
- “Grouping (Tabbing) Windows Together” on page 2-13

Moving Windows Within the MATLAB Desktop

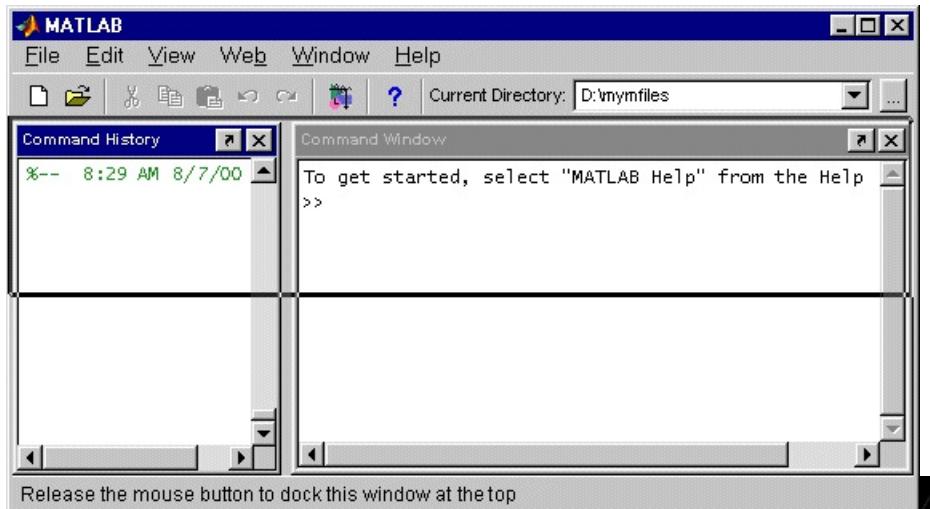
To move a window to another location in the MATLAB desktop:

- 1 Drag the title bar of the window towards where you want the window to be located.

As you drag the window, an outline of it appears. When the outline nears a position where you can dock (keep) it, the outline snaps to that location. The status bar displays instructions about moving the window while you drag the outline.

In the following example, the Command History window is originally to the left of the Command Window and is being dragged above the Command Window. When the top of Command History window touches the bottom of the toolbar, the outline appears.

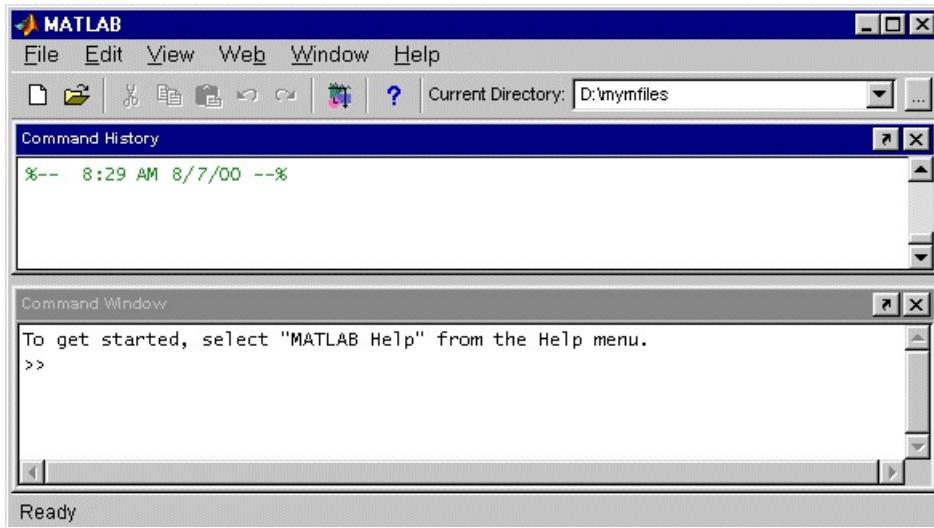
Drag the title bar of the Command History window to move it.
Release the mouse when the outline of the Command History window appears where you want it to be.



The status bar displays instructions about moving the window.

- 2 Release the mouse to dock the window at the new location.

Other windows in the desktop resize to accommodate the new configuration. The following example shows how the desktop looks after having moved the Command History window above the Command Window.



Moving Windows Out of the MATLAB Desktop

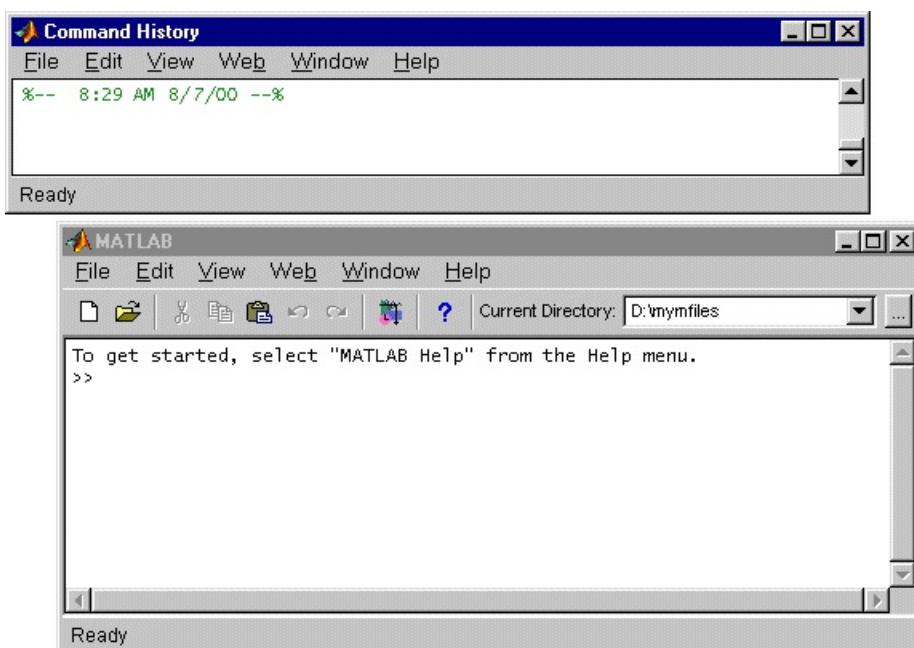
To move a window outside of the MATLAB desktop, do one of the following:

- Click the arrow in the title bar of the window you want to move outside of

the desktop.

- Select **Undock** for that tool from the **View** menu; the window must be the currently active window.
- Drag the title bar of the window outside of the desktop. As you drag, an outline of the window appears. When the cursor is outside of the MATLAB desktop, release the mouse.

The window appears outside of the MATLAB desktop. In the following example, the Command History window has been moved outside of the desktop.



Moving Windows Into the MATLAB Desktop

To move a window that is outside of the MATLAB desktop into the desktop, select **Dock** for that tool from the window's **View** menu.

Grouping (Tabbing) Windows Together

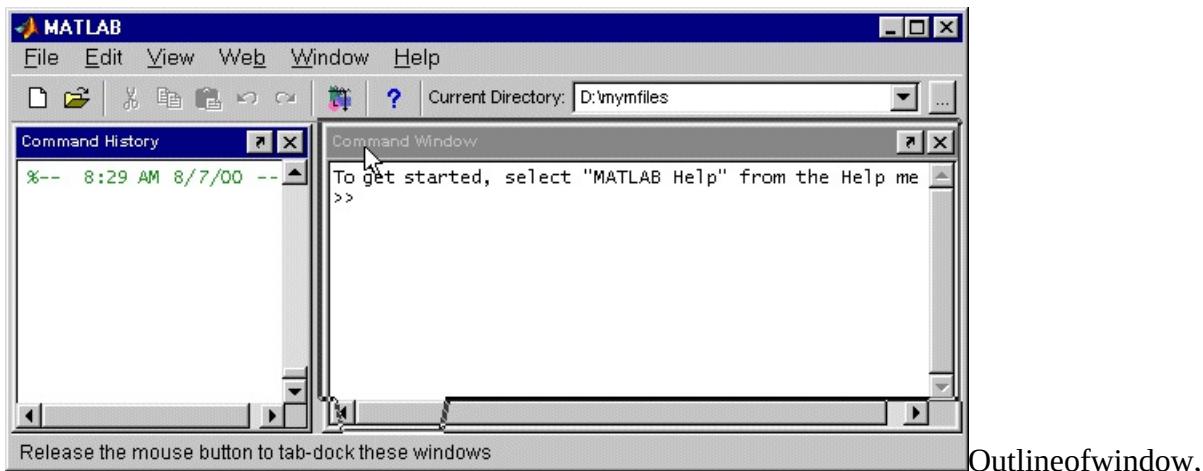
You can group windows so that they occupy the same space in the MATLAB desktop, with access to the individual windows via tabs. These are the main features in working with tabbed windows:

- “Grouping Windows” on page 2-14
- “Viewing Tabbed Windows” on page 2-15
- “Moving Tabbed Windows” on page 2-15
- “Closing Tabbed Windows” on page 2-15

Grouping Windows. To group (also called “to tab”) windows together:

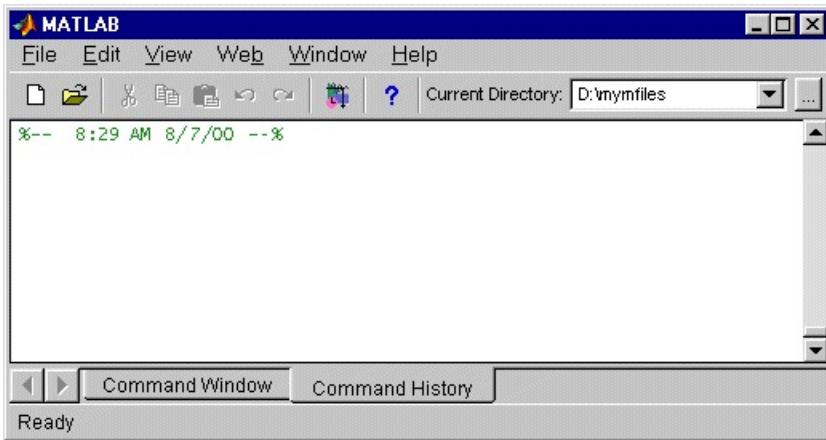
1 Drag the title bar of one window in the desktop on top of the title bar of another window in the desktop.

The outline of the window you’re dragging overlays the target window, and the bottom of the outline includes a tab. In the following example, the Command History window is originally to the left of the Command Window and its title bar is being dragged on top of the title bar of the Command Window.



2 Release the mouse.

Both windows occupy the same space and labeled tabs appear at the bottom of that space. In the following example, the Command History and Command Window are tabbed together, with the Command History tab currently selected.



Use arrows to show any tabs that aren't in view. In this example, the arrows are grayed, indicating all tabs are in view.

There are labeled tabs for each tool tabbed together in the window. Click a tab to view that tool.

Viewing Tabbed Windows. To view a tabbed window, click the window's tab. The window moves to the foreground and becomes the currently active window. If there are more tabs in a window than are currently visible, use the arrows to the left of the tabs to see additional tabs.

Moving Tabbed Windows. To move a tabbed window to another location, drag the title bar or the tab to the new location. You can move it inside or outside of the MATLAB desktop.

Closing Tabbed Windows. When you click the close box

for a window that is

part of a group of windows tabbed together, that window closes. You cannot close all of the tabbed windows at one time; instead close each window individually.

Using Predefined Desktop Configurations

There are six predefined MATLAB desktop configurations, which you can select from the **View** \rightarrow **Desktop Layout** menu:

- **Default** – Contains the Command Window, the Command History and Current Directory browser tabbed together, and the Launch Pad and Workspace browser tabbed together.
- **Command Window Only** – Contains only the Command Window. This makes MATLAB appear similar to how it looked in previous versions.
- **Simple** – Contains the Command History and Command Window, side-by-side.
- **Short History** – Contains the Current Directory browser and Workspace browser tabbed together above the Command Window and a small Command History.
- **Tall History** – Contains the Command History along the left, and the Current Directory browser and Workspace browser tabbed together above the Command Window.
- **Five Panel** – Contains the Launch Pad above the Command History along the left, the Workspace browser above the Current Directory browser in the center, and the Command Window on the right.

After selecting a predefined configuration, you can move, resize, and open and close windows.

Common Desktop Features

These common features are available for the desktop tools:

- “Desktop Toolbar” on page 2-17
- “Context Menus” on page 2-17
- “Keyboard Shortcuts and Accelerators” on page 2-18
- “Selecting Multiple Items” on page 2-19
- “Using the Clipboard” on page 2-20
- “Accessing The MathWorks on the Web” on page 2-20

Desktop Toolbar

The toolbar in the MATLAB desktop provides easy access to popular operations. Hold the cursor over a button and a tooltip appears describing the item. Note that some of the tools also have a toolbar within their window.

Create new M-file Undelete Create new Simulink model

Open file Copy Redo View or change current directory

last Select previously used Cut Paste undo Goto Help browser current directory



Tooltip describes button

Turn tool tips on or off using **General Preferences**

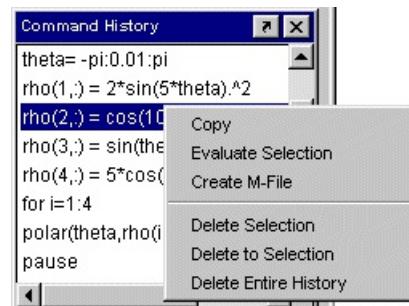
Browse to change current directory

Current Directory Field

The **Current Directory** field in the toolbar shows MATLAB’s current working directory. You can change the current directory using this field and perform other file operations using the Current Directory browser – for instructions, see “File Operations” on page 5-20.

Context Menus

Many of the features of the MATLAB desktop tools are available from context menus, also known as pop-up menus. To access a context menu, right-click on a selection and the context menu for it appears, presenting the available actions. For example, following is the context menu for a selection in the Command History window.



Access context (pop-up) menus by right-clicking on a selection.

Keyboard Shortcuts and Accelerators

You can access many of the menu items using keyboard shortcuts or accelerators for your platform, such as using **Ctrl+X** to perform a **Cut** on Windows platforms, or **Alt+F** to open the **File** menu. Many of the shortcuts and accelerators are listed with the menu item. For example, on Windows platforms, the **Edit** menu shows **Cut Ctrl+X**, and the **File** menu shows the **F** in **File** underlined, which indicates that **Alt+X** opens it. Many standard shortcuts for your platform will work but are not listed with the menu items.

Following are some additional shortcuts that are not listed on menu items.

Keys Result

Enter The equivalent of double-clicking, it performs the default action for a selection. For example, pressing **Enter** while a line in the Command History window is selected runs that line in the Command Window.

Escape Cancels the current action.

Moves to the next tab in the desktop, where the tab is **Ctrl+Tab**
or

for a tool, or for a file in the Editor/Debugger. When used in the Editor/Debugger in tabbed mode outside **Ctrl+F 6** of the desktop, moves to the next open file.

Keys Result (Continued) Ctrl+Shift+Tab

Ctrl+Page Up

Ctrl+Page Down Alt+F 4 Alt+Space

Moves to the previous tab in the desktop, where the tab is for a tool, or for a file in the Editor/Debugger. When used in the Editor/Debugger in tabbed mode outside of the desktop, moves to the previous open file.

Moves to the next tab within a group of tools or files tabbed together.

Moves to the previous tab within a window. Closes desktop or window outside of desktop.
Displays the system menu.

On the Alpha platform, even though shortcuts are not listed on the menu items, most standard Alpha shortcuts will work.

Selecting Multiple Items

In many of the desktop tools, you can select multiple items and then select an action to perform on all of the selected items. Select multiple items using the standard practices for your platform.

For example, if your platform is Windows, do the following to select multiple items:

1 Click on the first item you want to select.

2 Hold the **Ctrl** key and then click on the next item you want to select. Repeat this step until you've selected all the items you want.

If you hold the **Shift** key instead of the **Ctrl** key while clicking on an item, you'll select all the items between and including that item and the last item you clicked on.

Now you can perform an action, such as delete, on the selected items.

Using the Clipboard

You can cut and copy a selection from a desktop tool to the clipboard and then paste it from the clipboard into another desktop tool. Use the **Edit** menu, context menus, or standard keyboard shortcuts. For example, you can copy a selection of commands from the Command History window and paste them into the desktop.

The **Paste Special** item in the **Edit** menu opens the selection on the clipboard in the Import Wizard. You can use this to copy data from another application, such as Excel, into MATLAB. For details, see Chapter 6, “Importing and Exporting Data.”

To undo the most recent cut, copy, or paste command, select **Undo** from the **Edit** menu. Use **Redo** to reverse the **Undo**.

You can also copy by dragging a selection. For example, make a selection in the Command History window and drag it to the Command Window, which pastes it there. Edit the lines in the Command Window, if needed and then press the **Enter** key to run the lines from the Command Window.

Accessing The Math Works on the Web

You can access popular MathWorks Web pages from the MATLAB desktop. Select one of the following items from the **Web** menu – the Web page opens in your default Web browser:

- **The MathWorks Web Site** – Links you to the home page of the MathWorks

Web site (<http://www.mathworks.com>).

- **Technical Support Knowledge Base** – Links you to the MathWorks Support page (<http://www.mathworks.com/support>), where you can look for solutions for problems you are having or report new problems.

- **Products**– Links you to the MathWorks Products page (<http://www.mathworks.com/products/>), where you can get information about the full family of products.

- **Membership**– Links you to the MATLAB Access page (<http://www.mathworks.com/mla/index.shtml>) for Access members. If you are not a member, you can join online to help you keep up-to-date on the latest MATLAB developments.

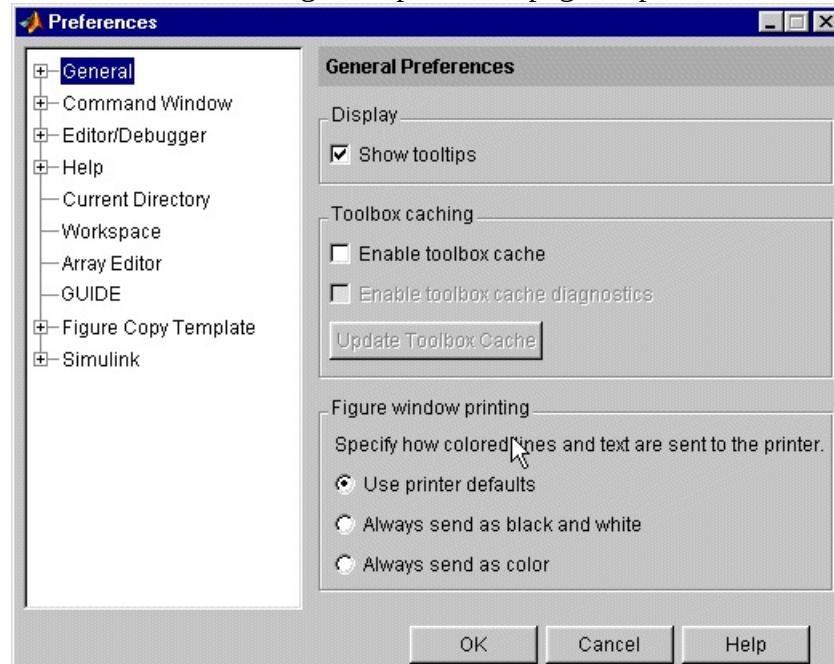
Setting Preferences

Set preferences to modify the default behavior of some aspects of MATLAB such as the font used in the Command Window. Preferences remain persistent across MATLAB sessions. Note that some tools allow you to control these aspects from within the tool without setting a preference – use that method if you only want the change to apply to the current session.

To set preferences:

1 Select **Preferences** from the **File** menu.

The **Preferences** dialog box opens. The page it opens to reflects the currently active window.



2 In the left pane, select the type of preferences you want to specify. In the above example, **General** preferences are selected. If a + appears to the left of an item, click the + to display more items, and then select the item you want to set preferences for.

The right pane reflects the type of preference you selected.

3 In the right pane, specify the preference values and click **OK**. The preferences take effect immediately.

Type of Preference and Where Described

Items You Can Set Preferences For

"General Preferences for MATLAB" on page 2-23

"Preferences for the Command Window" on page 3-13

"Preferences for the Editor/Debugger" on page 7-35

"Printing Documentation" on page 4-26

"Preferences for the Current Directory Browser" on page 5-30

"Preferences for the Workspace Browser" on page 5-9

"Preferences for the Array Editor" on page 5-12

For desktop display, caching, printing, fonts, colors, and source control system

Numeric format and display, echo, font, and colors
Startup options, font, colors, display, keyboard shortcuts, indenting, and printing
Documentation location, products, PDF reader location, synchronization, and fonts
Number of entries in history and file display options
Font and confirm deletion of variables
Font and numeric format

Type of Preference and Where Described

Items You Can Set Preferences For (Continued)

GUIDE

Figure Copy Template

Simulink Display options

Application, text, line, uicontrols, axis, format, background color, and size
Display, fonts, and simulation.

The preferences file is matlab.prf. Type `prefdir` in the Command Window to see the location of the file. The `matlab.prf` file is loaded when MATLAB starts and overwritten when you close MATLAB.

General Preferences for MATLAB

These preferences apply to all relevant tools in the MATLAB desktop.

Display

To show tooltips when you hold the cursor over a toolbar button, check the **Show tooltips** check box.

Toolbox caching

See “Reducing Startup Time with Toolbox Path Caching” on page 1-10.

Figure window printing

Specify how colored lines and text are sent to the printer. See the Printing documentation for more information.

Font & Colors

Desktop font. Desktop font preferences specify the characteristics of the font used in tools under the control of the MATLAB desktop. The font characteristics are:

- Type, for example, SansSerif
- Style, for example, bold
- Size in points, for example, 12 points

After you make a selection, the **Sample** area shows how the font will look. Lucida Console approximates the fixedsys font available in previous versions of MATLAB.

You can specify a different font for the Command Window, Editor/Debugger, Help browser, Workspace browser, and Array Editor using preferences for those tools.

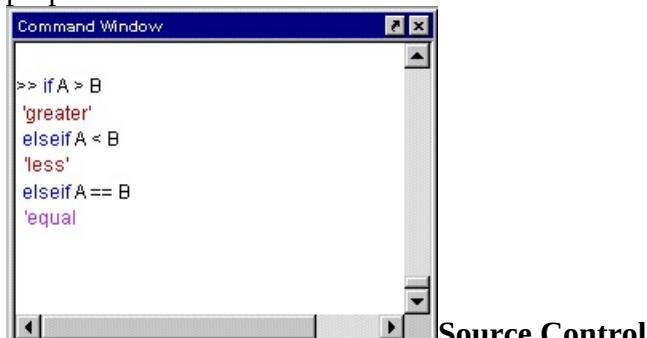
Syntax highlighting colors. Select the colors to use to highlight syntax. For more information, see “Syntax Highlighting” on page 3-5.

- **Key words**— Flow control and other functions such as for and if are colored.
- **Comments** – All lines beginning with a % are colored.
- **Strings**— Single quotes and whatever is between them are colored.
- **Unterminated strings** – A single quote without a matching single quote, and whatever follows or precedes the quote are colored.
- **System commands** – Commands such as the ! (shell escape) are colored.
- **Errors** – The error text is colored.

Click **Restore Default Colors** to return to the default settings. The following example uses the default values for color preferences.

Keywords, like
these flow control
commands are blue.

Closed strings are maroon.
Unclosed strings are
purple.



Source Control

Specify the source control system you want to interface MATLAB to. For more information, see Chapter 9, “Interfacing with Source Control Systems.”

3

Running MATLAB Functions

The Command Window	3-2
Opening the Command Window.....	3-2
Running Functions and Entering Variables	3-2
Controlling Input and Output.....	3-4
Running Programs.....	3-11
Keeping a Session Log.....	3-12
Preferences for the Command Window	3-13
Command History	3-17
Viewing Functions in the Command History Window	3-17

Running Functions from the Command History Window	3-18
Copying Functions from the Command History Window.....	3-19

The Command Windo w

The Command Window is the main way you communicate with MATLAB. It appears in the desktop when you first start MATLAB. Use the Command Window to run MATLAB functions (also referred to as commands) and perform MATLAB operations. The main features of the Command Window are:

- “Opening the Command Window” on page 3-2
- “Running Functions and Entering Variables” on page 3-2
- “Controlling Input and Output” on page 3-4, such as suppressing output and command line editing
- “Running Programs” on page 3-11, including M-files and external programs
- “Keeping a Session Log” on page 3-12
- “Preferences for the Command Window” on page 3-13

Opening the Comm and Windo w

To show the Command Window in the MATLAB Desktop, select **Command Windo w** from the **Vie w** menu – see “Opening and Closing Desktop Tools” on page 2-7 for details.



Running Functions and Entering Variables

The prompt (>>) in the Command Window indicates that MATLAB is ready to accept input from you. When you see the>> prompt, you can enter a variable or run a function. For example, to create A, a 3-by-3 matrix, type

A = [1 2 3; 4 5 6; 7 8 10]

When you press the **Enter** or **Return** key after typing the line, MATLAB responds with
A =

```
1 2 3
4 5 6
7 8 10
```

To run a function, type the function including all arguments and press **Return** or **Enter**. MATLAB displays the result. For example, type

```
magic(2)  
and MATLAB returns
```

```
ans =  
1 3  
4 2
```

If you want to enter multiple lines before running, use **Shift+Enter** or **Shift+Return** after each line until the last. Then press **Enter** or **Return** to run all of the lines.

The **K>>** prompt in the Command Window indicates that MATLAB is in debug mode. For more information, see Chapter 7, “Editing and Debugging M-Files.”

Evaluating a Selection

To run a selection in the Command Window, make the selection, and then right-click and select **Evaluate Selection** from the context menu. You cannot evaluate a selection while MATLAB is busy, for example, running an M-file.

Opening a Selection

To open the M-file for a function, select that function in the Command Window, and then right-click and select **Open Selection** from the context window. The M-file for that function opens in the Editor/Debugger.

Running One Process

You can only run one process at a time. If MATLAB is busy running one function, any commands you issue will be stacked. The next command will run when the previous one finishes. For example, you cannot set breakpoints from the Editor/Debugger while MATLAB is running a function in the Command Window.

Controlling Input and Output

You can control and interpret input and output in the Command Window in these ways:

- “Case and Space Sensitivity” on page 3-4
- “Entering Multiple Functions in a Line” on page 3-4
- “Entering Long Lines” on page 3-5
- “Syntax Highlighting” on page 3-5
- “Font Used in the Command Window” on page 3-6
- “Command Line Editing” on page 3-6
- “Clearing the Command Window” on page 3-9
- “Suppressing Output” on page 3-9
- “Paging of Output in the Command Window” on page 3-10
- “Controlling the Format and Spacing of Numeric Output” on page 3-10
- “Printing Command Window Contents” on page 3-11

Case and Space Sensitivity

MATLAB is case sensitive. For example, you cannot run the function `Plot` but must instead use `plot`. Similarly, the variable `le`a is not the same as the variable `A`. Note that if you use the help function, function names are shown in all uppercase, for example, `PLOT`, solely to distinguish

them. Do *not* use uppercase when running the functions. Some functions for interfacing to Java actually used mixed case and the M-file help accurately reflects that.
Blank spaces around operators such as $-,:,$ and $(),$ are optional, but they improve readability.

Entering Multiple Functions in a Line

To enter multiple functions on a single line, separate the functions with a comma (,) or semicolon (;). Using the semicolon instead of the comma will suppress the output for the command preceding it. For example, put three functions on one line to build a table of logarithms by typing

```
format short; x = (1:10)'; logs = [x log10(x)]
```

and then press **Enter** or **Return** to run the functions in left-to-right order.

Entering Long Lines

If a statement does not fit on one line, use an ellipsis (three periods, ...) to indicate that the statement continues on the next line, press **Enter** or **Return** to advance to the next line, and then continue entering the statement. For example,

```
s = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7 ...
- 1/8 + 1/9 - 1/10 + 1/11 - 1/12;
```

For items in single quotes, such as strings, put the quotes in each line. For example, entering the following long string

```
headers = ['Author Last Name, Author First Name, ' ... 'Author Middle Initial']
```

results in

```
headers =
```

Author Last Name, Author First Name, Author Middle Initial The maximum number of characters allowed on a single line is 4096.

Syntax Highlighting

Some entries appear in different colors to help you better find elements, such as matching if/else statements:

- Type a string and it is colored purple. When you close the string, it becomes maroon.
- Type a keyword, such as the flow control function for, or a continuation (ellipsis ...), and it is colored blue. Lines you enter between the opening and closing flow control functions are indented.
- Double-click an opening or closing token, a parenthesis (), bracket [], or brace { }. This selects the characters between the token and its mate.
- Type a closing (or opening) token and the matching opening (or closing) token is highlighted briefly.
- Type a mismatched closing (or opening) token and a strikethrough character appears on the token. For example

```
>> CC=C{:+}
```

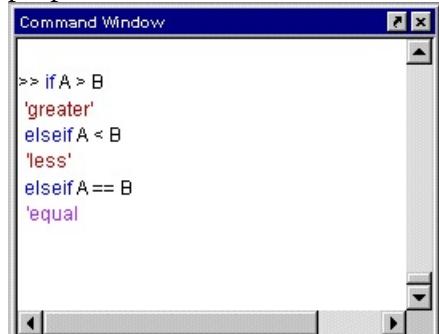
- Use an arrow key to move over an opening or closing token. That token and its matching closing or opening token briefly appear underlined.

- Type a comment symbol, %, and what follows on the line appears in green. That information is treated by MATLAB as a comment.
- Type a system command, such as the! (shell escape), and the line appears in gold.
- Errors appear in red.

Default colors are shown here—[to change them, use Preferences](#).

Keywords, like
these flow control
commands are blue.

Closed strings are maroon.
Unclosed strings are
purple.



A screenshot of the MATLAB Command Window titled "Command Window". The window contains the following text:

```
>> if A > B
'greater'
elseif A < B
'less'
elseif A == B
'equal'
```

The words "if", "else", "elseif", "==" and the strings "greater", "less", "equal" are highlighted in different colors: "if", "else", "elseif", "==" are blue, while the strings are maroon. The rest of the text is black.

To change the colors used for syntax highlighting, see “Font & Colors Preferences for the Command Window” on page 3-15.

Font Used in the Command Window

You can specify the font type, style, and size used in the Command Window. For instructions, see “Font & Colors Preferences for the Command Window” on page 3-15.

Command Line Editing

These are time-saving features you can use in the Command Window:

- Clipboard features
- Recalling previous lines
- Tab completion

Clipboard Features. Use the **Cut**, **Copy**, **Paste**, **Undo**, and **Redo** features from the **Edit** menu when working in the Command Window. Some of these features are also available in the context menu for the Command Window.

Recalling Previous Lines. Use the arrow, tab, and control keys on your keyboard to recall, edit, and reuse functions you typed earlier. For example, suppose you mistakenly enter

`rho = (1+ sqrt(5))/2`
MATLAB responds with
Undefined function or variable 'sqrt'.

because you misspelled sqrt. Instead of retyping the entire line, press the

■

key. The previously typed line is redisplayed. Use the left arrow key to move the cursor and add the missingr. Repeated use of the up arrow key recalls earlier lines.

The functions you enter are stored in a buffer. You can use *smart recall* to recall a previous function whose first few characters you specify. For example, typing the lettersplo and pressing the up arrow key recalls the last function that started withplo, as in the most recentplot function. This feature is case sensitive.

Following is the complete list of arrow and control keys you can use in the Command Window. Many of these keys should be familiar to users of the Emacs editor.

Key Control Key Operation

■ **Ctrl+P**

■

Ctrl+N

■

Ctrl+B

■

Ctrl+F

Ctrl+

■ **Ctrl+R**

■ **Ctrl+L**

Home Ctrl+A End Ctrl+E

Recall previous line. See also “Command History” on page 3-17, which is a log of previously used functions, and “Keeping a Session Log” on page 3-12.

Recall next line.

Move back one character.

Move forward one character.

Move right one word.

Move left one word.

Move to beginning of line.

Move to end of line.

Esc Ctrl+U Delete Ctrl+D Backspace Ctrl+H

Ctrl+K Shift+Home Shift+End

Clear line.

Delete character at cursor. Delete character before cursor. Delete (kill) to end of line. Highlight to beginning of line. Highlight to end of line.

Tab Completion. MATLAB completes the name of a function, variable, filename, or handle graphics property if you type the first few letters and then press the **Tab** key. If there is a unique name, the name is automatically completed. For example, if you created a variable costs_march, type

costs

and press **Tab**. MATLAB completes the name, displaying costs_march

Press **Return** or **Enter** to run the statement. In this example, MATLAB displays the contents of costs_march.

If there is more than one name that starts with the letters you typed, press the **Tab** key again to see a list of the possibilities. For example, type cos and press **Tab**. MATLAB does not display anything, indicating there are multiple names beginning with cos. Press **Tab** again and MATLAB displays cos cosh costfun cos_tr cosint costs_march

The resulting list of possibilities includes the variable name you created, costs_march, but also includes functions that begin with cos.

Note that the list might include files which are not valid commands, including private functions.
Clearing the Command Window

Select **Clear Command Window** from the **Edit** menu to clear it. This does not clear the workspace, but only clears the view. Afterwards, you still can use the up arrow key to recall previous functions.

Function Equivalent. Use clc to clear the Command Window. Similar to clc, the home function moves the prompt to the top of the Command Window.

Suppressing Output

If you end a line with a semicolon (;), MATLAB runs the statement but does not display any output when you press the **Enter** or **Return** key. This is particularly useful when you generate large matrices. For example, typing

A = magic(100);

and then pressing **Enter** or **Return** creates A but does not display the resulting matrix.

Paging of Output in the Command Window

If output in the Command Window is lengthy, it might not fit within the screen and will display too quickly for you to see it. Use the more function to control the paging of output in the Command Window. By default, more is off. When you type more on, MATLAB displays only a page (a screen full) of output at a time. After the first screen displays, press one of the following keys.

Key Action **Enter** or **Return** To advance to the next line **Space Bar** To advance to the next page **q** To stop displaying the output

Controlling the Format and Spacing of Numeric Output

By default, numeric output in the Command Windows is displayed as 5-digit scaled, fixed-point values. Use the text display preference to change the numeric format of output. The text display format affects only how numbers are shown, not how MATLAB computes or saves them.

Function Equivalent. Use the `format` function to control the output format of the numeric values displayed in the Command Window. The format you specify applies only to the current session. More advanced alternatives are listed in the “See Also” section of the `format` reference page.

Examples of Formats. Here are a few examples of the various formats and the output produced from the following two-element vector `x`, with components of different magnitudes.

```
x = [4/3 1.2345e-6]
format short e
1.3333e+000 1.2345e-006
format short
1.3333 0.0000
format + ++
```

For a complete list and description of available formats, see the reference page for `format`. If you want more control over the output format, use the `sprintf` and `fprintf` functions.

Controlling Spacing. Use the text display preference or `format` function to control spacing in the output. Use
`format compact`

to suppress blank lines, allowing you to view more information in the Command Window. To include the blank lines, which can help make output more readable, use

`format loose`

Printing Command Window Contents

To print the complete contents of the Command Window, select **Print** from the **File** menu. To print only a selection, first make the selection in the Command Window and then choose **Print Selection** from the **File** menu.

Running Programs Running M-Files

Run M-files, files that contain code in the MATLAB language, the same way that you would run any other MATLAB function. Type the name of the M-file in the Command Window and press **Enter** or **Return**.

To display each function in the M-file as it executes, use the **Display** preference and check **Echo on**, or use the `echo` function set to `on`.

Interrupting a Running Program

You can interrupt a running program by pressing **Ctrl+C** or **Ctrl+Break** at any time. On Windows platforms, you may have to wait until an executing built-in function or MEX-file has finished its operation. On UNIX systems, program execution will terminate immediately.

Running External Programs

The exclamation point character,!, is a *shell escape* and indicates that the rest of the input line is a command to the operating system. Use it to invoke utilities or run other programs without quitting from MATLAB. On UNIX, for example,

```
!vi darwin.m
```

invokes the vi editor for a file named darwin.m. After the program completes or you quit the program, the operating system returns control to MATLAB. See the functions unix and dos to run external programs that return results and status.

Opening M-Files

To open an M-file, select the file or function name in the Command Window, and then right-click and select **Open Selection** from the context window. The M-file opens in the Editor/Debugger.

Examining Errors

If an error message appears when running an M-file, click on the underlined portion of the error message, or press **Ctrl+Enter**. The offending M-file opens in the Editor, scrolled to the line containing the error.

Keeping a Session Log The diary Function

The diary function creates a copy of your MATLAB session in a disk file, including keyboard input and system responses, but excluding graphics. You can view and edit the resulting text file using any word processor. To create a file on your disk called sept23.out that contains all the functions you enter, as well as MATLAB's output, enter

```
diary('sept23.out')
```

To stop recording the session, use

```
diary('off')
```

Other Session Logs

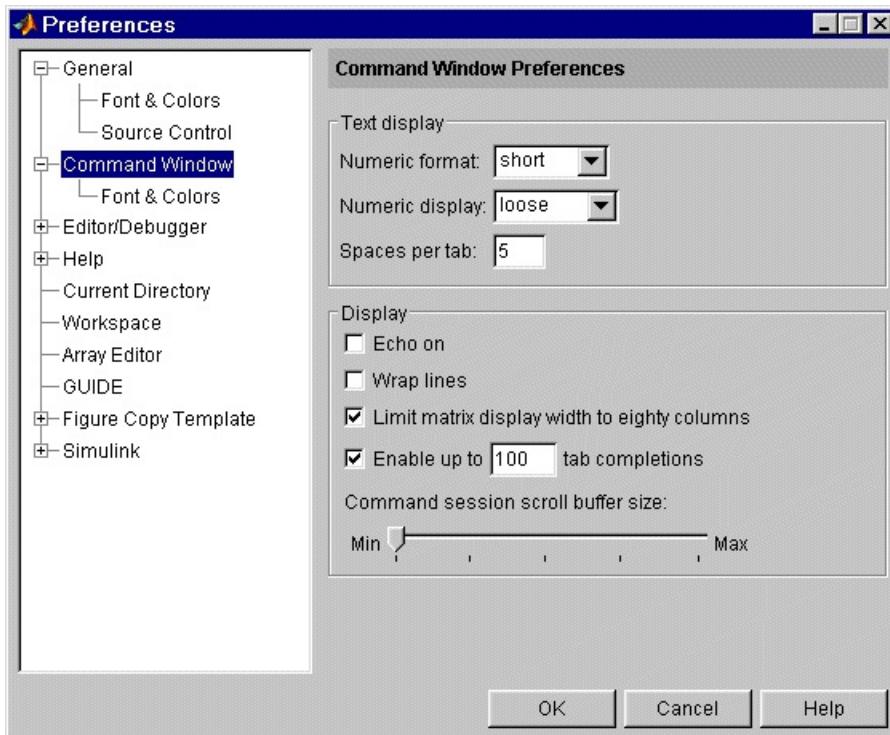
There are two other means of seeing session information:

- The Command History, which contains a log of all functions executed in the current and previous sessions
- For Windows platforms, the logfile startup option – see “Adding Startup Options for Windows Platforms” on page 1-5

Preferences for the Comm and Windo w

Using preferences, you can specify the format for how numeric values are displayed, set echoing on automatically for each session, specify the font type, style, and size, and set the colors used for syntax highlighting for contents of the Command Window.

To set preferences for the Command Window, select **Preferences** from the **File** menu in the Command Window. The **Preferences** dialog box opens showing **Command Windo w Preferences**.



Text Display and Display

Preferences for the Command Window Text display. Specify how output appears in the Command Window:

- **Numeric format** – Output format of numeric values displayed in the Command Window. This affects only how numbers are displayed, not how MATLAB computes or saves them. The [format](#) reference page includes the list of available formats.
- **Numeric display** – Spacing of output in the Command Window. To suppress blank lines, use `compact`. To display blank lines, use `loose`. For more information, see the reference page [forformat](#).
- **Spaces per tab** – Number of spaces assigned to a tab stop when displaying output. The default is 4.

Display. Specify echoing option and buffer size:

- **Echo on** – Check the box if you want commands running in M-files to display in the Command Window during the M-file execution. For more information, see the reference page [forecho](#).
- **Wrap lines** – Input and output in the Command Window uses multiple lines to fit within the current width of the Command Window. Note that the horizontal scroll bar does not appear in this mode because it is not needed.
- **Limit matrix display width to eighty columns** – Check the box if you want MATLAB to display only 80 columns of matrix output, regardless of the width of the Command Window. Uncheck the box if you make the Command Window wider than 80 columns and want matrix output to fill the width of the Command Window. See also the [display](#) reference page.

- **Enable up to n tab completions** – Check the box if you want to use tab completion when typing functions in the Command Window. Then enter a limit in the edit box. For example, if you enter 10, when you use the tab completion feature, MATLAB displays the list of possible completions if there are 10 or less. If there are more than 10, MATLAB displays a message stating there are more than 10 completions.

Uncheck the box if you do not want to use the tab completion feature. MATLAB moves the cursor to the next tab stop when you press the **Tab** key, rather than completing a function.

- **Command session scroll buffer size** – Set the size of the buffer that maintains a list of previously run commands to be used for command recall. See “Recalling Previous Lines” on page 3-7 for more information.

Font & Colors Preferences for the Command Window

Font. Command Window font preferences specify the characteristics of the font used in the Command Window. Select **Use desktop font** if you want the font in the Command Window to be the same as that specified for **General Font & Colors** preferences.

If you want the Command Window font to be different, select **Use custom font** and specify the font characteristics for the Command Window:

- Type, for example, SansSerif
- Style, for example, bold
- Size in points, for example, 12 points

After you make a selection, the **Sample** area shows how the font will look.

Colors. Specify the colors used in the Command Window:

- **Text color** – The color of nonspecial text; special text uses colors specified for

Syntax highlighting

- **Background color** – The color of background in the window
- **Syntax highlighting** – The colors to use to highlight syntax. If checked, click

Set Colors to specify them. For a description of syntax highlighting, see “Syntax Highlighting” on page 3-5.

Command History

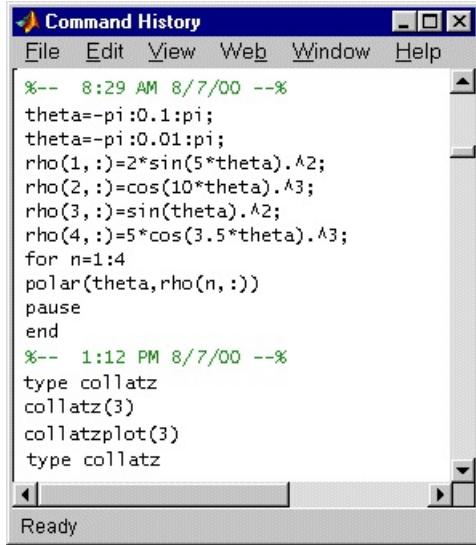
The Command History window appears when you first start MATLAB. The Command History window displays a log of the functions most recently run in the Command Window. To show or hide the Command History window, use the **View** menu – see “Opening and Closing Desktop Tools” on page 2-7 for details.

Use the Command History window for:

- “Viewing Functions in the Command History Window” on page 3-17
- “Running Functions from the Command History Window” on page 3-18
- “Copying Functions from the Command History Window” on page 3-19

Timestamp marks the start of each session.

Select one or more lines and right-click to copy, evaluate, or create an M-file from the



The screenshot shows the MATLAB Command History window. The menu bar includes File, Edit, View, Web, Window, and Help. The window displays a log of commands run on 8/7/00 at 8:29 AM and 1:12 PM. The code includes generating polar plots and running the Collatz conjecture function. The status bar at the bottom indicates 'Ready'.

```
%-- 8:29 AM 8/7/00 --%
theta=-pi:0.1:pi;
theta=-pi:0.01:pi;
rho(1,:)=2*sin(5*theta).^2;
rho(2,:)=cos(10*theta).^3;
rho(3,:)=sin(theta).^2;
rho(4,:)=5*cos(3.5*theta).^3;
for n=1:4
polar(theta,rho(n,:))
pause
end
%-- 1:12 PM 8/7/00 --%
type collatz
collatz(3)
collatzplot(3)
type collatz
```

Viewing Functions in the Command History Window

The log in the Command History window includes functions from the current session, as well as from previous sessions. The time and date for each session appear at the top of the history of functions for that session. Use the scroll bar or the up and down arrow keys to move through the Command History window. The Command History file is `history.m`. Type `prefdir` in the Command Window to see the location of the file. The `history.m` file is loaded when MATLAB starts and overwritten when you close MATLAB.

Deleting Entries in the Command History Window

Delete entries in the Command History window when you feel there are too many and it's inconvenient finding the ones you want. All entries remain until you delete them.

To delete entries in the Command History window, select an entry, or **Shift** +click or **Ctrl** +click to select multiple entries, or use **Ctrl+A** to select all entries. Then right-click and select one of the delete options from the context menu:

- **Delete Selection** – Deletes the selection
- **Delete to Selection** – Deletes all functions previous to (above) the selected function
- **Delete Entire History** – Deletes all functions in the Command History window

Another way to clear the entire history is by selecting **Clear Command History** from the **Edit** menu.

Running Functions from the Command History Window

Double-click on any function entry (entries) in the Command History window to execute that function(s). For example, double-clicked it myfile to open myfile.m in the Editor. You can also

run a function entry by right-clicking on it and selecting **Evaluate Selection** from the context menu, or by copying the function entry to the Command Window, as described in the next section.

Copying Functions from the Comm and History Windo w

Select a function entry, or **Shift+click** or **Ctrl+click** to select multiple entries, or use **Ctrl+A** to select all entries. Then you can do any of the following.

Action Ho w to Perform the Action

Run the functions in the command window

Copy the selection to the clipboard by right-clicking and selecting **Copy** from the context menu. Paste the selection into the Command Window. (Alternatively, drag the selection to the Command Window.) In the Command Window, edit the functions if desired, and press **Enter** or **Return** to execute the functions.

Copy the functions to another window

Copy the selection to the clipboard by right-clicking and selecting **Copy** from the context menu. Paste the selection into an open M-file in the Editor or any application.

Create an M-file from the functions

Right-click the selection and select **Create M-File** from the context menu. The Editor opens a new M-file that contains the functions you selected from the Command History window.

4

Getting Help

Types of Information	4-3
Using the Help Bro w ser	4-4
Using the Help Navigator.....	4-7
Vie w ing Documentation in the Display P ane.....	4-17
Preferences for the Help Bro w ser	4-21
Printing Documentation	4-26
Using Help Functions	4-28
Other Methods for Getting Help	4-31

The MathWorks provides online help for all products. Printed versions for some of the online documentation are also provided. The online material sometimes has information not included with the printed material and may be more current than the printed material.

The primary ways to access the online help are:

- “Using the Help Browser” on page 4-4 – Use the Help browser to find and view information about your MathWorks products. It includes a contents listing, global index, and search feature.
- “Using Help Functions” on page 4-28 – Typehelp functionname to get M-file help, which provides a brief description of the function and its syntax in the Command Window. Other help functions are available as well.

- “Other Methods for Getting Help” on page 4-31 – You can use product-specific help features, run demos, contact Technical Support, search documentation for other MathWorks products, view a list of other books, and participate in a MATLAB newsgroup.

In addition to using online help, you can print documentation – see “Printing Documentation” on page 4-26.

Types of Information

Types of Information

The Help browser and help functions provide access to the following types of documentation. Use the type of documentation most suited to your needs.

- Release Notes – An overview of new products and features in this release, it also includes upgrade information and any known problems and limitations. Review the Release Notes for all your products when you first start using the new release.
- Getting Started with ... – Primarily aimed at novice users, this documentation contains instructions for a product’s main features. Review Getting Started documentation before you begin using a product or feature for the first time. Then, to learn more, go to the “Using...” collections or reference pages.
- Using ... collections – This material contains overviews and complete instructions for using a product. Consult it after reviewing Getting Started material.
- Reference Pages – Every function has a reference page that provides the syntax, description, examples, and other information for that function. It includes links to related functions and additional information. Reference pages are also provided for blocks. Use reference pages to learn about a function or to see its syntax.
- M-File Help – Get M-file help in the Command Window to quickly access basic information for a function. It provides a brief description of a function and its syntax. It is called M-file help because the text of the help is a series of comments at the start of the M-file for a function.
- Product Page – Available on MathWorks Web site, a product page contains the latest product information, such as system requirements.
- Online Knowledge Base – This is the MathWorks Technical Support online knowledge base. It provides solutions to questions posed by users.

Using the Help Browser

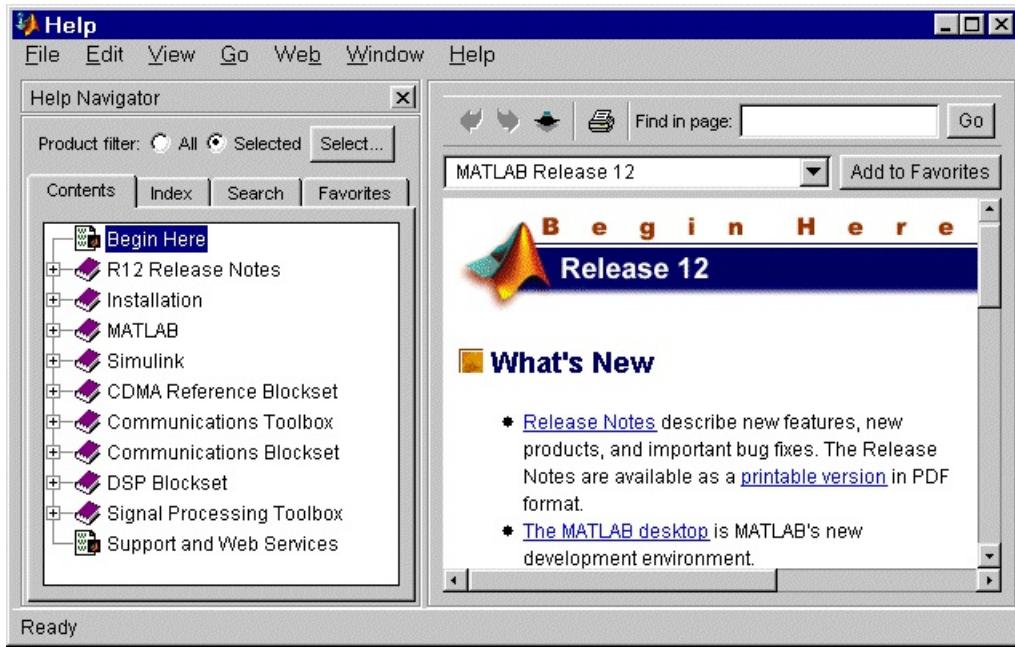
Use the Help browser to search and view documentation for MATLAB and your other MathWorks products. The Help browser is a Web browser integrated into the MATLAB desktop that displays HTML documents.

To open the Help browser, click the help button

 in the toolbar, or type

`helpbrowser` in the Command Window. You can also access the Help browser by selecting **Help** from the **View** menu or by using the **Help** menu in any tool. The Help browser opens.

Tabs in the **Help Navigator** pane provide different ways to find documentation. View documentation in the display pane. Use the close box to hide the pane. Drag the separator bar to adjust the width of the panes.



Ready

Using the Help

Browser

The Help browser consists of two panes:

- The Help Navigator on the left, which you use to find information. It includes a **Product Filter** and **Contents**, **Index**, **Search**, and **Favorites** tabs. For more information, see “Using the Help Navigator” on page 4-7.
- The display pane on the right, which is for viewing documentation.

Changing the Size of the Help Browser

To adjust the relative width of the two panes, drag the separator bar between them. You can also change the font in either of the panes – use “Help Fonts Preferences – Specifying Font Name, Style, and Size” on page 4-24.

Once you’ve found the documentation you want, you can close the **Help Navigator** pane so there is more screen space to view the documentation itself. This is shown in the following figure. To close the **Help Navigator** pane, click

the close box

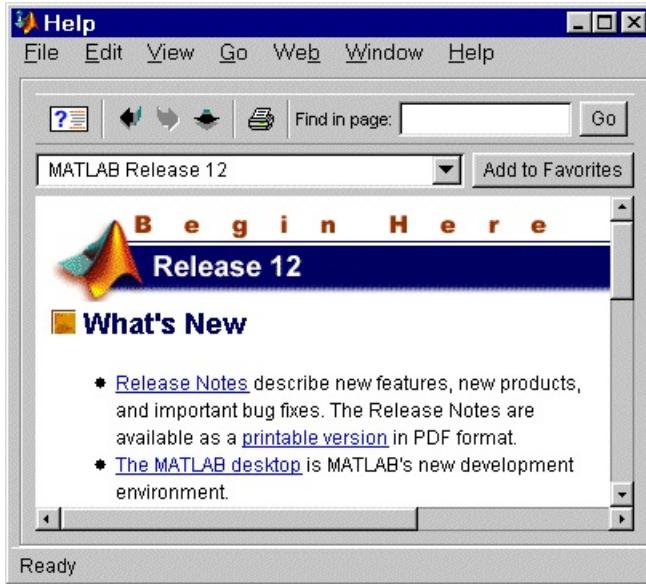
in the pane’s upper right corner or select **View -> Help View Options -> Show Help Navigator** from the menu, which unchecks it. To open the **Help Navigator** pane from the display pane, click the Help Navigator button

in the upper left corner of the Help browser, or select **View -> Help View Options -> Show Help Navigator**, which checks it.

To show only the display pane, as in this illustration, click the close box in the **Help Navigator** pane.

Click this button to

show the **Help Navigator** pane.



Using the Help Navigator

Use the Help Navigator, the left pane in the Help browser, to find information in the online help. These sections describe the main features:

- “Using the Product Filter” on page 4-7 – Show documentation only for specified products.
- “Viewing the Contents Listing in the Help Browser” on page 4-9 – View an expandable table of contents for documentation.
- “Finding Documentation Using the Index” on page 4-11 – Use keywords to find information.
- “Searching Documentation” on page 4-13 – Find documentation using full-text and other forms of search.
- “Bookmarking Favorite Pages” on page 4-15 – Designate favorite pages for later use.

Using the Product Filter

Use the **Product filter** in the **Help Navigator** to show documentation only for the products you specify.

Product filter: All Selected To show documentation for all MathWorks products installed on your system, select **All**.

To show only a subset of the documentation for MathWorks products installed on your system, set the **Product filter** to **Selected**, which results in the following:

- The **Contents** listing shows only the subset of products you specify.
- The **Index** shows only index terms for the subset of products you specify.
- The **Search** feature only looks through the subset of products you specify.

To specify the subset of products, click the **Select** button. The **Help Product Filter** dialog box opens.



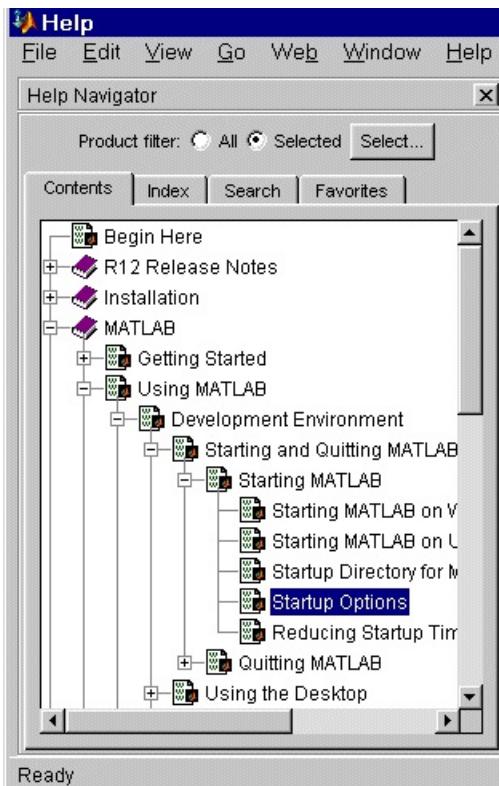
A checkmark appears for all products whose documentation is used in the Help Navigator. Make changes to the selected products and click **OK**. Then, with the **Product filter** set to **Selected**, the Help Navigator only shows documentation for those products you specified.

For example, if you're doing a search and know the information you're seeking is in MATLAB or the Communications Toolbox, in the **Help Product Filter**, click **Clear All** and then select MATLAB and Communications Toolbox. In the Help Navigator, set the **Product filter** to **Selected**. Then, the **Contents** would only show MATLAB and the Communications Toolbox documentation, the **Index** would only show entries for MATLAB and the Communications Toolbox, and the **Search** feature would only look in and show results for MATLAB and the Communications Toolbox.

The product filter settings are remembered for your next MATLAB session.

Viewing the Contents Listing in the Help Browser

To list the titles and table of contents for all product documentation, click the **Contents** tab in the **Help Navigator** pane.



Ready

In the **Contents** listing, you can:

- Click the + to the left of an item to expand the listing for the item.
- Click the - to the left of an item, or double-click the item to collapse the listings for that item.
- Select an item to show the first page of that document or section in the display pane.
- Double-click an item to expand the listing for that item and show the first page of that document or section in the display pane.
- Use the down and up arrow keys to move through the list of items.

The **Contents** listing shows documentation for all products installed on your system, or only shows documentation for specified products if you have the **Product filter** set to **Selected**.

Product Roadmap

When you select a product in the **Contents** pane (any entry with a book icon



), such as MATLAB or the Communications Toolbox, a *roadmap* of the documentation for that product appears in the display pane. The roadmap points to the key documentation for that product and provides links to:

- An index of documentation examples for that product
- The PDF version of the documentation, which is suitable for printing

Product Pages

After expanding the listing for a product in the Contents pane, the last entry is Product Page

(Web). This allows you to link to the MathWorks Web site for latest information about that product.

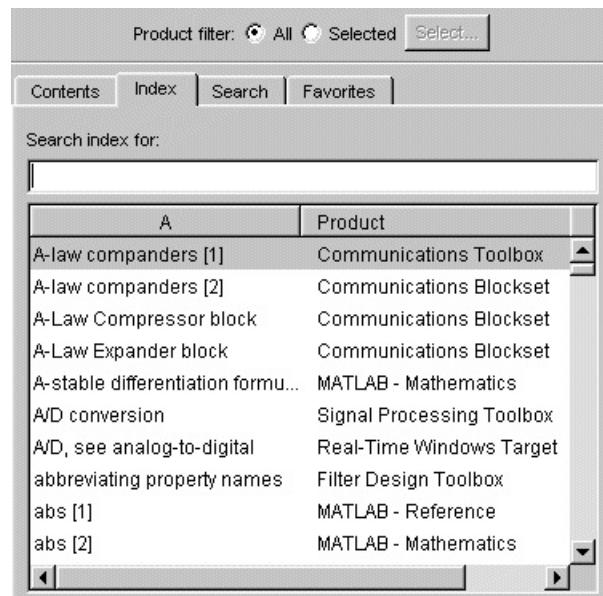
Contents Pane Is Synchronized with Display Pane

By default, the topic highlighted in the **Contents** pane always matches the title of the page appearing in the display pane. The **Contents** listing is said to be synchronized with the displayed document. This feature is useful if you access documentation with a method other than the **Contents** pane, for example, using a link in a page in the display pane. With synchronization, you always know what book and section the displayed page is part of.

You can turn off synchronization. To do so, use preferences – see “General – Synchronizing the Contents Pane with the Displayed Page” on page 4-23. Note that synchronization only applies to the **Contents** pane. The page shown in the display pane does not necessarily correspond to the selection in the **Search**, **Index**, or **Favorites** tabs.

Finding Documentation Using the Index

To find specific index entries (selected keywords) in the MathWorks documentation for your products, use the **Index** tab in the **Help Navigator** pane.



1 Set the **Product filter** to **All** or **Selected**.

2 Click the **Index** tab.

3 Type a word or words in the **Search index for** field. As you type, the index displays matching entries and their subentries (indented). It might take a few moments for the display to appear.

The product and title of the document that includes the matching index entry are listed next to the index entry, which is useful when there are multiple matching index entries. You might have to make the **Help Navigator** pane wider to see the product and document.

4 Select the index entry from the list to display that page.

The page appears in the display pane, scrolled to the location where the index entry appears.

5 To see more matching entries, scroll through the results.

Tips for Using the Index. If you're not satisfied with the resulting index entries, try one of the following:

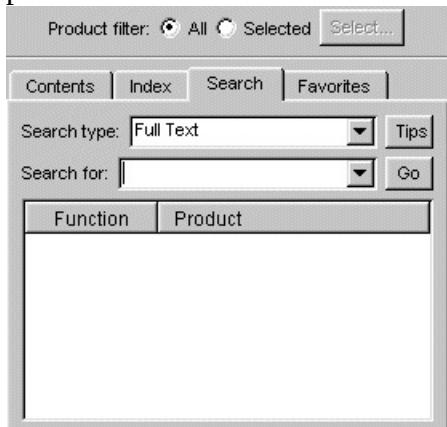
If the **Product filter** is set to **Selected**, change it to **All** to see more results. •

If it's set to **All**, change it to **Selected** to see fewer results.

- Type a different term or reverse the order of the words you type. For example, if you're looking for writing M-files, instead type M-files and look for the subentry creating.
- Search for the term in the displayed page using the **Find in page** field.
- Make the **Help Navigator** pane wider to see the name of the product and document to which the index entry pertains.
- Try the Search tab – for instructions, see “Searching Documentation” on page 4-13.

Searching Documentation

To look for a specific phrase in the documentation, use the **Search** tab in the **Help Navigator** pane.



1 Set the **Product filter** to **All** or **Selected**.

2 Click the **Search** tab.

3 Select a **Search type**:

- **Full Text** searches through all the text in the documentation. This can result in a very large number of results, so it's best to use this after using the **Document Titles** or **Function Name** search types.
- **Document Titles** searches through the headings in the documentation. This is the best way to start most searches.
- **Function Name** searches through the function reference pages. This is the equivalent of the doc function.
- **Online Knowledge Base** connects to the MATLAB Web site and searches through the Technical Support information.

4 Type the word or words you want to look for in the **Search for** field, and click **Go**(or press **Enter** or **Return**). The search automatically performs a boolean AND for multiple words. Do not explicitly type AND or similar boolean expressions between words and do not place quotes around terms.

The documents containing the search term are listed, grouped by product. The number of pages containing the search term is displayed in the status bar. For each result, the **Title** and **Section** of

the document containing the search phrase are displayed. You might have to make the **Help Navigator** pane wider to see the **Section** name.

5 Select an entry from the list of results.

The page containing the search term appears in the display pane and all occurrences of the search term are highlighted in the page. For example, if the search term is font preferences, all occurrences of font and all occurrences of preference are highlighted in the page. To see variants of the term in the page, use the **Find in page** field.

6 To see more matching search results, scroll through the list.

Tips for Using Search

Try “Narrowing Search Results” or “More Search Results and Information”.

Narrowing Search Results. If you see many results that are not relevant, try the following.

- Change the **Search type**. For example, for a general topic, use the type **Document Titles**, or to go to the reference page for a function use the **Function name** search type.
- Scroll through the list of results. Results are grouped by product, with a labeled banner separating each group.
- If the **Product filter** is set to **All**, change it to **Selected** to see fewer results.
- Type more than one word in the **Search for** field. For example, type font preferences. The search performs a Boolean AND between the search words. In the example font preferences, it finds all pages that have both the word font and the word preferences, although the page may not necessarily have the exact term “font preferences”. At the top of the results list are any pages that contain both words in the page title.
- Try the **Index**. For operators and special characters, you must use the **Index**, because **Search** won’t find them. Index results might return fewer but more relevant entries because the specified term is a keyword, whereas search results include pages with any mention of the specified term.

More Search Results and Information. To see more information about the search results or to get more results, try the following suggestions.

- Search for variants of the term or other terms on the displayed page using the **Find in page** field.
- If the **Product filter** is set to **Selected**, change it to **All** to see more results.
- Make the **Help Navigator** pane wider so you can see the second column, which displays the documentation section that the result appears in.
- To get more search results, try variations of the search word(s) since the search looks for an exact match. For example, search for preferences. If you don’t find what you want, try preference. If you are searching for information about an option, try including the - before the option, for example, save -append.

Book marking Favorite Pages

Click the **Favorites** tab in the **Help Navigator** to view a list of documents you previously designated as favorites. From the list of favorites you can:

- Select an entry – That document appears in the display pane.

- Remove an entry – Right-click the item in the favorites list and select **Remove** from the context menu, or press the **Delete** key.
- Rename an entry – Right-click the item in the favorites list and select **Rename** from the context menu. Type over the existing name to replace it with a new name.

Adding Favorites

To designate a document page as a favorite, do one of the following:

- While a page is open in the display pane, click the **Add to Favorites** button in the display pane toolbar.
- In the **Contents** listing, right-click an item and select **Add to Favorites** from the context menu.
- In the Help browser index or search results list, right-click an entry and select **Add to Favorites** from the context menu.

The bookmarks file is matlab_help.hst. Type prefdir in the Command Window to see the location of the file. Thematlab_help.hst file is loaded when MATLAB starts, and is overwritten when you close MATLAB.

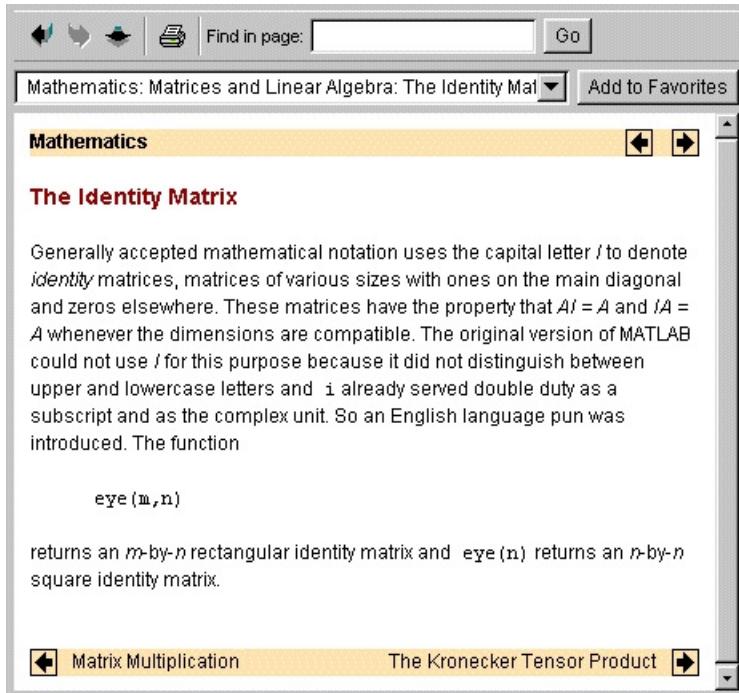
Viewing Documentation in the Display Pane

After finding documentation with the Help Navigator, view the documentation in the display pane. The features available to you while viewing the documentation are:

- “Browsing to Other Pages” on page 4-18
- “Bookmarking Pages” on page 4-18
- “Revisiting Pages” on page 4-19
- “Printing Pages” on page 4-19
- “Finding Terms in Displayed Pages” on page 4-19
- “Copying Information” on page 4-19
- “Evaluating a Selection” on page 4-19
- “Viewing the Page Source (HTML)” on page 4-20
- “Viewing Web Pages” on page 4-20

Browsing to Other Pages

Use the arrow buttons in the page and the toolbar to go to other pages.



View the next page in the document by clicking the right arrow at the top or bottom of the page. View the previous page in the document by clicking the left arrow .

at the top or bottom of the page. The arrows at the bottom of the page are labeled with the title of the page they go to.

View the page previously shown by clicking the back button

in the display pane toolbar. After using the back button, view the next page shown by clicking the forward button

in the display pane toolbar. You can also go back or forward by right-clicking on the page and selecting **Back** or **Forward** from the context menu, or by selecting **Back** or **Forward** from the **Go** menu.

Book marking Pages

Add the currently displayed page to your list of favorite documents by clicking **Add to Favorites** in the display pane toolbar.

Revisiting Pages

To display a page that you previously viewed in the current MATLAB session, select the page title from the drop-down list in the display pane toolbar.

Printing Pages

For instructions to print the currently displayed page, see “Printing a Page from the Help Browser” on page 4-26.

Finding Terms in Displayed Pages

To find a phrase in the currently displayed page:

- 1 In the **Find in page** field in the display pane toolbar, type the phrase you’re looking for. Then press **Enter** or **Return**, or click **Go**. You can type a partial word, for example, preference to find all occurrences of preference and preferences.

The page scrolls to the location containing the phrase, and highlights it.

- 2 Press **Enter** or **Return** again to find the next occurrence in that page.

See “Searching Documentation” on page 4-13 for instructions to look through all of the documentation instead of just on one page.

Copying Information

To copy information from the display pane, first select the information. Then either right-click and select **Copy** from the context menu, or select **Copy** from the **Edit** menu. You can then paste the information into another tool, such as the Command Window, or into another application, such as a word processor.

Evaluating a Selection

To run code examples that appear in the documentation, select the code in the display pane. Then right-click and select **Evaluate Selection** from the context menu, or select **Evaluate Selection** from the **Go** menu. The functions execute in the Command Window.

Viewing the Page Source (HTML)

To view the HTML source for the currently displayed page, select **View -> Help View Options -> Page Source**. The HTML version of the page appears in the Editor/Debugger. You can modify or copy the HTML source. To view a modified

page, use the reload button

in the display pane toolbar, or select **Reload** from the **Go** menu.

Viewing Web Pages

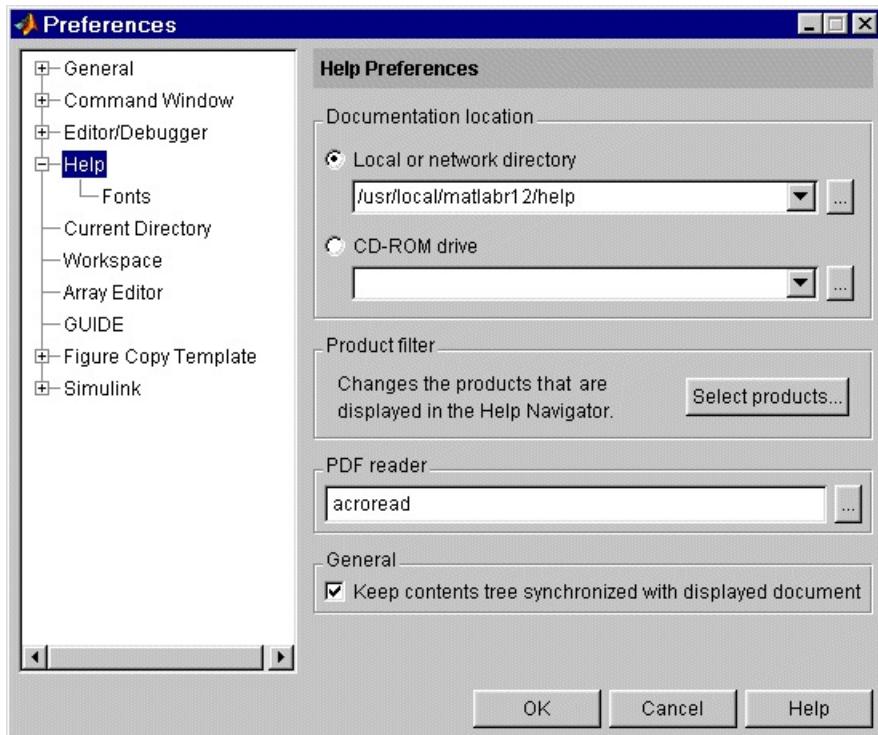
You can use the Help browser to view any Web page, although the Help browser might not support all the features you are used to in your usual Web browser. In the display pane page title field, type the URL and press the **Enter** key. For example, type www.mathworks.com. The MathWorks Web page appears in the Help browser.

Preferences for the Help Browser

Using preferences, you can specify the location of your help files, fonts used in the Help browser, and the products whose documentation you want to include in your Help browser.

To set preferences for the Help browser, select **Preferences** from the **File** menu. The **Preferences** dialog box opens showing **Help Preferences**.

CD-ROM drive preference is for Windows only



PDFreader preference is for UNIX systems only.

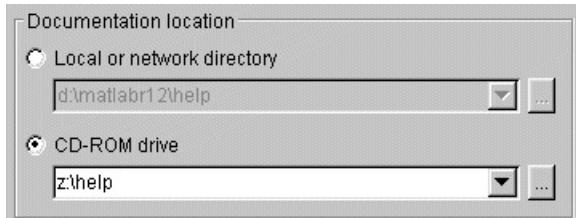
Set the preferences and then click **OK**. Help browser preferences include:

- “Documentation Location – Specifying the help Directory” on page 4-22
- “Product Filter – Limiting the Product Documentation” on page 4-23
- “PDF Reader – Specifying Its Location” on page 4-23
- “General – Synchronizing the Contents Pane with the Displayed Page” on page 4-23
- “Help Fonts Preferences – Specifying Font Name, Style, and Size” on page 4-24

Documentation Location – Specifying the help Directory

Use the **Documentation location** preference to specify where the MATLAB help directory resides for your system. The help directory contains the online help files used by the Help browser:

- If you elected to install the help files when you installed MATLAB, the documentation location should already be set to point to the help files. If the help files location changes or you want to access different help files, change the **Documentation location** for **Local or network directory**. On Windows, use the ... button to browse your file system to select the new location.
- On Windows platforms, if you did not install the help files during MATLAB installation, the Help browser attempts to find the files on the documentation CD. You need to specify the **Documentation location** to be **CD-ROM drive**. Use the ... button to browse your file system to select the drive’s location and select the help directory, as shown in the following example.



For UNIX platforms that do not support Java GUIs, use the `thedocopt` function to specify the location of your help directory.

Product Filter– Limiting the Product Documentation

If you have MathWorks products in addition to MATLAB, such as Simulink and toolboxes, you can use the **Product filter** to limit the product documentation used.

In **Help Preferences**, under **Product filter**, click **Select products**. The **Help Product Filter** dialog box opens. Select the products whose documentation you want to appear in the Help Navigator. Then, to use only those products you specified, in the Help browser set the **Product filter** to **Selected**. When you want to use all product documentation, in the Help browser set the **Product filter** to **All**. Note that you can also access the **Help Product Filter** dialog box by clicking the **Select** button in the Help browser.

PDF Reader– Specifying Its Location

If you want to access the PDF versions of the documentation, the Help system needs to know the location of your PDF reader (Adobe Acrobat). For Windows systems, MATLAB reads the location from the registry so you cannot specify its location in the preferences.

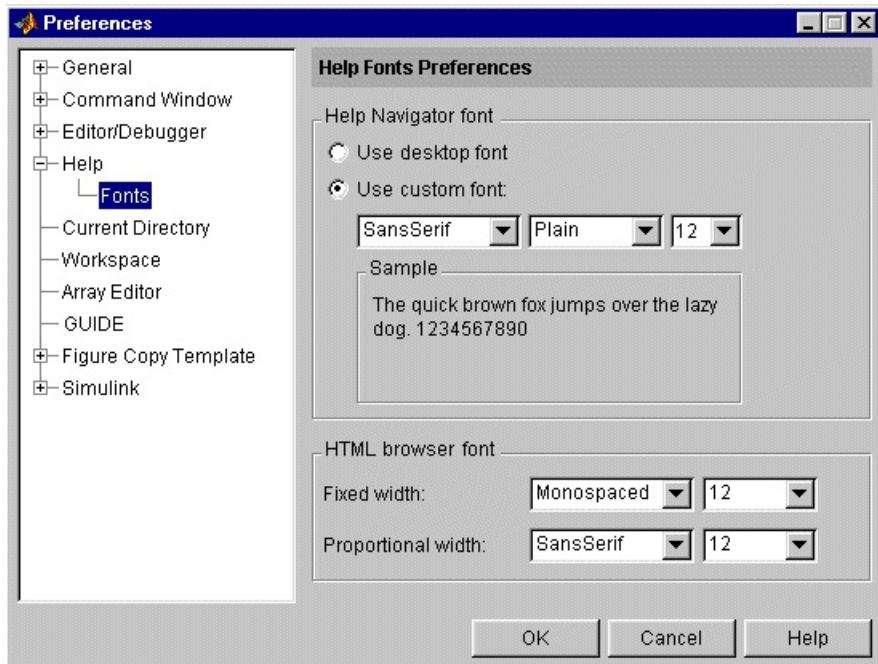
For UNIX systems, when you installed MATLAB, it looked for your system's PDF reader. If found, MATLAB automatically supplied the PDF reader location in the preferences **PDF reader** field. If MATLAB could not locate your PDF reader or if you moved your PDF reader since installation, change the location in the **PDF reader** field. Use the ... button to browse your file system to select the location.

General– Synchronizing the Contents Pane with the Displayed Page

To turn synchronization off, uncheck the item **Keep contents tree synchronized with displayed document**, which is in **Help Preferences**, **General**. Check the item to turn synchronization on. For more information, see “Contents Pane Is Synchronized with Display Pane” on page 4-10.

Help Fonts Preferences – Specifying Font Name, Style, and Size

You can specify the font name, style, and size used in the Help Navigator, and the font used in the display pane. Expand the Help listing in the left pane of the **Preferences** dialog box and select the **F**onts item. The **Help Fonts Preferences** panel appears.



Help Navigator Font

Use **Help Navigator font** preferences to specify the characteristics of the font in the Help Navigator. For example, specify a smaller font size for the Help Navigator to see more information without scrolling.

Select **Use desktop font** if you want the font in the Help Navigator to be the same as that specified under **General - Font & Colors**. If you want the Help Navigator font to be different, select **Use custom font** and specify the font characteristics for the Help Navigator:

- Type, for example, SansSerif
- Style, for example, bold
- Size in points, for example, 12 points

After you make a selection, the **Sample** area shows how the font will look.

HTMLBrowser Font

Specify the font type and size used in the display pane for **Fixed width** and **Proportional width** fonts. In MathWorks documentation, most of the text uses proportional-width fonts. A fixed-width font is used for code examples, function names, and system input and output, as shown in this example.

```
t = 0:pi/20:2*pi;  
y = exp(sin(t));  
plotyy(t,y,t,y,'plot','stem')
```

To easily distinguish code, function names, and system input and output from surrounding text in the documentation, specify a different font for fixed width than for proportional width.

Printing Documentation

You can print the current page displayed in the Help browser, or can print a page to an entire book from the PDF version of the documentation. If you wish to purchase printed documentation, see the online store at the MathWorks Web site at <http://www.mathworks.com>.

Printing a Page from the Help Browser

To print the page currently shown in the Help browser, select the print button



from the display pane toolbar, or select **Print** from the **File** menu. The **Print** dialog box appears. Complete the dialog box and press **OK** to print the page.

Printing the PDF Version of Documentation

If you need to print only a few pages and if the quality does not need to be equivalent to pages in a printed book, you can print directly from the MATLAB Help browser – see “Printing Pages” on page 4-19.

If you need to print more than a few pages of documentation, or if you need the pages to appear as if they came from a printed book, print the PDF version of the documentation. PDF documentation is shown and printed using your PDF reader, Adobe Acrobat Reader. The PDF documentation reproduces the look and feel of the printed book, complete with fonts, graphics, formatting, and images. In the PDF document, use links from the table of contents, index, or within the document to go directly to the page of interest.

To print a PDF version of documentation:

1 For Windows systems only, insert the documentation CD provided with MATLAB into your CD-ROM drive. PDF files are on the CD and are not installed on your system. (For UNIX systems, the PDF files are installed). If you have problems, check the Help preferences – see “Documentation Location – Specifying the help Directory” on page 4-22.

2 In the Help browser, go to the **Contents** tab and select the title (first entry) for a product. The Roadmap page opens for that product, providing links to key documentation for that product.

Printing Documentation

3 On the bottom of the Roadmap page, listed under **Printing the Documentation**, is a link for printing. Click that link.

If there is only one manual for the product, Acrobat Reader opens, displaying the table of contents and first page of the manual.

If there is more than one item you can print for the product, a page listing the choices appears. Select the item you want to print. Acrobat Reader opens, displaying the documentation.

If you have problems, check the Help preferences – see “PDF Reader – Specifying Its Location” on page 4-23.

4 To print the documentation, select **Print** from the **File** menu in Acrobat.

Using Help Functions

There are several help functions that provide different forms of help than the Help browser, or provide alternative ways to access help.

Function Description

doc Displays the reference page for the specified function in the Help browser, providing syntax, a description, examples, and links to related functions.

docopt

help

helpbrowser

helpdesk

helpwin

lookfor

web

For UNIX platforms that do not support Java GUIs, use docopt to specify the location of help files.

Displays M-file help (a description and syntax) in the Command Window for the specified function.

Opens the Help browser, the MATLAB interface for accessing documentation.

Opens the Help browser. In previous releases, helpdesk displayed the Help Desk, which was the precursor to the Help browser. In a future release, thehelpdesk function will be phased out.

Displays in the Help browser a list of all functions, providing access to M-file help for the functions.

Displays in the Command Window a list and brief description for all functions whose brief description includes the specified keyword.

Opens the specified URL in the specified Web browser, with the default being the MATLAB Help browser. You can use theweb function in your own M-files to display documentation.

Using Help Functions

Viewing Function Reference Pages – the doc Function

To view the reference page for a function in the Help browser, use doc. This is like using the Help browser search feature, with **Search type** set to **Function name**. For example, type

doc format

to view the reference page for theformat function.

Overloaded Functions

When a function name is used in multiple products, it is said to be an overloaded function.

The doc function displays the reference page for the first function with that name found on the path, and lists the overloaded functions

in the Command Window. To get help for an overloaded function, specify the name of the directory containing the function you want the reference page for, followed by the function

name. For example, to display the reference page for the set function in the Database Toolbox, type

```
doc database/set
```

Getting Help in the Command Window with the help Function

To quickly view a brief description and syntax for a function in the Command Window, use the help function. For example, typing

```
help bar
```

displays a description and syntax for the bar function in the Command Window. This is called the M-file help. For other arguments you can supply, see the reference page for help.

If you need more information than the help function provides, use the doc function, which displays the reference page in the Help browser. It can include color, images, links, and more extensive examples than the M-file help. For example, typing

```
doc bar
```

displays the reference page for the bar function in the Help browser.

Note M-file help displayed in the Command Window uses all uppercase characters for the function and variable names to make them stand out from the rest of the text. When typing function names, however, use lowercase characters. Some functions for interfacing to Java do use mixed case; the M-file help accurately reflects that and you should use mixed case when typing them.

Overloaded Functions

When a function name is used in multiple products, it is said to be an overloaded function. The help function displays M-file help for the first function with that name found on the path, and lists the overloaded functions at the end. To get help for an overloaded function, specify the name of the directory containing the function you want help for, followed by the function name. For example, to get help for the set function in the Database Toolbox, type

```
help database/set
```

Creating M-File Help for Your Own M-Files

You can create M-file help for your own M-files and access it using the help command. See the help reference page for details.

Other Methods for Getting Help

In addition to using the Help browser and help functions, these are the other ways to get help for MathWorks products:

- “Product-Specific Help Features” on page 4-31
- “Running Demos” on page 4-31
- “Contacting Technical Support” on page 4-32
- “Providing Feedback” on page 4-32
- “Getting Version and License Information” on page 4-32
- “Accessing Documentation for Other Products” on page 4-33
- “Participating in the Newsgroup for MathWorks Products” on page 4-33

Product-Specific Help Features

In addition to the Help browser and help functions, some products and tools allow other methods for getting help. You will encounter some methods in the course of using a product, such as entries in the **Help** menu, **Help** buttons in dialog boxes, and selecting **Help** from a context menu. These methods all display context-sensitive help in the Help browser. Other methods for getting help, such as pressing the **F 1** key, are described in the documentation for the product or tool that uses the method.

Running Demos

Many products include demos that show the key features. It's often helpful to run demos when you first use a product. To see a list of the demos available for a product, use the Launch Pad, and then select a demo to run it. Some products also provide access to their demos on the **Help** menu.

Contacting Technical Support

If your computer is connected to the Internet, you can contact MathWorks Technical Support for help with product problems.

- Find specific Technical Support information using the Help browser **Search** feature, with the **Search type** set to **Online Knowledge Base**. The knowledge base provides the most up-to-date solutions for questions users pose.
- Select **Technical Support Knowledge Base** from the **Web** menu to go the Technical Support Web page. The page displays in your system's default Web browser. You can find out about other types of information including third-party books, ask questions, make suggestions, and report possible bugs.

Providing Feedback

To report any problems or provide any comments or suggestions to The MathWorks about the documentation and help features, send e-mail to doc@mathworks.com.

Alternatively, you can fill out a form on the Web. To access the form, go to the **Contents** pane in the Help browser. Open the last entry, **Support and Web Services**, and select **Feedback on Help**.

To suggest enhancements for or provide feedback about MathWorks products, send e-mail to tosuggest@mathworks.com. To report problems, send e-mail to bugs@mathworks.com or contact Technical Support.

Getting Version and License Information

If you need the product version or license information, select **About** from the **Help** menu for that product. The version is displayed in an **About** dialog box. Click **Show License** in the dialog box to view license information. Note that the information displayed does not cover your specific license agreement. If the product does not have a **Help** menu, use the **ver** function. To see the

license number for MATLAB, type license in the Command Window. See also the ever and version functions.

Accessing Documentation for Other Products

The Help browser provides access to documentation for all products installed on your system. If you want to look through documentation for MathWorks products you don't have, you can:

- View any product's online documentation at the MathWorks Web site, <http://www.mathworks.com>. Use Access login. If you are not an Access member, select **Membership** from the **Web** menu and follow the instructions to join.
- You can access documentation for all products from the documentation CD provided with MATLAB. It contains PDF files for all products. Use preferences for the Help browser to set the documentation location to the CD-ROM drive and use the product filter to specify those products whose documentation you want to see. For instructions, see "Preferences for the Help Browser" on page 4-21.

Participating in the Newsgroup for Math Works Products

The USENET newsgroup for MATLAB and related products, comp.soft-sys.matlab, is read by thousands of users worldwide. Access the newsgroup to ask or provide help or advice, and to share code or examples. You can view and search through a sizable archive of postings from the comp.soft-sys.matlab link on the Technical Support Web page. Select **Technical Support Knowledge Base** from the **Web** menu.

5

Workspace, Search Path, and File Operations

MATLAB Workspace	5-3
Workspace Browser.....	5-3
Viewing and Editing Workspace Variables Using the Array Editor	5-10
 Search Path	5-14
How the Search Path Works	5-14
Viewing and Setting the Search Path	5-15
 File Operations	5-20
Current Directory Field.....	5-20
Current Directory Browser	5-20
Viewing and Making Changes to Directories	5-22
Creating, Renaming, Copying, and Removing Directories and Files	5-23
Opening, Running, and Viewing the Content of Files.....	5-26
Finding and Replacing Content Within Files.....	5-28
Preferences for the Current Directory Browser	5-30
When you work with MATLAB, you'll need to understand these important aspects:	

- Workspace – The workspace is the set of variables maintained in memory during a MATLAB session. Use the Workspace browser or equivalent functions to view the workspace.
- Search path – MATLAB uses a search path to find M-files and other MATLAB related files. Use the **Set Path** dialog box or equivalent functions to view and change the path.
- File operations – To search for, view, open, and make changes to MATLAB related directories and files, use the MATLAB Current Directory browser or equivalent functions.

MATLAB Workspace

The MATLAB workspace consists of the set of variables (named arrays) built up during a MATLAB session and stored in memory. You add variables to the workspace by using functions, running M-files, and loading saved workspaces. For example, if you type

```
t = 0:pi/4:2*pi;
y = sin(t);
```

the workspace includes two variables, *y* and *t*, each having nine values.

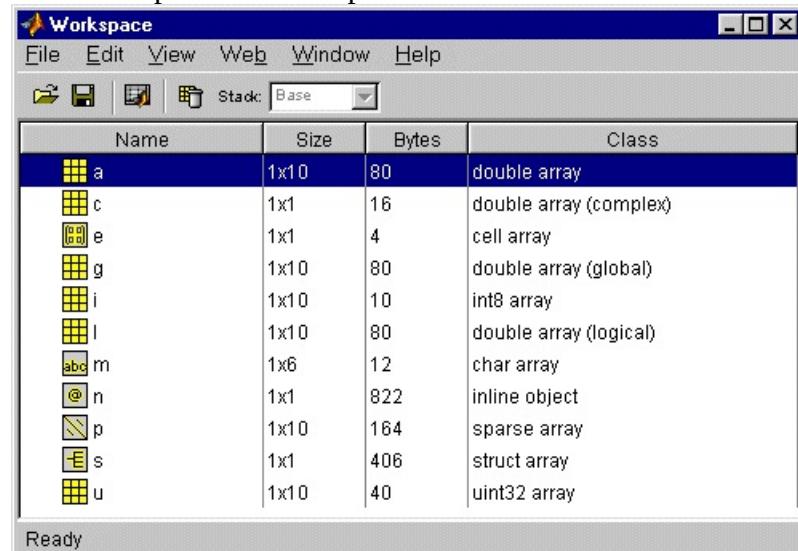
Workspace Browser

Use the Workspace browser to perform operations on the MATLAB workspace. Equivalent functions are available and are documented for each feature of the Workspace browser.

To open the Workspace browser, do one of the following:

- From the **View** menu in the MATLAB desktop, select **Workspace**.
- In the Launch Pad, under MATLAB, double-click **Workspace**.
- Type `workspace` at the Command Window prompt.

The Workspace browser opens.



The screenshot shows the MATLAB Workspace browser window. The window title is "Workspace". The menu bar includes File, Edit, View, Web, Window, and Help. Below the menu is a toolbar with icons for New, Open, Save, and Stack. A dropdown menu shows "Stack" and "Base". The main area is a table listing workspace variables:

Name	Size	Bytes	Class
a	1x10	80	double array
c	1x1	16	double array (complex)
e	1x1	4	cell array
g	1x10	80	double array (global)
i	1x10	10	int8 array
l	1x10	80	double array (logical)
m	1x6	12	char array
n	1x1	822	inline object
p	1x10	164	sparse array
s	1x1	406	struct array
u	1x10	40	uint32 array

Workspace operations you can perform from the Workspace browser or with functions are:

- “Viewing the Current Workspace” on page 5-4

- “Saving the Current Workspace” on page 5-5
- “Loading a Saved Workspace” on page 5-7
- “Clearing Workspace Variables” on page 5-8
- “Viewing Base and Function Workspaces Using the Stack” on page 5-8
- “Creating Graphics from the Workspace Browser” on page 5-9
- “Viewing and Editing Workspace Variables Using the Array Editor” on page 5-10

You can also set preferences – see “Preferences for the Workspace Browser” on page 5-9.

Viewing the Current Workspace

The Workspace browser shows the name of each variable, its array size, its size in bytes, and the class. The icon for each variable denotes its class. To resize the columns of information, drag the column header borders. To show or hide any of the columns, or to specify the sort order, select **Workspace View Options** from the **View** menu.

You can select the column on which to sort in the Workspace browser as well as reverse the sort order of any column. Click on a column heading to sort on that column. Click on the column heading again to reverse the sort order in that column. For example, to sort on **Size**, click the column heading once. To change from ascending to descending, click on the heading again.

Function Alternative. Use `who` to list the current workspace variables. Use `whos` to list the variables and information about their size and class. For example:

```
who
```

Your variables are:

```
A M S v
```

```
whos
```

```
Name Size Bytes Class A 4x4 128 double array M 8x1 2368 cell array S 1x1 398 struct array v
5x9 90 char array Grand total is 286 elements using 2984 bytes
```

Use `exist` function to see if the specified variable is in the workspace.

Saving the Current Workspace

The workspace is not maintained across MATLAB sessions. When you quit MATLAB, the workspace is cleared. You can save any or all of the variables in the current workspace to a MAT-file, which is a MATLAB specific binary file. You can then load the MAT-file at a later time during the current or another session to reuse the workspace variables. MAT-files use a `.mat` extension. Note that the `.mat` extension is also used by Microsoft Access.

Saving All Variables. To save all of the workspace variables using the Workspace browser:

- 1 From the **File** or context menu, select **Save Workspace As**, or click the save button  in the Workspace browser toolbar.

The **Save** dialog box opens.

- 2 Specify the location and **File name**. MATLAB automatically supplies the `.mat` extension.

- 3 Click **Save**.

The workspace variables are saved under the MAT-file name you specified. You can also save the workspace variables from the Command Window by selecting **Save Workspace As** from the

File menu.

Saving Selected Variables. To save some but not all of the current workspace variables:

1 Select the variable in the Workspace browser. To select multiple variables, **Shift** +click or **Ctrl**+click.

2 Right-click and from the context menu, select **Save Selection As**.

The **Save to MAT-File** dialog box opens.

3 Specify the location and **File name**. MATLAB automatically supplies the .mat extension.

4 Click **Save**.

The workspace variables are saved under the MAT-file name you specified.

Function Alternative. To save workspace variables, use the **save** function followed by the filename you want to save to. For example,

```
save('june10')
```

saves all current workspace variables to the file `june10.mat`.

If you don't specify a filename, the workspace is saved to `matlab.mat` in the current working directory. You can specify which variables to save, as well as control the format in which the data is stored, such as `ascii`. For these and other forms of the function, see the reference page `forsave`. MATLAB provides additional functions for saving information – see Chapter 6, “Importing and Exporting Data.”

Loading a Saved Workspace

To load a workspace that you previously saved:

1 Click the load data button

 on the toolbar in the Workspace browser, or right-click in the Workspace browser and select **Import Data** from the context menu.

The **Open** dialog box opens.

2 Select the MAT-file you want to load and click **Open**.

The variables and their values, as stored in the MAT-file, are loaded into the workspace.

You can also load a saved workspace using the Import Wizard. Select **Import Data** from the **File** menu and then select a file from the Import dialog box. The Import Wizard opens. Use the Import Wizard to open the saved workspace. For instructions, see “Using the Import Wizard with Binary Data Files” on page 6-20.

Function Alternative. Use `load` to open a saved workspace. For example, `load('june10')` loads all workspace variables from the file `june10.mat`.

Note If the saved MAT-file `june10` contains the variables `A`, `B`, and `C`, then loading `june10` places the variables `A`, `B`, and `C` back into the workspace. If the variables already exist in the workspace, they are overwritten with the variables from `june10`. For more information, see the reference page `forload`. MATLAB provides other functions for loading information – see Chapter 6, “Importing and Exporting Data.”

Clearing Workspace Variables

You can delete a variable, which removes it from the workspace. To clear a variable using the Workspace browser:

1 In the Workspace browser, select the variable, or **Shift** +click or **Ctrl** +click to select multiple variables. To select all variables, choose **Select All** from the **Edit** or context menus.

2 Do one of the following to clear the selected variables:

- Press the **Delete** key.
- From the **Edit** menu, select **Delete**.

- Click the delete button

 on the toolbar.

- Right-click and select **Delete Selection** from the context menu.

3 A confirmation dialog box may appear. If it does, click **Yes** to clear the variables.

The confirmation dialog box appears if you specify it as a preference. See “Preferences for the Workspace Browser” on page 5-9 to change the preference.

To delete all variables at once, select **Clear Workspace** from the **Edit** menu, or from the context menu in the Workspace browser.

Function Alternative. Use the `clear` function. For example,
`clear A M`
clears the variables `A` and `M` from the workspace.

Viewing Base and Function Workspaces Using the Stack

When you run M-files, MATLAB assigns each function its own workspace, called the function workspace, which is separate from MATLAB’s base workspace. You can access the base and function workspaces when debugging M-files by using the **Stack** field in the Workspace browser. The **Stack** field is only available in debug mode; otherwise it is grayed out. The **Stack** field is also accessible from the Editor/Debugger. See “Debugging M-Files” on page 7-17 for more information.

Creating Graphics from the Workspace Browser

From the Workspace browser, you can generate a graph of a variable. Right-click on the variable you want to graph. From the context menu, select **Graph Selection** and then choose the type of graph you want to create. The graph appears in a figure window. For more information about creating graphs in MATLAB, see MATLAB graphics documentation.

Preferences for the Workspace Browser

You can specify as a preference the fonts to use in the Workspace browser and whether or not you want a confirmation dialog box to appear when you clear variables using the Workspace browser.

From the Workspace browser **File** menu, select **Preferences**. The **Preferences** dialog box opens to the **Workspace Preferences** panel.

Font. Workspace browser font preferences specify the characteristics of the font used in the Workspace browser. Select **Use desktop font** if you want the font in the Workspace browser to be the same as that specified for **General Font & Colors** preferences.

If you want the Workspace browser font to be different, select **Use custom font** and specify the font characteristics for the Workspace browser:

- Type, for example, SansSerif
- Style, for example, bold
- Size in points, for example, 12 points

After you make a selection, the **Sample** area shows how the font will look.

Confirm Deletion of Variables. Check the box for **Confirm deletion of variables** if you want a confirmation dialog box to appear when you delete a variable.

Vie w ing and Editing Workspace Variables Using the Arra y Editor

Use the Array Editor to view and edit a visual representation of one or two-dimensional numeric arrays, strings, and cell arrays of strings. The Array Editor features are:

- “Opening the Array Editor” on page 5-10
- “Changing Values of Elements in the Array Editor” on page 5-12
- “Controlling the Display of Values in the Array Editor” on page 5-12

In addition, you can set preferences. See “Preferences for the Array Editor” on page 5-12.

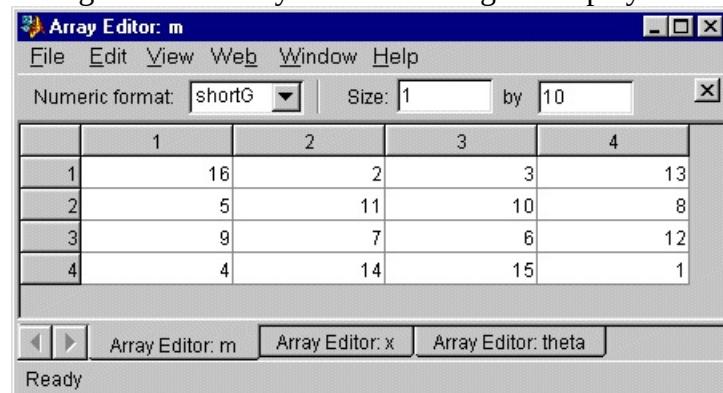
Opening the Array Editor

You can open the Array Editor from the Workspace browser:

- 1 In the Workspace browser, select the variable you want to open. **Shift+click** or **Ctrl+click** to select multiple
- 2 variables to open.
- 3 Click the open selection button  on the toolbar, or right-click and select **Open Selection** from the context menu.

Alternatively, for one variable, you can double-click it to open it. The Array Editor opens, displaying the values for the selected variable. You cannot open an array with 10,000 or more elements.

Changevaluesof arrayelements.Changethedisplay format. Changethedimensionsofthearray.



Use the tabs to view the different

variables you have open in the Array Editor.

Repeat the steps to open additional variables in the Array Editor. Access each variable via its tab at the bottom of the window, or use the **Wind o w** menu.

Function Alternative. To see the contents of a variable in the workspace, just type the variable

name at the Command Window prompt. For example, type

m

and MATLAB returns

```
m =  
16 2 3 13  
5 11 10 8  
9 7 6 12  
4 14 15 1
```

To open a variable in the Array Editor, use `openvar` with the name of the variable you want to open as the argument. For example, type `openvar('m')` MATLAB opens `m` in the Array Editor.

Changing Values of Elements in the Array Editor

In the Array Editor, click in the cell whose value you want to change. Type a new value. Press **Enter** or **Return**, or click in another cell and the change takes effect.

To change the dimensions of an array, type the new values for the rows and columns in the **Size** fields. If you increase the size, the new rows and columns are added to the end and are filled with zeros. If you decrease the size, you will lose data – MATLAB removes rows and columns from the end. Some data types do not allow you to change the dimension; for these variables, the **Size** field is not editable.

If you opened an existing MAT-file and made changes to it using the Array Editor, you'll have to save that MAT-file if you want the changes to be saved. For instructions, see “Saving the Current Workspace” on page 5-5.

Controlling the Display of Values in the Array Editor

In the Array Editor, select an entry in the **Numeric format** list box to control how numeric values are displayed. For descriptions of the formats, see the reference page for `format`. The format applies only to the Array Editor display for that variable for the current session; it does not affect how MATLAB computes or saves the numeric value, nor does it affect the format used for display in the Command Window.

To specify a format for all variables in the Array Editor and keep it persistent across sessions, specify the format for the Array Editor using preferences as discussed in the next topic.

Preferences for the Array Editor

Using preferences for the Array Editor, you can specify the format for how numeric values are displayed, as well as their font type, style, and size. To set preferences for the Array Editor, select **Preferences** from the **File** menu. The **Preferences** dialog box opens showing **Array Editor Preferences**.

Specify the **Font** and **Default format** preferences.

Font. Array Editor font preferences specify the characteristics of the font used in the Array Editor. Select **Use desktop font** if you want the font in the Array Editor to be the same as that specified for **General Font & Colors** preferences. If you want the Array Editor font to be different, select **Use custom font** and specify the font characteristics for the Array Editor:

- Type, for example, SansSerif
- Style, for example, bold
- Size in points, for example, 12 points

After you make a selection, the **Sample** area shows how the font will look.

Default Format. Specify the output format of numeric values displayed in the Array Editor. This affects only how numbers are displayed, not how MATLAB computes or saves them. For more information, see “Controlling the Display of Values in the Array Editor” on page 5-12 or see the reference page for `format`.

Search Path

MATLAB uses a *search path* to find M-files and other MATLAB related files, which are organized in directories on your file system. These files and directories are provided with MATLAB and associated toolboxes. Any file you want to run in MATLAB must reside in a directory that is on the search path or in the current directory. By default, the files supplied with MATLAB and MathWorks toolboxes are included in the search path.

If you create any MATLAB related files, add the directories containing the files to MATLAB’s search path. When you create your own M-files or if you modify any MATLAB supplied M-files, save them in a directory that is *not* in \$matlabroot/toolbox/matlab. If you do keep any of your files in \$matlabroot/toolbox/matlab, they may be overwritten when you install a new version of MATLAB. In addition, you might need to restart MATLAB or use `threhash` function before you use the new or updated file. This is because the locations of those files are loaded and cached in memory at the beginning of each MATLAB session to improve performance, and changes are not always recognized automatically.

For instructions to view and modify the search path, see “Viewing and Setting the Search Path” on page 5-15.

How the Search Path Works

The search path is also referred to as the *MATLAB path*. Files included are considered to be *on the path*. When you include a directory on the search path, you *add it to the path*. Subdirectories must be explicitly added to the path; they are not on the path just because their parent directories are. The search path is stored in the file `pathdef.m`.

The order of directories on the path is relevant. MATLAB looks for a named element, for example, `foo`, as described here. If you enter `foo` at the MATLAB prompt, MATLAB performs the following actions:

- 1 Looks for `foo` as a variable.
- 2 Checks for `foo` as a built-in function.
- 3 Looks in the current directory for a file named `foo.m`.
- 4 Searches the directories on the MATLAB search path, in order, for `foo.m`. Although the actual search rules are more complicated because of the restricted scope of private functions,

subfunctions, and object-oriented functions, this simplified perspective is accurate for the ordinary M-files you usually work with.

The order of the directories on the search path is important if there is more than one function with the same name. When MATLAB looks for that function, only the first one in the search path order is found; other functions with the same name are considered to be *shadowed* and cannot be executed. For more information, see “How MATLAB Determines Which Method to Call”.

To see the pathname used, use `which` for a specified function. For more information, see the reference page for `which`.

Viewing and Setting the Search Path

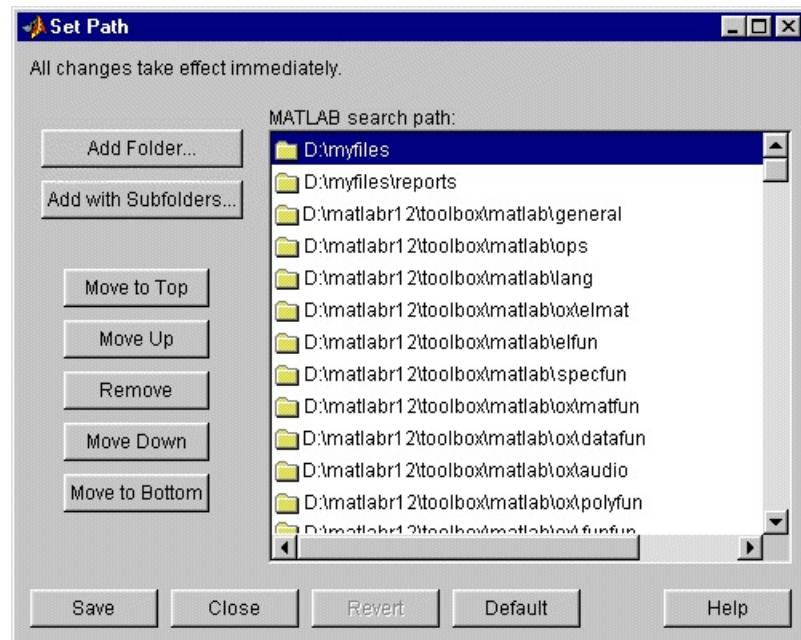
Use the **Set Path** dialog box to view and modify MATLAB’s search path and see files in directories that are on the path. Equivalent functions are documented for each feature of the **Set Path** dialog box.

Select **Set Path** from the **File** menu, or type `path` at the Command Window prompt. The **Set Path** dialog box opens.

When you click one of these buttons, the change is made to the current search Directories on the current MATLAB search path. However, the search path is not automatically saved for future sessions.

Make changes to the search path.

Save changes for use in future MATLAB sessions.



Use the Set Path dialog box for the following:

- “Viewing the Search Path” on page 5-17
- “Adding Directories to the Search Path” on page 5-17
- “Moving Directories within the Search Path” on page 5-17
- “Removing Directories from the Search Path” on page 5-18
- “Restoring the Default Search Path” on page 5-18
- “Reverting to the Previous Path” on page 5-18
- “Saving Settings to the Path” on page 5-18

Viewing the Search Path

The **MATLAB search path** field in the **Set Path** dialog box lists all of the directories on the search path.

Function Alternative. Use the `path` function to view the search path.

Adding Directories to the Search Path

To add directories to the MATLAB search path using the **Set Path** dialog box:

- 1 Click the **Add Folder** or the **Add with Subfolders** button.
 - If you want to add only the selected directory but do not want to add all of its subdirectories, click **Add Folder**.
 - If you want to add the selected directory and all of its subdirectories, click **Add with Subfolders**.

The **Browse for Folder** dialog box opens.

- 2 In the **Browse for Folder** dialog box, use the view of your file system to select the directory to add, and then click **OK**.

The selected directory, and subdirectories if specified, are added to the front (top) of the search path. They remain on the search path until you end the current MATLAB session. To use the newly modified search path in subsequent sessions, you need to save the path – see “Saving Settings to the Path” on page 5-18.

You cannot add method directories (directories that start with @) or private directories directly to the search path. Instead, add their parent directories.

Function Equivalent. To add directories to the search path, use `addpath`. The `addpath` function offers an option to get the path as a string and to concatenate multiple strings to form a new path.

You can include `addpath` in your startup M-file to automatically modify the path when MATLAB starts.

Moving Directories within the Search Path

The order of files on the search path is relevant – for more information, see “How the Search Path Works” on page 5-14.

To modify the order of directories within the search path, first select the directory you want to move. Then select one of the **Move** buttons, such as **Move to Top**. The top of the list corresponds to the front of the search path and the bottom of the list corresponds to the end of the search path.

The new order of files on the search path remains in effect until you end the current MATLAB session. To use the newly modified search path in subsequent sessions, you need to save the path – see “Saving Settings to the Path” on page 5-18.

Removing Directories from the Search Path

To remove directories from the MATLAB search path using the **Set Path** dialog box:

- 1 Select the directory to remove.
- 2 Click **Remove**.

The directory is removed from the search path for the remainder of the current MATLAB session. To use the newly modified search path in subsequent sessions, you need to save the path – see “Saving Settings to the Path” on page 5-18.

Function Equivalent. To remove directories from the search path, `usermpath`. You can include `dermpath` functions in your startup M-file to automatically modify the path when MATLAB starts.

Restoring the Default Search Path

To restore the default search path, click **Default** in the **Set Path** dialog box. This changes the search path so that it uses the factory settings.

Reverting to the Previous Path

To restore the previous path, click **Revert** in the **Set Path** dialog box. This cancels any unsaved changes you’ve made in the **Set Path** dialog box.

Saving Settings to the Path

When you make changes to the search path, they remain in effect during the current MATLAB session. To keep the changes in effect for subsequent sessions, save the changes. To save changes using the **Set Path** dialog box, click **Save**.

The search path is stored in `thepathdef.m` file. By default, `pathdef.m` is stored in `$matlabroot\toolbox\local`. On Windows platforms, you can use a different `pathdef.m` if you store it in your startup directory – see “Startup Directory for MATLAB” on page 1-3.

You can directly edit `pathdef.m` with a text editor to change the path. On UNIX workstations you may not have file system permission to edit `pathdef.m`. In this case, `putpath` and `addpath` functions in your startup M-file to change your path defaults.

File Operations

MATLAB file operations use the current directory as a reference point. Any file you want to run must either be in the current directory or on the search path. Also, when you open a file in MATLAB, the starting point for the file open dialog box is the current directory. The keys tools for performing file operations are:

- Current directory field
- Current directory browser

Current Directory Field

A quick way to view or change the current directory is by using the **Current Directory** field in the desktop toolbar.

Current Directory: D:\mymfiles To change the current directory from this field, do one of the following:

- In the field, type the path for the new current directory.
- Click the down arrow to view a list of previous working directories, and select an item from the list to make that directory become MATLAB's current working directory. The directories are listed in order, with the most recently used at the top of the list. You can clear the list and set the number of directories saved in the list – see “Preferences for the Current Directory Browser” on page 5-30.
- Click the browse button (...) to set a new current directory.

Current Directory Browser

To search for, view, open, and make changes to MATLAB related directories and files, use the MATLAB Current Directory browser. Equivalent functions are documented for each feature of the Current Directory browser.

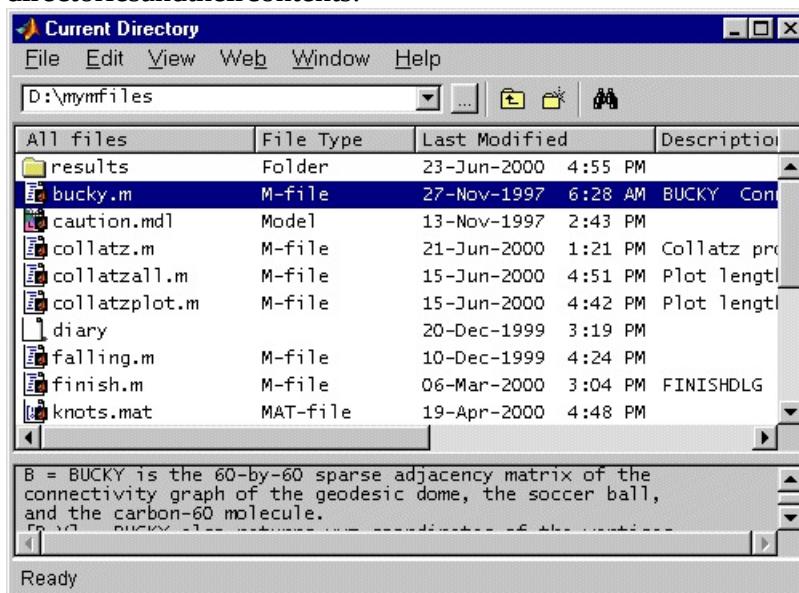
To open the Current Directory browser, select **Current Directory** from the **View** menu in the MATLAB desktop, or type `filebrowser` at the Command

Double-click a file to open it in an appropriate tool.

View the help portion of the selected M-file.

Window prompt. You can also open it from the Launch Pad, under MATLAB. The Current Directory browser opens.

Use the path name edit box to view. Click the find button to search for content within M-files, directories and their contents.



The main file operations you can perform using the Current Directory browser are:

- “Viewing and Making Changes to Directories” on page 5-22
- “Creating, Renaming, Copying, and Removing Directories and Files” on page 5-23

- “Opening, Running, and Viewing the Content of Files” on page 5-26
- “Finding and Replacing Content Within Files” on page 5-28 You can also set preferences – see “Preferences for the Current Directory Browser” on page 5-30.

Vie w ing and M aking Changes to Directories

The ways to view and make changes to directories are:

- “Changing the Current Working Directory and Viewing Its Contents” on page 5-22
- “Adding Directories to the MATLAB Search Path” on page 5-23
- “Changing the Display” on page 5-23

Changing the Current Working Directory and Viewing Its Contents To change the current directory, type the directory name in the pathname edit box in the Current Directory browser, and press the **Enter** or **Return** key. That directory becomes the current working directory and the files and subdirectories in it are listed.

To view a directory that has recently been displayed, click the down arrow at the right side of the pathname edit box in the Current Directory browser. The previously displayed directories are listed, sorted by most recent to least recent. Select an entry to view the contents of that directory. You can clear the list and set the number of directories saved in the list – see “Preferences for the Current Directory Browser” on page 5-30.

To view the contents of a subdirectory within the directory being displayed, double-click the subdirectory in the Current Directory browser, or select the subdirectory and press the **Enter** or **Return** key. You can also right-click on that subdirectory and select **Open** from the context menu.

To move up one level in the directory structure, click the up button  in the Current Directory browser toolbar, or press the **Back Space** key.

Function Alternative. Usedir to view the contents of the current working directory or another specified directory.

Use what to see only the MATLAB related files in a directory. With no arguments,what displays the MATLAB related files in the current working directory. Usewhich to display the pathname for the specified function. Use exist to see if a directory or file exists.

Adding Directories to the MATLAB Search Path

From the Current Directory browser, you can add directories to the MATLAB search path. Right-click and from the context menu, select **Add to P ath**. Then select one of the options:

- **Current Directory** – Adds the current directory to the path.
- **Selected Folders**– Adds the directory selected in the Current Directory browser to the path.
- **Selected Folder and Subfolders** – Adds the directory selected in the Current Directory browser to the path, and adds all of its subdirectories to the path.

Changing the Display

To specify the types of files shown in the Current Directory browser, use **Vie w -> Current Directory Filter**. For example, you can show only M-files.

You can sort the information shown in the Current Directory browser by column. Click the title

of column on which you want to sort. The display is sorted, with the information in the that column shown in ascending order. Click a second time on the column title to sort the information in descending order.

If you make changes to the current directory from your file system, the Current Directory browser display won't immediately reflect those changes. Select **Refresh** from the context menu to update the Current Directory browser display.

Creating, Renaming, Copying, and Removing Directories and Files

If you have write permission, you can create, copy, remove, and rename MATLAB related files and directories for the directory shown in the Current Directory browser. If you do not have write permission, you can still copy files and directories to another directory.

Creating New Files

To create a new file in the current directory:

1 Select **New** from the context menu or **File** menu and then select the type of file to create.

An icon for that file type, for example, an M-file icon

 with the default name Untitled appears at the end of the list of files shown in the Current Directory browser.

2 Type over Untitled with the name you want to give to the new file. **3** Press the **Enter** or **Return** key.

The file is added.

4 To enter the contents of the new M-file, open the file – see “Opening, Running, and Viewing the Content of Files” on page 5-26. If you created the file using the context menu, the new file opens with a template for writing an M-file function.

Create New Directories

To create a new directory in the current directory:

1 Click the new folder button

 in the Current Directory browser toolbar, or select **New -> Folder** from context menu.

An icon, with the default name NewFolder appears at the end of the list of files shown in the Current Directory browser.

2 Type over NewFolder with the name you want to give to the new directory.

3 Press the **Enter** or **Return** key.

The directory is added.

Function Equivalent. To create a directory, use the `mkdir` function. For example,

`..\\testdata newdir`

creates the directory newdir within the directory testdata.

Renaming Files and Directories

To rename a file or directory, select the item, right-click, and select **Rename** from the context menu. Type over the existing name with the new name for the file or directory, and press the **Enter** or **Return** key. The file or directory is renamed.

Cutting or Deleting Files and Directories To cut or delete files and directories:

1 Select the files and directories to remove. Use **Shift+click** or **Ctrl+click** to select multiple items.

2 Right-click and select **Cut** or **Delete** from the context menu. **Cut** is also available from the **Edit** menu.

The files and directories are removed.

Function Equivalent. To delete a file, use the `delete` function. For example,
`delete('d:\mymfiles\testfun.m')`
deletes the file `testfun.m`.

Copying and Pasting Files

You can copy and paste files, but not directories. To copy and paste files:

1 Select the files. Use **Shift+click** or **Ctrl+click** to select multiple items.

2 Right-click and select **Copy** from the context menu, or select **Copy** from the **Edit** menu.

3 Move to the directory where you want to paste the files you just copied or cut.

4 Paste the files by right-clicking and selecting **Paste** from the context menu, or by selecting **Paste** from the **Edit** menu.

Opening, Running, and Viewing the Content of Files

Opening Files

You can open a file using the `open` feature of the Current Directory browser. The file opens in the tool associated with that file type.

To open a file, select one or more files and perform one of the following actions:

- Press the **Enter** or **Return** key.
- Right-click and select **Open** from the context menu.
- Double-click on the file(s).

The files open in the appropriate tools. For example, the Editor/Debugger opens for M-files, and Simulink opens for model (mdl) files.

To open any file in the Editor, no matter what type it is, select **Open as Text** from the context menu.

You can also import data from a file. Select the file, right-click, and select **Import Data** from the context menu. The Import Wizard opens. See Chapter 6, “Importing and Exporting Data” for instructions to import the data.

Function Alternative. Use the `open` function to open a file into the tool appropriate for the file, given its file extension. Default behavior is provided for standard MATLAB file types. You can extend the interface to include other file types and to override the default behavior for the standard files.

File Type Extension Action Figure file fig Open figure in a figure window. HTML file html Open HTML filename in the Help browser. M-file m Open M-filename in the Editor.

MAT-file mat Model mdl P-file p Open MAT-filename in the Import Wizard. Open modelname in Simulink.

Open the corresponding M-file, name.m, if it exists, in the Editor.

File Type Extension Action (Continued) Variable not Open the numeric or string arrayname in the applicable Array Editor; open callsopenvar. Other custom Openname.custom by calling the helper function opencustom, where opencustom is a user-defined function.

To view the content of an ASCII file, such as an M-file, use the type function. For example
type('startup')

displays the contents of the file startup.m in the Command Window.

Running M-Files

To run an M-file from the Current Directory browser, select it, right-click, and select **Run** from the context menu. The results appear in the Command Window.

Viewing Help for an M-File

You can view help for the M-file selected in the Current Directory browser. From the context menu, select **View Help**. The reference page for that function appears in the Help browser, or if a reference page does not exist, the M-file help appears.

You can view the M-file help in the Current Directory browser – for instructions, see “Preferences for the Current Directory Browser” on page 5-30.

Finding and Replacing Content Within Files

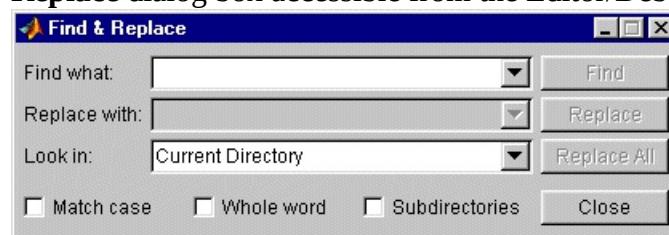
From the Current Directory browser, you can search for a specified string within files. If the file is open in the Editor, you can replace the specified string in a file.

Finding a Specified String Within a File To search for a specified string in files:

1 Click the find button

Min the Current Directory browser toolbar.

The **Find & Replace** dialog box appears. This provides the same features as the **Find & Replace** dialog box accessible from the Editor/Debugger.



2 Complete the **Find & Replace** dialog box

to find all occurrences of the string you specify.

- Type the string in the **Find what** field.

- Select the directories to search through from the **Look in** listbox, or type a directory name directly in this field.

- Constrain the search by checking **Match case** or **Whole word**.

- Check the box for **Subdirectories** if you want the search to also look through the subdirectories.

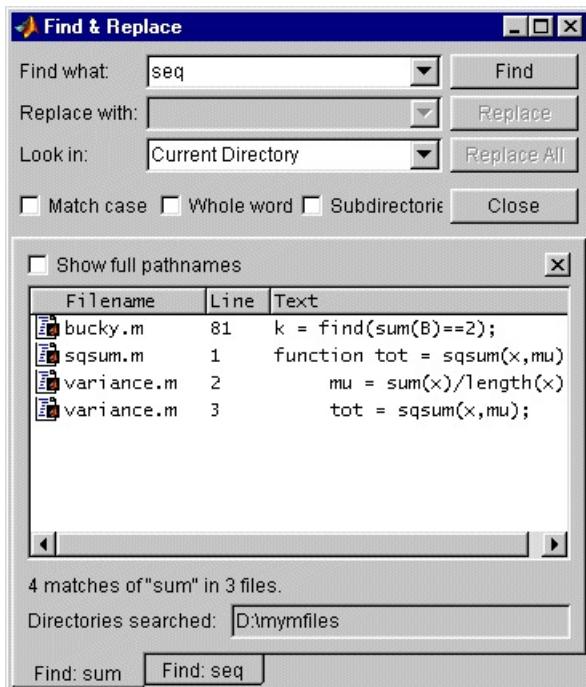
3 Click **Find**.

Results appear in the lower part of the **Find & Replace** dialog box and include the filename, M-

file line number, and content of that line.

Double-click on a file in the results list to open it.

The results of previous find operations are available via tabs.



Close the results section.

4 Open any M-file(s) in the results list by doing one of the following:

- Double-clicking the file(s)
- Selecting the file(s) and pressing the **Enter** or **Return** key
- Right-clicking the file(s) and selecting **Open** from the context menu

The M-file(s) opens in the Editor, scrolled to the line number shown in the results section of the **Find & Replace** dialog box.

5 If you perform another search, the results of each search are accessible via tabs just below the current results list. Click a tab to see that results list as well as the search criteria.

Function Equivalent. Use **lookfor** to search for the specified string in the first line of help in all M-files on the search path.

Replacing a Specified String Within Files

After searching for a string within a file, you can replace the string.

1 Open the file in MATLAB Editor. You can open the file from the Current Directory browser **Find & Replace** dialog box results list – see step 4 in “Finding a Specified String Within a File” on page 5-28. Be sure that the file in which you want to replace the string is the current file in the Editor.

2 In the **Look in** field in the **Find & Replace** dialog box, select the name of the file in which you want to replace the string.

The **Replace** button in the **Find & Replace** dialog box becomes selectable.

3 In the **Replace with** field, type the text that is to replace the specified string.

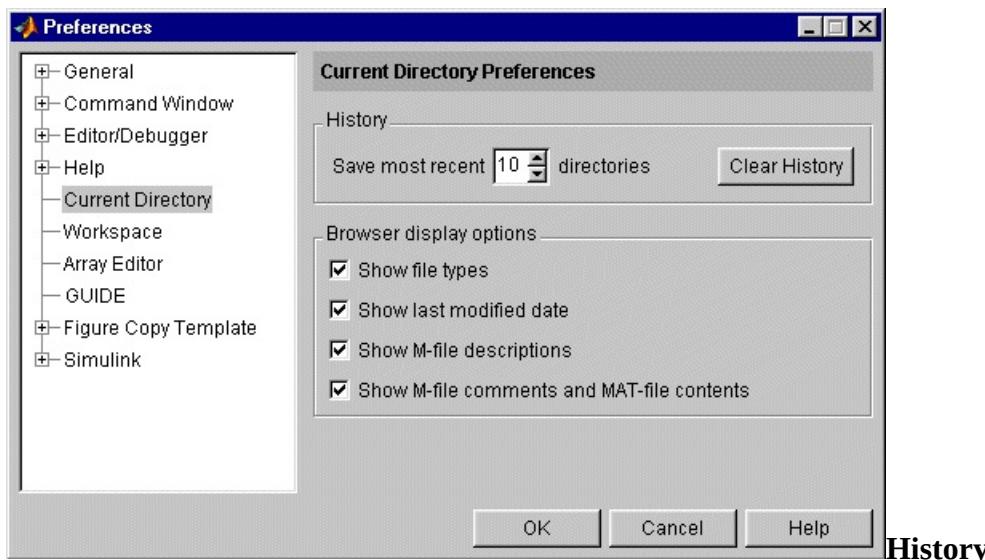
4 Click **Replace** to replace the string in the selected line, or click **Replace All** to replace all instances in the currently open file.

The text is replaced.

5 To save the changes, select **Save** from the **File** menu in the Editor.

Preferences for the Current Directory Browser

You can specify the number of recently used current directories to maintain in the history list as well as the type of information to display in the Current Directory browser using preferences. From the Current Directory browser **File** menu, select **Preferences**. The **Current Directory Preferences** panel appears in the **Preferences** dialog box.



The drop down list in the Current Directory browser toolbar, as well as in the MATLAB desktop Current Directory field, show the most recently used current directories. The list contains all of the current directories used in the current MATLAB session.

Removing Directories. To remove the entries in the list, select **Clear History**. The list is cleared immediately.

Saving Directories. When the MATLAB session ends, the list of directories will be maintained. Use the **Save most recent directories** field to specify how many directories will appear on the list in at the start of the next MATLAB session.

Browser display options

In the Current Directory browser, you can view the file type, last modified date, M-file descriptions, and M-file comments and MAT-file contents by checking the appropriate **Browser display options**.

Importing and Exporting Data

Importing Text Data	6-4
Exporting ASCII Data	6-16
Importing Binary Data	6-20
Exporting Binary Data	6-25
Working with HDF Data	6-29
Using Low-Level File I/O Functions	6-51

MATLAB provides many ways to load data from disk files or the clipboard into the workspace, a process called *importing* data, and to save workspace variables to a disk file, a process called *exporting* data. Your choice of which mechanism to use depends on which operation you are performing, importing or exporting, and the format of the data, text or binary.

Note The easiest way to import data into MATLAB is to use the Import Wizard. When you use the Import Wizard, you do not need to know the format of the data. You simply specify the file that contains the data and the Import Wizard processes the file contents automatically. For more information, see “Using the Import Wizard with Text Data” on page 6-4 and “Using the Import Wizard with Binary Data Files” on page 6-20.

Text Data

In text formats, the data values are American Standard Code for Information Interchange (ASCII) codes that represent alphabetic and numeric characters. ASCII text data can be viewed in a text editor. For more information about working with text data, see:

- “Importing Text Data” on page 6-4
- “Exporting ASCII Data” on page 6-16

Binary Data

In binary format, the values are not ASCII codes and cannot be viewed in a text editor. Binary files contain data that represents images, sounds, and other formats. For more information about working with binary data, see:

- “Importing Binary Data” on page 6-20
- “Exporting Binary Data” on page 6-25

Other Formats

MATLAB also supports the importing of scientific data that uses the Hierarchical Data Format (HDF). See “Working with HDF Data” on page 6-29 for more information.

Low-Level File I/O

MATLAB also supports C-style, low-level I/O functions that you can use with any data format. For more information, see “Using Low-Level File I/O Functions” on page 6-51.

Importing Text Data

The easiest way to import text data into the workspace is to use the MATLAB Import Wizard. You simply start the Import Wizard and specify the file that contains the data you want to import. The Import Wizard can process most numeric data files automatically, even if they contain text headers. See “Using the Import Wizard with Text Data” below for more detailed information.

If you need to work from the MATLAB command line or perform import operations as part of an M-file, you must use one of the MATLAB import functions. Your choice of which function to use depends on the type of data in the file and how the data is formatted. MATLAB has functions that work with numeric data and other functions that can handle both alphabetic and numeric data. See “Using Import Functions with Text Data” on page 6-9 for more information about choosing the function that is right for your data.

Caution When you import data into the MATLAB workspace, you overwrite any existing variable in the workspace with the same name.

Using the Import Wizard with Text Data

To import text data using the Import Wizard, perform these steps:

1 Start the Import Wizard, by selecting the **Import Data** option on the MATLAB **File** menu. MATLAB displays a file selection dialog box. You can also use the `uiimport` function to start the Import Wizard.

To use the Import Wizard to import data from the clipboard, select the **Paste Special** option on the MATLAB **Edit** menu. You can also right-click in the MATLAB command window and choose **Paste Special** from the context menu. Skip to step 3 to continue importing from the clipboard.

2 Specify the file you want to import in the file selection dialog box and click **Open**. The Import Wizard opens the file and attempts to process its contents.

3 Specify the character used to separate the individual data items. This character is called the delimiter or column-separator. The Import Wizard can determine the delimiter used in many cases. However, you may need to specify the character used in your text file. See “Specifying the Delimiter” on page 6-5 for more information. Once the Import Wizard has correctly processed the data, click **Next**.

4 Select the variables that you want to import. By default, the Import Wizard puts all the numeric data in one variable and all the text data in other variables but you can choose other options. See “Selecting the Variables to Import” on page 6-7 for more information.

5 Click **Finish** to import the data into the workspace.

Specifying the Delimiter

When the Import Wizard opens a text file, or copies data from the clipboard, it displays a portion of the raw data in the Preview pane of the dialog box. You can use this display to verify that the file contains the data you expected. The Import Wizard also attempts to process the data, identifying the delimiter used in the data. The Import Wizard displays the variables it has created based on its interpretation of the delimiter, using tabbed panels to display multiple variables.

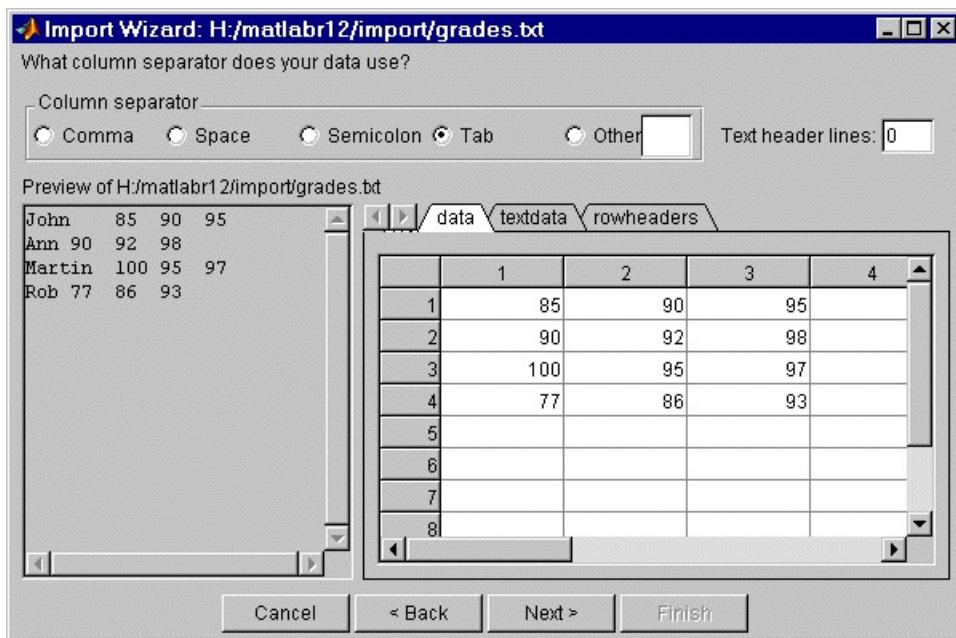
For example, in the following figure, the Import Wizard has opened this sample file,grades.txt.

John 85 90 95
Ann 90 92 98
Martin 100 95 97
Rob 77 86 93

Numberoflines ofheader text ignored

Delimiterfound infile
Previewof the datainthe file.

Previewof the variables the Import Wizard creates.



In the figure, note how the Import Wizard has correctly identified the tab character as the delimiter used in the file and has created three variables from the data:

- data -- containing all the numeric data in the file
- textdata -- containing all the text found in the file
- rowheaders -- containing the names in the left-most column of data.

Handling Alphabetic Data. The Import Wizard recognize data files that use row or column headers and extract these headers into separate variables. It can also ignore any text header lines that may precede the data in a file.

Specifying Other Delimiters. If the Import Wizard cannot determine the delimiter used in the data, it displays a preview of the raw data, as before, but the variables it displays will not be correct. If your data uses a character other than a comma, space, tab, or semicolon as a delimiter, you must specify it by clicking the **Other** button and entering the character in the text box. The

Import Wizard immediately reprocesses the data, displaying the new variables it creates.

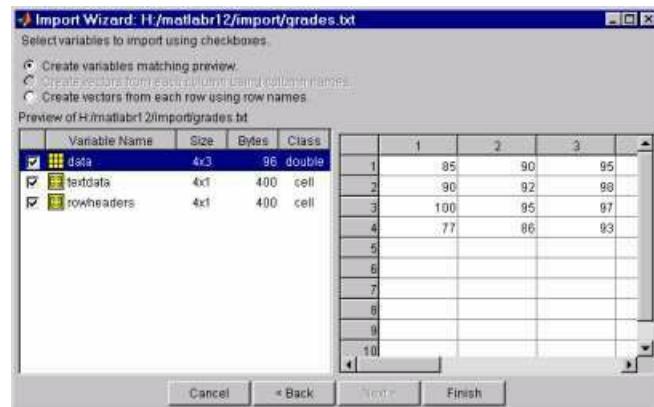
Selecting the Variables to Import

The Import Wizard displays a list of the variables it has created from your data. You can select which variables you want to import by clicking in the check box next to its name. By default, all variables are selected.

The Import Wizard displays the contents of the variable that is highlighted in the list in the right pane of the dialog box. To view the contents of one of the other variables, click on it. Choose the variables you want to import and click **Next**.

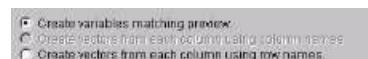
List of variables to be imported.

Import Wizard displays content of the variable highlighted in the list.



Changing the Variable Selection. By default, the Import Wizard puts all the numeric data in the file into one variable. If the file contains text data, the Import Wizard puts it in a separate variable. If the file contains row- or column-headers, the Import Wizard puts them in a separate variables, called **rowheaders** or **colheaders**, respectively.

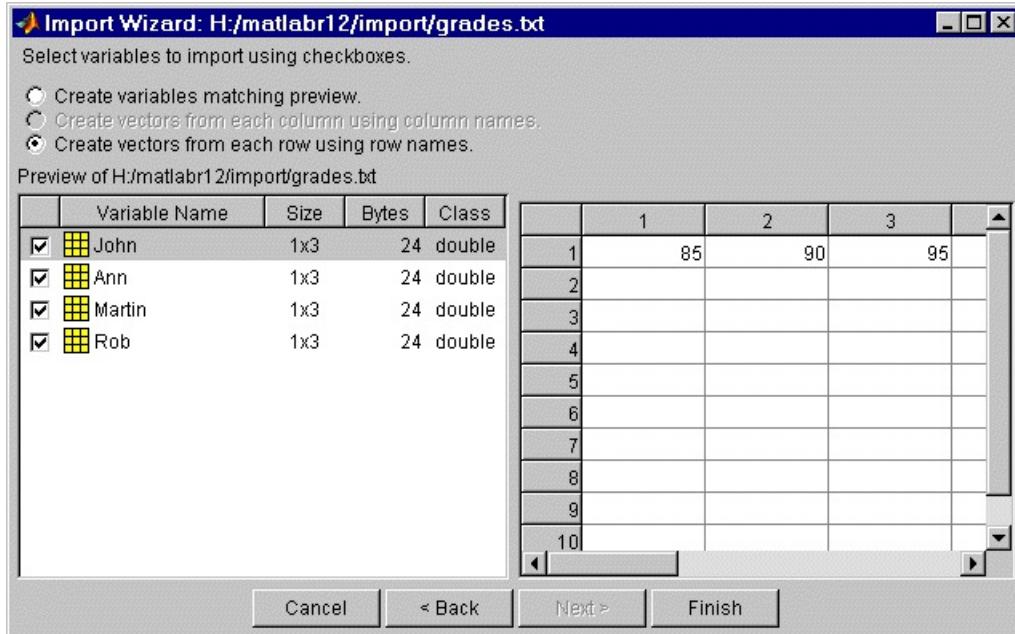
In some cases, it might be more convenient to create variables out of each row or column of data and use the row headers or column header text as the name of each variable. To do this, click the appropriate button from the list of buttons at the top of the dialog box.



For example, it would ease calculations of the student averages if we created separate variables for each student that contained that student's grades. To create these variables, click the **Create variables from each column using row names** button. When you click this option, the Import Wizard reprocesses the file creating these new variables.

Select option to create

variables from row header.
List contains variables by
row header.
Variable is a vector made
from row of data file.



When you are satisfied with the list of variables to be imported, click **Next** to bring the data into the MATLAB workspace. This button also dismisses the Import Wizard. The Import Wizard displays a message in the MATLAB command window, reporting that it created variables in the workspace. In the following example, note how the numeric text data in each variable is imported as an array of doubles.

Import Wizard created variables in the current workspace. `>> whos`

Name	Size	Bytes	Class
Ann	1x3	24	double array
John	1x3	24	double array
Martin	1x3	24	double array
Rob	1x3	24	double array

Grand total is 12 elements using 96 bytes

Using Import Functions with Text Data

To import text data from the command line or in an M-file, you must use one of the MATLAB import functions. Your choice of function depends on how the data in the text file is formatted.

The text data must be formatted in a uniform pattern of rows and columns, using a text character, called a *delimiter* or *column separator*, to separate each data item. The delimiter can be space, comma, semicolon, tab, or any other character. The individual data items can be alphabetic or numeric characters or a mix of both.

The text file may also contain one or more lines of text, called *header lines*, or may use text headers to label each column or row. The following example illustrates a tab-delimited text file with header text and row and column headers.

Textheader line

ColumnHeaders

RowHeaders

Class Grades for Spring Term Grade1 Grade2 Grade3

John 85 90 95

Ann 90 92 98

Martin 100 95 97

Rob 77 86 93 Tab-delimiteddata

To find out how your data is formatted, view it in a text editor. After you determine the format, scan the data format samples in Table 6-1 and look for the sample that most closely resembles the format of your data. Read the topic referred to in the table for more information.

Table 6- 1: ASCII Data File Formats and MATLAB Import Commands Data Format

Sample File

Extension Description

1 2 3 4 5

6 7 8 9 10 .txt

.dat

or other See “Importing Numeric Text Data” on page 6-11 for more information. You can also use the Import Wizard for this data format. See “Using the Import Wizard with Text Data” on page 6-4 for more information.

1; 2; 3; 4; 5

6; 7; 8; 9; 10 or

1, 2, 3, 4, 5

6, 7, 8, 9, 10 .txt

.dat

.csv

or other

Ann Type1 12.34 45 Yes Joe Type2 45.67 67 No .txt

.dat

or other See “Importing Delimited ASCII Data Files” on page 6-12 for more information. You can also use the Import Wizard for this data format. See “Using the Import Wizard with Text Data” on page 6-4 for more information.

See “Importing Numeric Data with Text Headers” on page 6-13 for more information.

Grade1 Grade2 Grade3

91.5 89.2 77.3

88.0 67.8 91.0

67.3 78.1 92.5 .txt

.dat

or other See “Importing Numeric Data with Text Headers” on page 6-13 for more information. You can also use the Import Wizard for this data format. See “Using the Import Wizard with Text Data” on page 6-4 for more information.

If you are familiar with MATLAB import functions but not sure when to use them, view Table 6-2 which compares the features of each function.

Table 6- 2: ASCII Data Import Function Feature Comparison

Function Data Type Delimiters Number of Return

Values

Notes

csvread Numeric data Only

commas One

dlmread Numeric data

fscanf

load

Alphabetic and numeric;
however, both types returned in a single
return variable

Numeric data Any
character

Any
character

One Primarily used with spreadsheet data. See also the binary format spreadsheet import functions.

Flexible and easy to use.
One Part of low-level file I/O

routines. Requires use of fopen to obtain file
identifier and fclose after read.

Only spaces One
textread Alphabetic and

numeric

Any

character Multiple return

values. Easy to use. Use the functional form ofload to specify the name of the output variable.

Flexible, powerful, and easy to use. Use format string to specify
conversions.

Importing Numeric Text Data

If your data file contains only numeric data, you can use many of the MATLAB import functions (listed in Table 6-2), depending on how the data is delimited. If the data is rectangular, that is, each row has the same number of elements, the simplest command to use is the load command. (The load command can also be used to import MAT-files, MATLAB's binary format for saving the workspace.)

For example, the file namedmy_data.txt contains two rows of numbers delimited by space characters.

```
1 2 3 4 5
```

```
6 7 8 9 10
```

When you use load as a command, it imports the data and creates a variable in the workspace with the same name as the filename, minus the file extension.

```
load my_data.txt;
```

```
whos
```

Name	Size	Bytes	Class
------	------	-------	-------

my_data	2x5	80	double array
---------	-----	----	--------------

my_data

```
my_data =
```

```
1 2 3 4 5
```

```
6 7 8 9 10
```

If you want to name the workspace variable something other than the file name, use the functional form ofload. In the following example, the data from my_data.txt is loaded into the workspace variable A.

```
A = load('my_data.txt');
```

Importing Delimited ASCII Data Files

If your data file uses a character other than a space as a delimiter, you have a choice of several import functions you can use. (See Table 6-4 for a complete list.) The simplest to use is the dlmread function.

For example, consider a file namedph.dat whose contents are separated by semicolons.

```
7.2;8.5;6.2;6.6
```

```
5.4;9.2;8.1;7.2
```

To read the entire contents of this file into an array namedA, enter A = dlmread('ph.dat', ';');

You specify the delimiter used in the data file as the second argument to dlmread. Note that, even though the last items in each row are not followed by a delimiter, dlmread can still process the file correctly. dlmread ignores space characters between data elements. So, the preceding dlmread command works even if the contents ofph.dat are

```
7.2; 8.5; 6.2;6.6
```

```
5.4; 9.2 ;8.1;7.2
```

Importing Numeric Data with Text Headers

To import an ASCII data file that contains text headers, use the `textread` function, specifying the `headerlines` parameter. `textread` accepts a set of predefined parameters that control various aspects of the conversion. (For a complete list of these parameters, see the `textread` reference page.) The `headerlines` parameter lets you specify the number of lines at the head of the file that `textread` should ignore.

For example, the file, `grades.dat`, contains formatted numeric data with a one-line text header.

```
Grade1 Grade2 Grade3
78.8 55.9 45.9
99.5 66.8 78.0
89.5 77.0 56.7
```

To import this data, use this command:

```
[grade1 grade2 grade3] = textread('grades.dat','%f %f %f',... 'headerlines',1)
```

```
grade1 =
78.8000
99.5000
89.5000
```

```
grade2 =
55.9000
66.8000
77.0000
grade3 =
45.9000
78.0000
56.7000
```

Importing Mixed Alphabetic and Numeric Data

If your data file contains a mix of alphabetic and numeric ASCII data, use the `textread` function to import the data. `textread` can return multiple output variables and you can specify the data type of each variable.

For example, the file `mydata.dat` contains a mix of alphabetic and numeric data.

```
Sally Type1 12.34 45 Yes
Larry Type2 34.56 54 Yes
Tommy Type1 67.89 23 No
```

Note To read an ASCII data file that contains primarily numeric data but with text column headers, see “Importing Numeric Data with Text Headers” on page 6-13.

To read the entire contents of the file `mydata.dat` into the workspace, specify the name of the data file and the format string as arguments to `textread`. In the format string, you include conversion specifiers that define how you want each data item to be interpreted. For example, specify `%s` for

strings data, %f for floating point data, and so on. (For a complete list of format specifiers, see the [textread](#) reference page.)

For each conversion specifier in your format string, you must specify a separate output variable. [textread](#) processes each data item in the file as specified in the format string and puts the value in the output variable. The number of output variables must match the number of conversion specifiers in the format string.

In this example, [textread](#) reads the file mydata.dat, applying the format string to each line in the file until the end of the file.

```
[names,types,x,y,answer] = textread('mydata.dat','%s %s %f ...
```

```
%d %s',1)
```

```
names =
```

```
'Sally' 'Larry' 'Tommy'
```

```
types =
```

```
'Type1' 'Type2' 'Type1'
```

```
x =
```

```
12.3400
```

```
34.5600
```

```
67.8900
```

```
y =
```

```
45
```

```
54
```

```
23
```

```
answer = 'Yes' 'Yes' 'No'
```

If your data uses a character other than a space as a delimiter, you must use the [textread](#) parameter 'delimiter' to specify the delimiter. For example, if the file mydata.dat used a semicolon as a delimiter, you would use this command.

```
[names,types,x,y,answer]=textread('mydata.dat','%s %s %f ... %d %s', 'delimiter',';')
```

See the [textread](#) reference page for more information about these optional parameters.

Exporting ASCII Data

MATLAB supports several ways to export data in many different ASCII formats. For example, you may want to export a MATLAB matrix as text file where the rows and columns are represented as space-separated, numeric values.

This section describes how to use MATLAB functions to export data in several common ASCII formats, including:

- “[Exporting Delimited ASCII Data Files](#)” on page 6-17

- “[Using the diary Command to Export Data](#)” on page 6-18 The function you use depends on the amount of data you want to export and its format.

If you are not sure which section describes your data, scan the data format samples in Table 6-3 and look for the sample that most nearly matches the data format you want to create. Then, read the section referred to in the table.

If you are familiar with MATLAB export function but not sure when to use them, view Table 6-4, which compares the features of each function.

Note If C or Fortran routines for writing data files in the form needed by other applications exist, create a MEX file to write the data. See the *External Interfaces Guide* for more information.

Table 6- 3: ASCII Data File Formats and MATLAB Export Commands
Data Format Sample MATLAB Export Function

1 2 3 4 5 See “Exporting Delimited ASCII Data Files” on page 6-17 and
6 7 8 9 10 “Using the diary Command to Export Data” on page 6-18 for information about these options.

1; 2; 3; 4; 5; See “Exporting Delimited ASCII Data Files” on page 6-17 for
6; 7; 8; 9; 10; more information. The example shows a semicolon-delimited file but you can specify other characters as the delimiter.

Table 6- 4: ASCII Data Export Function Feature Comparison

Function Use With Delimiters Notes

csvwrite Numeric data Only

commas

diary Numeric data or cell array

Only spaces

dlmwrite Numeric data

fprintf Alphabetic and

numeric data Any

character

Any

character

save Numeric data

Tabs or spaces Primarily used with spreadsheet data. See also the binary format spreadsheet export functions

Can be used for small arrays. Requires editing of data file to remove extraneous text.

Easy to use, flexible.

Part of low-level file I/O routines. Most flexible command but most difficult to use. Requires use of fopen to obtain file identifier and fclose after write.

Easy to use; output values are high precision.

Exporting Delimited ASCII Data Files

To export an array as a delimited ASCII data file, you can use either the save command, specifying the-ASCII qualifier, or the dlmwrite function. The save command is easy to use; however, the dlmwrite function provides more flexibility, allowing you to specify any character as a delimiter and to export subsets of an array by specifying a range of values.

Using the save Command

To export the array A,

```
A = [ 1 2 3 4 ; 5 6 7 8 ];  
use the save command, as follows.  
save my_data.out A -ASCII
```

If you view the created file in a text editor, it looks like this.

```
1.0000000e+000 2.0000000e+000 3.0000000e+000 4.0000000e+000 5.0000000e+000  
6.0000000e+000 7.0000000e+000 8.0000000e+000 By default, save uses spaces as delimiters but  
you can use tabs instead of spaces by specifying the -tabs qualifier.
```

When you use save to write a character array to an ASCII file, it writes the ASCII equivalent of the characters to the file. If you write the character string 'hello' to a file, save writes the values

```
104 101 108 108 111
```

Using the dlmwrite Function

To export an array in ASCII format and specify the delimiter used in the file, use the dlmwrite function.

For example, to export the array A,

```
A = [ 1 2 3 4 ; 5 6 7 8 ];  
as an ASCII data file that uses semicolons as a delimiter, use this command  
dlmwrite('my_data.out',A, ';')  
If you view the created file in a text editor, it looks like this  
1;2;3;4 5;6;7;8
```

Note that dlmwrite does not insert delimiters at the end of rows.

By default, if you do not specify a delimiter, dlmwrite uses commas as a delimiter. You can specify a space (' ') as a delimiter or, if you specify empty quotes (""), no delimiter.

Using the diary Command to Export Data

To export small numeric arrays or cell arrays, you can use the diary command. diary creates a verbatim copy of your MATLAB session in a disk file (excluding graphics).

For example, if you have the array A, in your workspace

```
A = [ 1 2 3 4 ; 5 6 7 8 ];  
execute these commands at the MATLAB prompt to export this array using diary:  
1 Turn on the diary function. You can optionally name the output file diary creates.
```

```
diary my_data.out
```

2 Display the contents of the array you want to export. This example displays the array A. You could also display a cell array or other MATLAB data type.

```
A =  
1 2 3 4  
5 6 7 8
```

3 Turn off the diary function. diary off

diary creates the file, my_data.out, and records all the commands executed in the MATLAB session until it is turned off.

```
A =  
1 2 3 4 5 6 7 8  
diary off
```

4 Open the diary file, my_data.out, in a text editor and remove all the extraneous text.

Importing Binary Data

The easiest way to import binary data is by using the Import Wizard. You simply start the Import Wizard and specify the file that contains the data you want to import. For more information, see “Using the Import Wizard with Binary Data Files” on page 6-20.

If you need to work from the MATLAB command line or perform import operations as part of an M-file, you must use one of the MATLAB import functions. MATLAB supports many functions to import data in different binary formats, such as image files or spreadsheet data files. Your choice of which function to use depends on the type of data in the file and how the data is formatted.

Caution When you import data into the MATLAB workspace, it overwrites any existing variables in the workspace with the same name.

Using the Import Wizard with Binary Data Files

To import text data using the Import Wizard, perform these steps:

1 Start the Import Wizard, by selecting the **Import Data** option on the MATLAB **File** menu. MATLAB displays a file selection dialog box. You can also use the uiimport function to start the Import Wizard.

To use the Import Wizard to import data from the clipboard, select the **Paste Special** option on the MATLAB **Edit** menu. You can also right-click in the MATLAB command window and choose **Paste Special** from the context menu. Skip to step 3 to continue importing from the clipboard.

2 Specify the file you want to import in the file selection dialog box and click **Open**. The Import Wizard opens the file and attempts to process its contents. See *Viewing the variables* for more information.

3 Select the variables that you want to import. By default, the Import Wizard creates variables depending on the type of data in the file.

4 Click **Finish** to import the data into the workspace.

Viewing the Variables

When the Import Wizard opens a binary data file, it attempts to process the data in the file, creating variables from the data it finds in the file. For example, if you use the Import Wizard to import this sample MAT-file, my_data.mat,

```
A =  
1 2 3 4 5  
6 7 8 9 10
```

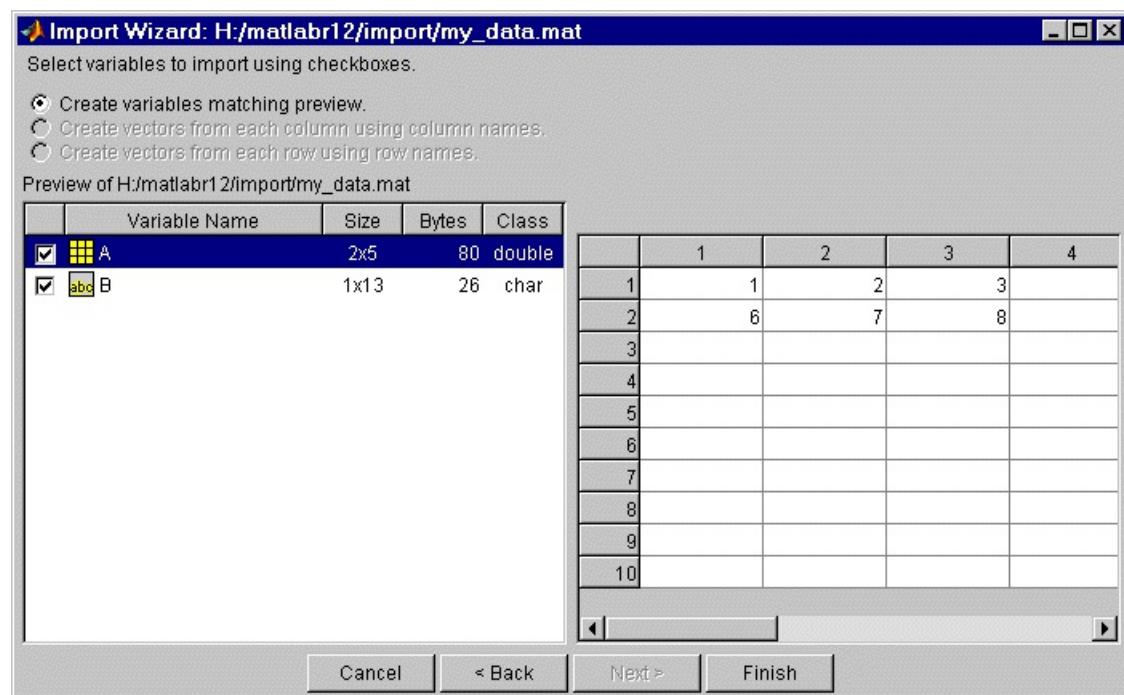
B =

Preview of the variables in the file.

Preview of the data in each variable.

a test string

it creates two variables, listed in the Preview pane. You can select the variables you want to import by clicking in the check box next to its name. All variables are preselected by default.

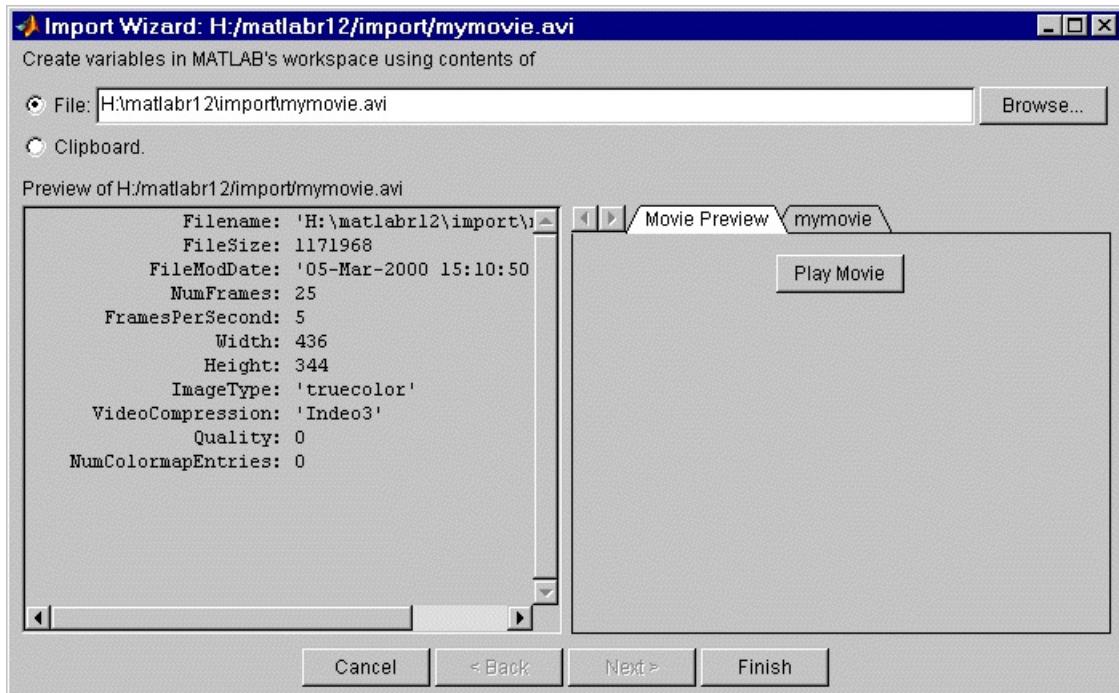


For other binary data types, such as images and sound files, the Import Wizard displays information about the data in the left pane and provides a Preview button in the right pane of the dialog box. Click the preview button to view (or listen to) the data.

Information about the data.

Preview button.

For example, when used to import a movie in Audio Video Interleaved (AVI) format, the Import Wizard displays this dialog box..



Using

Import Functions with Binary Data

To import binary data from the command line or in an M-file, you must use one of the MATLAB import functions. Your choice of function depends on how the data in the text file is formatted.

To find the function designed to work with a particular binary data format, scan the data formats listed in Table 6-5. The table lists the binary formats and the MATLAB high-level functions you use to import them, along with pointers to additional information.

Table 6- 5: Binary Data Formats and MATLAB Import Functions

Data Format File

Extension Description

Audio files .au

.wav

Audio-video

Interleaved (AVI) Hierarchical Data Format (HDF)

.hdf

Image files

.jpeg

.tiff .bmp

.gif

.png

.hdf

.pcx

.xwd

Use the auread function to import audio files on Sun Microsystems platforms and the wavread

function to import audio files on Microsoft Windows systems.

Use the aviread function to import Audio-video Interleaved (AVI) files.

For HDF image files, use imread. For all other HDF files, see “Working with HDF Data” on page 6-29 for complete information.

Use the imread function to import image data in many different formats. See Reading, Writing, and Querying Graphics Image Files for more information.

MATLAB
proprietary format (MAT-files)

Spreadsheets
.avi
.mat
.xls .wk1

Use the load command to import data in MATLAB proprietary format. See “Loading a Saved Workspace” on page 5-7 for more information.

Use xlsread to import Excel spreadsheet data or wk1read to import Lotus 123 data.

To view the alphabetical list of MATLAB binary data export functions, see Table 6-6.

If MATLAB does not support a high-level function that works with a data format, you can use the MATLAB low-level file I/O functions, if you know how the binary data is formatted in the file. See “Using Low-Level File I/O Functions” on page 6-51 for more information.

Table 6- 6: Binary Data Import Functions

Function **File Extension** **Data Format**
auread .au Import sound data in Sun Microsystems format.
aviread .avi Import audio-visual data in AVI format.
hdf .hdf
imread .jpeg,.tiff,.bmp, .png,.hdf,.pcx, .xwd, .gif

Import data in Hierarchical Data Format (HDF). For HDF image file formats, use imread. For all other HDF files, see “Working with HDF Data” on page 6-29 for complete information.

Import images in many formats.

load .mat
wavread .wav
wk1read .wk1
xlsread .xls

Import MATLAB workspace variables in MAT-files format.

Import sound data in Microsoft Windows format. Import data in Lotus 123 spreadsheet format.
Import data in Microsoft Excel spreadsheet format.

Exporting Binary Data

To export binary data in one of the standard binary formats, you can use the MATLAB high-

level function designed to work with that format.

To find the function designed to work with a particular binary data format, scan the data formats listed in Table 6-7. The table lists the binary formats and the MATLAB high-level functions you use to import them, with pointers to sources of additional information.

Table 6- 7: Binary Data Formats and MATLAB Export Functions

Data Format File Extension Description

Audio files .au .wav

Audio Video .avi Interleaved (AVI)

Hierarchical Data .hdf Format (HDF)

Image files .jpeg .tiff .bmp .png .hdf .pcx .xwd

Use the auwrite function to export audio files on Sun Microsystems platforms and the wavwrite function to export audio files in Microsoft Windows format.

You must use the avifile, addframe, and the close function overloaded for AVI data to export a sequence of MATLAB figures in AVI format. See “Exporting MATLAB Graphs in AVI Format” on page 6-27 for more information.

To export image data in HDF format, use imwrite. For all other HDF formats, see “Working with HDF Data” on page 6-29 for complete information.

Use the imwrite function to export image data in many different formats. See Reading, Writing, and Querying Graphics Image Files for more information.

Data Format File

Extension Description

MATLAB

proprietary format (MAT-files)

Spreadsheets

.mat

.xls .wk1

Use the save command to export data in MATLAB proprietary format. See “Saving the Current Workspace” on page 5-5 for more information.

Use the wk1write function to export data in Lotus 123 format.

To view the alphabetical list of MATLAB binary data export functions, see Table 6-6.

If MATLAB does not support a high-level function that works with a data format, you can use the MATLAB low-level file I/O functions, if you know how the binary data is formatted in the file. See “Using Low-Level File I/O Functions” on page 6-51 for more information.

Table 6- 8: Binary Data Export Functions

Function File Extension Data Format

addframe .au
auwrite .au
avifile .au
hdf .hdf
imwrite

save .jpeg,.tiff, .bmp,.png,
.hdf,.pcx,.xwd

.mat

Capture snapshots of the current axes and put them in an AVI file object. See also avifile.
Export sound data in Sun Microsystems format. Export audio-visual data in AVI format. Creates an AVI file object. See also addframe.

Export data in Hierarchical Data Format (HDF). For HDF image file formats, useimwrite. For all other HDF files, see “Working with HDF Data” on page 6-29 for complete information.

Export image files in many formats.

Export MATLAB variables in MAT-files format.

Function File Extension Data Format

wavwrite .wav Export sound data on Microsoft Windows platforms. wk1write .wk1 Export data in Lotus 123 spreadsheet format.

Ex porting M ATLAB Graphs in AVI Form at

In MATLAB, you can save a sequence of graphs as a *movie* that can then be played back using themovie function. You can export a MATLAB movie by saving it in MAT-file format, like any other MATLAB workspace variable. However, anyone who wants to view your movie must have MATLAB. (For more information about MATLAB movies, see the “Animation” section in *Using MATLAB Graphics*.)

To export a sequence of MATLAB graphs in a format that does not require MATLAB for viewing, save the figures in Audio Video Interleaved (AVI) format. AVI is a file format that allows animation and video clips to be played on a PC running Windows or on UNIX systems.

Creating an AVI Format Movie

To export a sequence of MATLAB graphs as an AVI format movie, perform these steps:

- 1 Create an AVI file, using theavifile function.
- 2 Capture the sequence of graphs and put them into the AVI file, using the addframe function.
- 3 Close the AVI file, using theclose function, overloaded for AVI files.

Note To convert an existing MATLAB movie into an AVI file, use the movie2avi function.

For example, this code example exports a sequence of MATLAB graphs as the AVI filuemovie.avi. The numbers in comments correspond to notes following the code example.

```
aviobj = avifile('mymovie.avi','fps',5);
```

```

for k=1:25
    h = plot(fft(eye(k+16))); set(h,'EraseMode','xor'); axis equal;
    frame = getframe(gca);
    aviobj = addframe(aviobj,frame);
end

```

aviobj = close(aviobj); Note the following items in this code example:

- The `aveavifile` function creates an AVI file and returns a handle to an AVI file object. AVI file objects support properties that let you control various characteristics of the AVI movie, such as colormap, compression, and quality. (See the `aveavifile` reference page for a complete list.) `aveavifile` uses default values for all properties, unless you specify a value. In the example, the call to `aveavifile` explicitly sets the value of one frames per second (`fps`) property.
- The example uses a `for` loop to capture the series of graphs to be included in the movie. You typically use `addframe` to capture a sequence of graphs for AVI movies. However, because this particular MATLAB animation uses XOR graphics, you must call `getframe` to capture the graphs and then call `addframe` to add the captured frame to the movie. See the `addframe` reference page for more information.
- The example calls the `close` function to finish writing the frames to the file and to close the file.

Working with HDF Data

Hierarchical Data Format (HDF) is a general-purpose, machine-independent standard for storing scientific data in files. The National Center for Supercomputing Applications (NCSA), the original developer of HDF, distributes libraries of C and Fortran routines that scientists use to read and write data in HDF format.

This section describes how to call these NCSA routines from within MATLAB. Topics include:

- An overview of MATLAB HDF support
- MATLAB HDF function calling conventions
- Importing HDF data into the MATLAB workspace
- Exporting MATLAB data in an HDF file
- Including metadata in an HDF file
- Using MATLAB specific HDF functions

Additional Information Sources

This section does not attempt to describe all HDF features and routines. To use the MATLAB HDF interface effectively, you must use this documentation in conjunction with the official HDF documentation, available in many formats at the NCSA Web site (<http://hdf.ncsa.uiuc.edu/>). In particular, consult the following documentation:

- The *HDFUser's Guide*, which describes key HDF concepts and programming models and provides tutorial information about using the library routines
- The *HDF Reference Manual*, which provides detailed reference information about the hundreds of HDF routines, their arguments, and return values

The National Aeronautics and Space Administration (NASA) has created an extension of HDF as one of the data standards for the Earth Observing System (EOS). For information about this extension to HDF, consult the official HDF-EOS documentation at the EOS Web site (<http://hdfeos.gsfc.nasa.gov/hdfeos/workshop.html>).

Finally, for details about the syntax of the MATLAB HDF functions, consult the MATLAB online Function Reference. For most HDF routines, the MATLAB syntax is essentially the same as the HDF version; however, for certain routines, the MATLAB version has a different syntax.

Overview of MATLAB HDF Support

The NCSA organizes the routines in the HDF library into collections, called *External Interfaces (APIs)*. Each API works with particular type of data. For example, the HDF Scientific Data (SD) API works with multidimensional arrays of numeric data. Table , , lists all the HDF APIs supported by MATLAB (listed alphabetically by acronym). The table includes a MATLAB specific API made up of utility functions.

In addition to these standard HDF APIs, MATLAB also supports the EOS extension to HDF. HDF-EOS provides functions to store, manage, and retrieve multidimensional arrays of numeric data, point data, and swath data. See the HDF-EOS documentation for more information.

Note MATLAB supports the version 4.1r3 of the NCSA multifile APIs to HDF data. The multifile APIs replace the original NCSA APIs. MATLAB does not support HDF version 5.0, which is a completely new format and is not compatible with version 4.1r3.

Table 6 - 9: Supported HDF Application Programming Interfaces (APIs)

Application Acronym Programming Interface	Description
--	--------------------

Annotations AN

General Raster DF24

Images DFR8

Stores, manages, and retrieves text used to describe an HDF file or any of the data structures contained in the file.

Stores, manages, and retrieves raster images, their dimensions and palettes. It can also manipulate unattached palettes.

Note: Use the MATLAB high-level functions imread and imwrite with HDF raster image formats.

Table 6 - 9: Supported HDF Application Programming Interfaces (APIs)

Application Programming Interface
--

Acronym Description

HDF Utilities H, HD, and HE

MATLAB HDF Utilities
Scientific Data SD
V Groups V
V Data
ML

VS VF VH

Provides functions to open and close HDF files and handle errors.
Provides utility functions that help you work with HDF files in the MATLAB environment.

Stores, manages, and retrieves multidimensional arrays of character or numeric data, along with their dimensions and attributes.

Creates and retrieves groups of other HDF data objects, such as raster images or V data.
Stores, manages, and retrieves multivariate data stored as records in a table.

M ATLAB HDF Function Calling Conventions

Each HDF API includes many individual routines to read data from files, write data to files, and perform other related functions. For example, the HDF SD API includes separate C routines to open (SDopen), close (SDend), and read data (SDreaddata).

MATLAB, instead of supporting a corresponding function for each individual HDF API routine, supports a single function for each HDF API. You use this single function to access all the individual routines in the HDF API, specifying the name of the individual HDF routine as the first argument.

For example, to call the HDF SD API routine to terminate access to an HDF file in a C program, you use

```
status = SDend(sd_id);
```

To call this routine from MATLAB, use the MATLAB function associated with the API. By convention, the name of the MATLAB function associated with an HDF API includes the API acronym in the function name. For example, the MATLAB function used to access routines in the HDF SD API is calledhdfsd. As the first argument to this function, specify the name of the API routine, minus the acronym, and pass the remaining arguments expected by the routine in the order they are required. Thus, to call theSDend routine from MATLAB, use this syntax.

```
status = hdfsd('end',sd_id);
```

Note For some HDF API routines, particularly those that use output arguments to return data, the MATLAB calling sequence is different. (See “Handling HDF Routines with Output Arguments” on page 6-32 for more information.) Refer to the MATLAB online Function Reference to make sure you have the correct syntax.

Handling HDF Routines with Output Arguments

When calling HDF API routines that use output arguments to return data, you must specify all

output arguments as return values. For example, in C syntax, the SDfileinfo routine returns data about an HDF file in two output arguments,ndatasets and nglobal_atts.

status = SDfileinfo(sd_id, ndatasets, nglobal_atts); To call this routine from MATLAB, change the output arguments into return values.

[ndatasets, nglobal_atts, status] = hdfsd('fileinfo',sd_id); Specify the return values in the same order as they appear as output arguments. The function status return value is always specified last.

Handling HDF Library Symbolic Constants

The C versions of the HDF APIs use symbolic constants, defined in header files, to specify modes and data types. For example, the SDstart routine uses a symbolic constant to specify the mode in which to open an HDF file.

```
sd_id = SDstart("my_file.hdf",DFACC_RDONLY);
```

When calling this routine from MATLAB, specify these constants as text strings.

```
sd_id = hdfsd('start','my_file.hdf','DFACC_RDONLY')
```

In MATLAB, you can specify the entire constant or leave off the prefix. For example, in this call toSDstart, you can use any of these variations as the constant text string:'DFACC_RDONLY','dfacc_rdonly', or'rdonly'. Note that you can use any combination of upper- and lower-case characters.

Importing HDF Data into the MATLAB Workspace

To import HDF data into MATLAB, you must use the routines in the HDF API associated with the particular HDF data type. Each API has a particular programming model, that is, a prescribed way to use the routines to open the HDF file, access data sets in the file, and read data from the data sets. (In HDF terminology, the numeric arrays stored in HDF files are called data sets.)

To illustrate this concept, this section details the programming model of one particular HDF API: the Scientific Data (SD) API. For information about working with other HDF APIs, see the official NCSA documentation.

Note The following sections, when referring to specific routines in the HDF SD API, use the C library name rather than the MATLAB function name. The MATLAB syntax is used in all examples.

The HDF SD Import Programming Model

To import data in HDF SD format, you must use API routines to perform these steps:

- 1** Open the file containing HDF SD data sets.
- 2** Select the data set in the file that you want to import.
- 3** Read the data from the data set.
- 4** Close access to the data set and HDF file.

There are several additional steps that you may also need to perform, such as retrieving information about the contents of the HDF file or the data sets in the file. The following sections

provide more detail about the basic steps as well as optional steps.

Opening HDF Files

To import an HDF SD data set, you must first open the file containing the data set. In the HDF SD API, you use the SDstart routine to open the file and initialize the HDF interface. In MATLAB, you use the hdfsd function with start specified as the first argument.

SDstart accepts these arguments:

- A text string specifying the name of the file you want to open
- A text string specifying the mode in which you want to open it

For example, this code opens the file mydata.hdf for read access. `sd_id = hdfsd('start','mydata.hdf','read');`

If SDstart can find and open the file specified, it returns an HDF SD file identifier, named `sd_id` in the example. Otherwise, it returns -1.

The HDF SD API supports several file access modes. You use a symbolic constant, defined by the HDF SD API, to specify each mode. In MATLAB, you specify these constants as text strings. You can specify the full HDF constant or one of the abbreviated forms listed in table.

HDF File Creation Mode Create a new file

Read access

Read and write access

HDF Symbolic Constant MATLAB String 'DFACC_CREATE' 'create'

'DFACC_RDONLY' 'read' or 'rdonly' 'DFACC_RDWR' 'rdwr' or 'write'

Retrieving Information About an HDF File

After opening a file, you can get information about what the file contains using the SDfileinfo routine. In MATLAB, you use the `thehdfsd` function with `fileinfo` specified as the first argument. This function returns the number of data sets in the file and whether the file includes any global attributes. (For

more information about global attributes, see “Retrieving Attributes from an

HDF File” on page 6-35.)

As an argument, `SDfileinfo` accepts the SD file identifier, `sd_id`, returned by `SDstart`. In this example, the HDF file contains three data sets and one global attribute.

```
[ndatasets, nglobal_atts, stat] = hdfsd('fileinfo',sd_id)  
ndatasets = 3  
nglobal_atts = 1  
status = 0
```

Retrieving Attributes from an HDF File

HDF files are self-documenting, that is, they can optionally include information, called *attributes*, that describes the data the file contains. Attributes associated with an HDF file are called *global* attributes. (You can also associate attributes with data sets or dimensions. For more information about these attributes, see “Including Metadata in an HDF File” on page 6-47.)

In the HDF SD API, you use the SDreadattr routine to retrieve global attributes from an HDF file. In MATLAB, you use the hdfsd function, specifying readattr as the first argument. As other arguments, you specify:

- The HDF SD file identifier (sd_id) returned by the SDstart routine
- The index value specifying the attribute you want to view. HDF uses zero-based indexing so the first global attribute has an index value zero, the second has an index value one, and so on.

For example, this code returns the contents of the first global attribute, which is simply the character string ‘my global attribute’.

```
attr_idx = 0;  
[attr, status] = hdfsd('readattr', sd_id, attr_idx)  
attr =  
my global attribute  
stat = 0
```

MATLAB automatically sizes the return value, attr, to fit the data in the attribute.

Retrieving Attributes by Name. Attributes have names as well as values. If you know the name of an attribute, you can use the SDfindattr function to determine its index value so you can retrieve it. In MATLAB, you use the hdfsd function, specifying findattr as the first argument.

As other arguments, you specify:

- The HDF SD file identifier, when searching for global attributes
- A text string specifying the name of the attribute

SDfindattr searches all the global attributes associated with the file. If it finds an attribute with this name, SDfindattr returns the index of the attribute. You can then use this index value to retrieve the attribute using SDreadattr.

This example uses SDfindattr to obtain the index for the attribute named my_att and then passes this index as an argument to SDreadattr.

```
attr_idx = hdfsd('findattr', sd_id, 'my_att');  
[attr, status] = hdfsd('readattr', sd_id, attr_idx);
```

Selecting Data Sets in HDF Files

After opening an HDF file, you must specify the data set in the file that you want to read. An HDF file can contain multiple data sets. In the HDF SD API, you use the SDselect routine to select a data set. In MATLAB, you use the hdfsd function, specifying select as the first argument. As arguments, this function accepts:

- The HDF SD file identifier (sd_id) returned by SDstart
- The index value specifying the attribute you want to view. HDF uses zero-based indexing so the first global attribute has an index value zero, the second has an index value one, and so on.

For example, this code selects the third data set in the HDF file identified by `sd_id`. If `SDselect` finds the specified data set in the file, it returns an HDF SD data set identifier, called `sds_id` in the example. If it cannot find the data set, it returns -1.

Note Do not confuse HDF SD *file* identifiers, named `sds_id` in the examples, with HDF SD *data set* identifiers, named `sds_id` in the examples.

```
sds_idx = 2; % HDF uses zero-based indexing. sds_id = hdfsd('select',sd_id,sds_idx)
```

Retrieving Data Sets by Name

Data sets in HDF files can be named. If you know the name of the data set you are interested in, but not its index value, you can determine its index by using the `SDnametoindex` routine. In MATLAB, use the `hdfsd` function, specifying `nametoindex` as the first argument.

Getting Information About a Data Set

After you select a data set in an HDF file, you can obtain information about the data set, such as the number and size of the array dimensions. You need this information to read the data set using the `SDreadadd` function. (See “Reading Data from an HDF File” on page 6-39 for more information.)

In the HDF SD API, you use the `SDgetinfo` routine to gather this information. In MATLAB, use the `hdfsd` function, specifying `getinfo` as the first argument. In addition, you must specify the HDF SD data set identifier returned by `SDselect` (`sds_id`).

This table lists the information returned by `SDgetinfo`.

Data Set Information Returned	MATLAB Data Type	Name	Character array	Number of dimensions
Scalar	Scalar	Size of each dimension		
Data type of the data stored in the array				
Number of attributes associated with the data set				
Vector	Character array			

Scalar For example, this code retrieves information about the data set identified by `sds_id`.

```
[dsname, dsndims, dsdims, dstype, dsatts, stat] = hdfsd('getinfo',sds_id)
```

```
dsname =
```

```
A
```

```
dsndims = 2
dsdims =
5 3
dstype =
double
dsatts = 0
stat = 0
```

Retrieving Data Set Attributes

Like HDF files, HDF SD data sets are self-documenting, that is, they can optionally include information, called *attributes*, that describes the data in the data set. Attributes associated with a

data set are called *local* attributes. (You can also associate attributes with files or dimensions. For more information about these attributes, see “Including Metadata in an HDF File” on page 6-47.)

In the HDF SD API, you use the SDreadattr routine to retrieve local attributes. In MATLAB, use thehdfs function, specifyingreadattr as the first argument. As other arguments, specify

- The HDF SD data set identifier (sds_id) returned bySDselect
- The index of the attribute you want to view. HDF uses zero-based indexing so the first attribute has the index value zero, the second has an index value one, and so on.

This code example returns the contents of the first attribute associated with a data set identified bysds_id. In this case, the value of the attribute is the character string'my local attribute'. MATLAB automatically sizes the return value,ds_attr, to fit the value of the attribute.

```
attr_idx = 0;
[ds_attr, status] = hdfs('readattr', sds_id, attr_idx)
ds_attr =
my local attribute
stat = 0
```

Reading Data from an HDF File

After you open an HDF file and select a data set in the file, you can read the entire data set, or part of the data set. In the HDF SD API, you use the SDreaddata routine to read a data set. In MATLAB, use thehdfs function, specifyingreaddata as the first argument. As other arguments, specify:

- The HDF SD data set identifier (sds_id) returned bySDselect
- The location in the data set where you want to start reading data, specified as a vector of index values, called the *start* vector in HDF terminology
- The number of elements along each dimension to skip between each read operation, specified as a vector scalar values, called the *stride* vector in HDF terminology
- The total number of elements to read along each dimension, specified as a vector of scalar values, called the *edges* vector in HDF terminology

For example, to read the entire contents of a data set containing this 3-by-5 matrix of numeric values

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
```

you could use this code.

```
[ds_name, ds_ndims, ds_dims, ds_type, ds_atts, stat] = hdfs('getinfo',sds_id);
ds_start = zeros(1,ds_ndims); % Creates the vector [0 0] ds_stride = [];
ds_edges = ds_dims; [ds_data, status] =
hdfs('readdata',sds_id,ds_start,ds_stride,ds_edges);

disp(ds_data)
```

```
1 2 3 4 5  
6 7 8 9 10 11 12 13 14 15
```

In this example, note the following:

- The return values of SDgetinfo are used to specify the dimensions of the return values and as arguments to SDreaddata.
- To read from the beginning of a data set, specify zero for each element of the start vector (ds_start). Note how the example uses SDgetinfo to determine the length of the start vector.
- To read every element of a data set, specify one for each element of the stride vector or specify an empty array ([]).
- To read every element of a data set, set each element of the edges vector to the size of each dimension of the data set.

Note Use the dimensions vector returned by SDgetinfo, dsdims, to set the value of the edges vector because SDgetinfo returns these values in row-major order, the ordering used by HDF. MATLAB stores data in column-major order. An array referred to as a 3-by-5 array in MATLAB is described as a 5-by-3 array in HDF.

Reading a Portion of a Data Set

To read less than the entire data set, use the start, stride, and edges vectors to specify where you want to start reading data and how much data you want to read.

For example, this code fragment uses SDreaddata to read the entire second row of the sample data set.

```
1 2 3 4 5  
6 7 8 9 10  
11 12 13 14 15
```

Note that in the start, stride, and edges arguments, you must specify the dimensions in column-major order, that is, [columns,rows]. In addition, note that you must use zero-based indexing in these arguments.

```
ds_start = [0 1] % Start reading at the first column, second row  
ds_stride = []; % Read each element  
ds_edges = [5 1]; % Read a 1-by-5 vector of data
```

```
[ds_data, status] =  
hdfsd('readdata', sds_id, ds_start, ds_stride, ds_edges);  
disp(ds_data)  
6 7 8 9 10
```

For more information about specifying ranges in data sets, see “Writing MATLAB Data to an HDF File” on page 6-44.

Closing HDF Files and HDF Data sets

After reading data from a data set in an HDF file, you must close access to the data set and the

file. The HDF SD API includes functions to perform these tasks. See “Closing HDF Data Sets” on page 6-46 for more information.

Exporting M ATLAB Data in an HDF File

To export data from MATLAB in an HDF file, you must use the functions in the HDF API associated with the HDF data type. Each API has a particular programming model, that is, a prescribed way to use the routines to open the HDF file, access data sets in the file, and write data to the data sets. (In HDF terminology, the numeric arrays stored in HDF files are called data sets.)

To illustrate this concept, this section details the programming model of one particular HDF API: the Scientific Data (SD) API. For information about working with other HDF APIs, see the official NCSA documentation.

The HDF SD Export Programming Model

The programming model for exporting HDF SD data involves these steps:

- 1 Create the HDF file, or open an existing one.
- 2 Create a data set in the file, or select an existing one.
- 3 Write data to the data set.
- 4 Close access to the data set and the HDF file.

You can optionally include information in the HDF file that describes your data. See “Including Metadata in an HDF File” on page 6-47 for more information.

Creating an HDF File

To export MATLAB data in HDF format, you must first create an HDF file, or open an existing one. In the HDF SD API, you use the SDstart routine. In MATLAB, use the hdfsd function, specifying start as the first argument. As other arguments, specify:

- A text string specifying the name you want to assign to the HDF file (or the name of an existing HDF file)
- A text string specifying the HDF SD interface file access mode

For example, this code creates an HDF file named mydata.hdf. `sd_id = hdfsd('start','mydata.hdf','DFACC_CREATE');`

If it can create (or open) the file, SDstart returns an HDF SD file identifier, named `sd_id` in the example. Otherwise, it returns -1.

When you specify the DFACC_CREATE access mode, SDstart creates the file and initializes the HDF SD multifile interface. If you specify DFACC_CREATE mode and the file already exists, SDstart fails, returning -1. To open an existing HDF file, you must use HDF read or write modes. For information about using SDstart in these modes, see “Opening HDF Files” on page 6-34.

Creating an HDF Data Set

After creating the HDF file, or opening an existing one, you must create a data set in the file for

each MATLAB array you want to export. In the HDF SD API, you use the SDcreate routine to create data sets. In MATLAB, you use the hdfsd function, specifying create as the first argument. To write data to an existing data set, you must obtain the HDF SD data set identifier. See “Selecting Data Sets in HDF Files” on page 6-36 for more information.

This table lists the other arguments to SDcreate.

Argument	MATLAB Data Type	Valid HDF SD file identifier	Returned from SDstart	Name you want assigned to the data set
		Text string		Text string
		Data type of the data set		Data type of the data set

Number of dimensions in the data set. This is called the *rank* of the data set in HDF terminology

Size of each dimension	Text string. For information about specifying data types, see “Importing HDF Data into the MATLAB Workspace” on page 6-33
------------------------	---

Scalar numeric value

Vector

The values you assign to these arguments depend on the MATLAB array you want to export. For example, to export this MATLAB 3-by-5 array of doubles,

A = [1 2 3 4 5 ; 6 7 8 9 10 ; 11 12 13 14 15];
you could set the values of these arguments as in this code fragment.

```
ds_name = 'A';  
ds_type = 'double';  
ds_rank = ndims(A);  
ds_dims = fliplr(size(A));  
  
sds_id = hdfsd('create',sd_id,ds_name,ds_type,ds_rank,ds_dims);
```

If SDcreate can successfully create the data set, it returns an HDF SD data set identifier (sds_id). Otherwise, SDcreate returns -1.

Note In this example, note how the code fragment reverses the order of the values in the dimensions argument (ds_dims). This processing is necessary because the MATLAB size function returns the dimensions in column-major order and HDF expects to receive dimensions in row-major order.

Once you create a data set, you cannot change its characteristics. You can, however, modify the data it contains. To do this, initiate access to the data set, using SDselect, and write to the data set as described in “Writing MATLAB Data to an HDF File” on page 6-44.

Writing MATLAB Data to an HDF File

After creating an HDF file and creating a data set in the file, you can write data to the entire data set or just a portion of the data set. In the HDF SD API, you use the SDwritedata routine. In

MATLAB, use the `hdfsd` function, specifying `writedata` as the first argument. This table lists the other arguments to `SDwritedata`.

Argument MATLAB Data Type Valid data set identifier (`sds_id`) Returned by `SDcreate`

Location in the data set where you want to start writing data, called the *start* vector in HDF terminology

Number of elements along each dimension to skip between each write operation, called the *stride* vector in HDF terminology

Total number of elements to write along each dimension, called the *edges* vector in HDF terminology

MATLAB array to be written

Vector of index values

Vector of scalar values

Vector of scalar values

Array of doubles **Note** You must specify the values of the `start`, `stride`, and `edges` arguments in row-major order, rather than the column-major order used in MATLAB. Note how the example uses `fliplr` to reverse the order of the dimensions in the vector returned by `thesize` function before assigning it as the value of the `edges` argument.

The values you assign to these arguments depend on the MATLAB array you want to export. For example, this code fragment writes this MATLAB 3-by-5 array of doubles,

```
A = [ 1 2 3 4 5; 6 7 8 9 10; 11 12 13 14 15 ];  
into an HDF file.
```

```
ds_start = zeros(1:ndims(A)); % Start at the beginning  
ds_stride = []; % Write every element.  
ds_edges = fliplr(size(A)); % Reverse the dimensions.
```

```
stat = hdfsd('writedata',sds_id,  
ds_start, ds_stride, ds_edges, A)
```

If it can write the data to the data set, `SDwritedata` returns 0; otherwise, it returns -1.

Note `SDwritedata` queues write operations. To ensure that these queued write operations are executed, you must close the file, using the `SDend` routine. See “Closing an HDF File” on page 6-46 for more information. As a convenience, MATLAB provides a function, `MLcloseall`, that you can use to close all open data sets and file identifiers with a single call. See “Using the MATLAB HDF Utility API” on page 6-49 for more information.

Writing Data to Portions of Data Sets

To write less than the entire data set, use the `start`, `stride`, and `edges` vectors to specify where you want to start writing data and how much data you want to write.

For example, this code fragment uses `SDwritedata` to replace the values of the entire second row of the sample data set

```
1 2 3 4 5  
6 7 8 9 10  
11 12 13 14 15
```

with the vector, B.

```
B = [ 9 9 9 9 9]
```

In the example, the start vector specifies that you want to start the write operation in the first column of the second row. Note how HDF uses zero-based indexing and specifies the column dimension first. In MATLAB, you would specify this location as(2,1). The edges argument specifies the dimensions of the data to be written. Note that the size of the array of data to be written must match the edge specification.

```
ds_start = [0 1] % Start writing at the first column, second row. ds_stride = []; % Write every element.  
ds_edges = [5 1]; % Each row is a 1-by-5 vector.  
  
stat = hdfsd('writedata',sds_id,ds_start,ds_stride,ds_edges,B);
```

Closing HDF Data Sets

After writing data to a data set in an HDF file, you must close access to the data set. In the HDF SD API, you use the SDendaccess routine to close a data set. In MATLAB, use thehdfsd function, specifyingendaccess as the first argument. As the only other argument, specify a valid HDF SD data set identifier,sds_id in this example.

```
stat = hdfsd('endaccess',sds_id);
```

Closing an HDF File

After writing data to a data set and closing the data set, you must also close the HDF file. In the HDF SD API, you use theSDend routine. In MATLAB, use the hdfsd function, specifyingend as the first argument. As the only other argument, specify a valid HDF SD file identifier, sd_id in this example.

```
stat = hdfsd('end',sd_id);
```

You must close access to all the data sets in an HDF file before closing it.

Note Closing an HDF file executes all the write operations that have been queued usingSDwritedata. As a convenience, the MATLAB HDF Utility API provides a function,MLcloseall, that can close all open data set and file identifiers with a single call. See “Using the MATLAB HDF Utility API” on page 6-49 for more information.

Including Metadata in an HDF File

You can optionally include information in an HDF file that describes your data. HDF defines an separate annotation API, however, the HDF SD API includes an annotation capability. This section only describes the annotation capabilities of the HDF SD API. For information about the Annotation API, see the official NCSA documentation.

Types of Attributes

Using HDF SD API, you can associate attributes with three types of HDF objects:

- An entire HDF file – File attributes, also called *global* attributes, generally contain information pertinent to all the data sets in the file.
- A data set in an HDF file – Data set attributes, also called *local* attributes, describe individual data sets.
- A dimension of a data set – Dimension attributes provide information applicable to an individual data set dimension.

Multiple Attributes

You can associate multiple attributes with a single HDF object. HDF maintains an attribute index for each object. The attribute index is zero-based. The first value has the index value zero, the second has the value one, and so on. You access an attribute by its index value.

Each attribute has the format name=value, where name (called label in HDF terminology) is a text string up to 256 characters in length and value contains one or more entries of the same data type. A single attribute can have multiple values.

Associating Attributes with HDF SD Objects

In the HDF SD API, you use the SDsetattr routine to associate an attribute with a file, data set or dimension. In MATLAB, use the hdfsd function, specifying setattr as the first argument. As other arguments, specify

- A valid HDF SD identifier associated with the object. This value could be a file identifier (sd_id), a data set identifier (sds_id), or a dimension identifier (dim_id).
- A text string that defines the name of the attribute. The SD interface supports predefined attributes that have reserved names and, in some cases, data types. For information about these attributes, see “Creating Predefined Attributes” on page 6-48.
- The attribute value

For example, this code creates a global attribute, named my_global_attr, and associates it with the HDF file identified by sd_id.

```
status = hdfsd('setattr',sd_id,'my_global_attr','my_attr_val');
```

Note In the NCSA documentation, the SDsetattr routine has two additional arguments: data type and the number of values in the attribute. When calling this routine from MATLAB, you do not have to include these arguments. The MATLAB HDF function can determine the data type and size of the attribute from the value you specify.

Creating Predefined Attributes

Predefined attributes are identical to user-defined attributes except that the the HDF SD API has already defined their names and data types. For example, the HDF SD API defines an attribute, named cordsys, in which you can specify the coordinate system used by the data set. Possible values of this attribute include the text strings 'cartesian', 'polar', and 'spherical'. Predefined attributes can be useful because they establish conventions that applications can depend on. The HDF SD API supports predefined attributes for data sets and dimensions only; there are no predefined attributes for files. For a complete list of the predefined attributes, see the NCSA

documentation. In the HDF SD API, you create predefined attributes the same way you create user-defined attributes, using the SDsetattr routine. In MATLAB, use the hdfsd function, specifying setattr as the first argument.

```
attr_name = 'cordsys';
attr_value = 'polar';
status = hdfsd('setattr',sds_id,attr_name,attr_value);
```

The HDF SD API also includes specialized functions for writing and reading the predefined attributes. These specialized functions, such as SDsetdatastrs, are sometimes easier to use, especially when reading or writing multiple related predefined attributes. You must use specialized functions to read or write the predefined dimension attributes.

Using the MATLAB HDF Utility API

In addition to the standard HDF APIs, listed in Table , , on page 6-30, MATLAB supports an API of utility functions that are designed to make using HDF in the MATLAB environment easier.

For example, the MATLAB utility API includes a function, MLlistinfo, which you can use to view all types of open HDF identifiers, such as HDF SD file identifiers. MATLAB updates these lists whenever HDF identifiers are created or closed.

This code obtains a list of all open HDF file and data set identifiers, using the MLlistinfo function. In this example, only two identifiers are open.

```
hdfml('listinfo')
No open RI identifiers
No open GR identifiers
No open grid identifiers
No open grid file identifiers
No open annotation identifiers
No open AN identifiers
Open scientific dataset identifiers:
262144
Open scientific data file identifiers:
393216
No open Vdata identifiers
No open Vgroup identifiers
No open Vfile identifiers No open point identifiers No open point file identifiers No open swath
identifiers No open swath file identifiers No open access identifiers No open file identifiers
```

Closing All Open HDF Identifiers

To close all open HDF identifiers in a single call, use the MLcloseall function. This call closes all open HDF identifiers.

```
hdfml('closeall')
```

Using Low-Level File I/O Functions

MATLAB includes a set of low-level file I/O functions that are based on the I/O functions of the ANSI Standard C Library. If you know C, therefore, you are probably familiar with these routines.

For example, the MATLAB file I/O functions use the same programming model as the C language routines. To read or write data, you perform these steps:

1 Open the file, using `fopen`. `fopen` returns a file identifier which you use with all the other low-level file I/O routines.

2 Operate on the file.

a Read binary data using `fread`.

b Write binary data using `fwrite`.

c Read text strings from a file line-by-line using `fgets`/`fgetl`. **d** Read formatted ASCII data using `fscanf`.

e Write formatted ASCII data using `fprintf`.

3 Close the file, using `fclose`.

This section also describes how these functions affect the current position in the file where read or write operations happen and how you can change the position in the file.

Note While the MATLAB file I/O commands are modeled on the C language I/O routines, in some ways their behavior is different. For example, the `fread` function is “vectorized,” that is, it continues reading until it encounters a text string or the end of file. These sections, and the MATLAB reference pages for these functions, highlight any differences in behavior.

Opening Files

Before reading or writing a text or binary file, you must open it with the `fopen` command.
`fid = fopen('filename','permission')`

Specifying the Permissions String

The `permission` string specifies the kind of access to the file you require. Possible permission strings include:

- `r` for reading only
- `w` for writing only
- `a` for appending only
- `r+` for both reading and writing

Note Systems such as Microsoft Windows that distinguish between text and binary files may require additional characters in the permission string, such as '`rb`' to open a binary file for reading.

Using the Returned File Identifier (`fid`)

If successful, `fopen` returns a nonnegative integer, called a *file identifier* (`fid`). You pass this value as an argument to the other I/O functions to access the open file. For example, this `fopen` statement opens the data file named `penny.dat` for reading.

```
fid = fopen('penny.dat','r')
Iffopen fails, for example if you try to open a file that does not exist,fopen:
```

- Assigns -1 to the file identifier.
- Assigns an error message to an optional second output argument. Note that the error messages are system dependent and are not provided for all errors on all systems. The function `ferror` may also provide information about errors.

It's good practice to test the file identifier each time you open a file in your code. For example, this code loops a readable filename is entered.

```
fid=0;
while fid < 1
filename=input('Open file: ','s');
(fid,message) = fopen(filename, 'r');
if fid == -1
disp(message)
end
end
```

Now assume that `nofile.mat` does not exist but that `goodfile.mat` does exist. On one system, the results are

```
Open file: nofile.mat
Cannot open file. Existence? Permissions? Memory? ...
Open file: goodfile.mat
```

Opening Temporary Files and Directories

The `tempdir` and `tempname` commands assist in locating temporary data on your system.

Function Purpose `tempdir` Get temporary directory name. `tempname` Get temporary filename.

You can create temporary files. Some systems delete temporary files every time you reboot the system. On other systems, designating a file as temporary may mean only that the file is not backed up.

A function named `tempdir` returns the name of the directory or folder that has been designated to hold temporary files on your system. For example, issuing `tempdir` on a UNIX system returns the `/tmp` directory.

MATLAB also provides a `tempname` function that returns a filename in the temporary directory. The returned filename is a suitable destination for temporary data. For example, if you need to store some data in a temporary file, then you might issue the following command first.

```
fid = fopen(tempname, 'w');
```

Note The filename that `tempname` generates is not guaranteed to be unique; however, it is likely to be so.

Reading Binary Data

The `fread` function reads all or part of a binary file (as specified by a file identifier) and stores it in a matrix. In its simplest form, it reads an entire file and interprets each byte of input as the

next element of the matrix. For example, the following code reads the data from a file namednickel.dat into matrixA.

```
fid = fopen('nickel.dat','r');
A = fread(fid);
```

To echo the data to the screen after reading it, usechar to display the contents of A as characters, transposing the data so it displays horizontally. `disp(char(A'))`

The char function causes MATLAB to interpret the contents of A as characters instead of as numbers. TransposingA displays it in its more natural horizontal format.

Controlling the Number of Values Read

fread accepts an optional second argument that controls the number of values read (if unspecified, the default is the entire file). For example, this statement reads the first 100 data values of the file specified byfid into the column vectorA.

```
A = fread(fid,100);
```

Replacing the number100 with the matrix dimensions[10 10] reads the same 100 elements into a 10-by-10 array.

Controlling the Data Type of Each Value

An optional third argument to fread controls the data type of the input. The data type argument controls both the number of bits read for each value and the interpretation of those bits as character, integer, or floating-point values. MATLAB supports a wide range of precisions, which you can specify with MATLAB specific strings or their C or Fortran equivalents.

Some common precisions include:

- 'char' and 'uchar' for signed and unsigned characters (usually 8 bits)
- 'short' and'long' for short and long integers (usually 16 and 32 bits, respectively)
- 'float' and'double' for single and double precision floating-point values (usually 32 and 64 bits, respectively)

Note The meaning of a given precision can vary across different hardware platforms. For example, a'uchar' is not always 8 bits.fread also provides a number of more specific precisions, such as'int8' and'float32'. If in doubt, use these precisions, which are not platform dependent. Look upfread in online help for a complete list of precisions.

For example, if fid refers to an open file containing single-precision floating-point values, then the following command reads the next 10 floating-point values into a column vectorA.

```
A = fread(fid,10,'float');
```

Writing Binary Data

The fwrite function writes the elements of a matrix to a file in a specified numeric precision, returning the number of values written. For instance, these lines create a 100-byte binary file containing the 25 elements of the 5-by-5 magic square, each stored as 4-byte integers.

```
fwriteid = fopen('magic5.bin','w');
count = fwrite(fwriteid,magic(5),'int32');
status = fclose(fwriteid);
```

In this case,fwrite sets thecount variable to25 unless an error occurs, in which case the value is less.

Controlling Position in a File

Once you open a file with fopen, MATLAB maintains a file position indicator that specifies a particular location within a file. MATLAB uses the file position indicator to determine where in the file the next read or write operation will begin. The following sections describe how to:

- Determine if the file position indicator is at the end of the file
- Move to specific location in the file
- Retrieve the current location of the file position indicator
- Reset the file position indicator to the beginning of the file

Determining End-of-file

Thefseek andftell functions let you set and query the position in the file at which the next input or output operation takes place:

- Thefseek function repositions the file position indicator, letting you skip over data or back up to an earlier part of the file.
- Theftell function gives the offset in bytes of the file position indicator for a specified file.

The syntax for fseek is

```
status = fseek(fid,offset,origin)
```

fid is the file identifier for the file.offset is a positive or negative offset value, specified in bytes.origin is an origin from which to calculate the move, specified as a string.

'b0f' Beginning of file

'c0f' Current position in file

'e0f' End of file

Understanding File Position

To see howfseek andftell work, consider this short M-file.

```
A = 1:5;
fid = fopen('five.bin','w');
fwrite(fid, A,'short');
status = fclose(fid);
```

This code writes out the numbers 1 through 5 to a binary file named five.bin. The call tofwrite specifies that each numerical element be stored as ashort. Consequently, each number uses two storage bytes.

Now reopen five.bin for reading.

```
fid = fopen('five.bin','r');
```

This call to fseek moves the file position indicator forward 6 bytes from the beginning of the file.
status = fseek(fid,6,'bof');

File Position bof 1 2 3 4 5 6 7 8 9 10 eof File Contents 0 1 0 2 0 3 0 4 0 5 File Position Indicator

This call to fread reads whatever is at file positions 7 and 8 and stores it in variable four.

```
four = fread(fid,1,'short');
```

The act of reading advances the file position indicator. To determine the current file position indicator, call ftell.

```
position = ftell(fid)
```

```
position =
```

```
8
```

File Position bof 1 2 3 4 5 6 7 8 9 10 eof File Contents 0 1 0 2 0 3 0 4⁰ 5 File Position Indicator

This call to fseek moves the file position indicator back 4 bytes.

```
status = fseek(fid,-4,'cof');
```

File Position bof 1 2 3 4 5 6 7 8 9 10 eof File Contents 0 1 0 2 0 3 0 4 0 5 File Position Indicator

Calling fread again reads in the next value (3).

```
three = fread(fid,1,'short');
```

Reading Strings Line-By-Line from Text Files

MATLAB provides two functions, fgetl and fgets, that read lines from formatted text files and store them in string vectors. The two functions are almost identical; the only difference is that fgets copies the newline character to the string vector but fgetl does not.

The following M-file function demonstrates a possible use of fgetl. This function uses fgetl to read an entire file one line at a time. For each line, the function determines whether an input literal string (literal) appears in the line.

If it does, the function prints the entire line preceded by the number of times the literal string appears on the line.

```
function y = litcount(filename, literal)
% Search for number of string matches per line.
```

```
fid = fopen(filename, 'rt');
y = 0;
while feof(fid) == 0

tline = fgetl(fid);
matches = findstr(tline, literal); num = length(matches);
```

```

if num > 0

y = y + num;
fprintf(1,'%d:%s\n',num,tline); end
end
fclose(fid);

```

For example, consider the following input data file calledbadpoem.

Oranges and lemons,
 Pineapples and tea.
 Orangutans and monkeys,
 Dragonflys or fleas.

To find out how many times the string'an'appears in this file, uselitcount.

```

litcount('badpoem','an')
2: Oranges and lemons,
1: Pineapples and tea.
3: Orangutans and monkeys,

```

Reading Form atted ASCII Data

The fscanf function is like thefscanf function in standard C. Both functions operate in a similar manner, reading data from a file and assigning it to one or more variables. Both functions use the same set of conversion specifiers to control the interpretation of the input data.

The conversion specifiers forfscanf begin with a% character; common conversion specifiers include:

- %s to match a string
- %d to match an integer in base 10 format
- %g to match a double-precision floating-point value

You can also specify that fscanf skip a value by specifying an asterisk in a conversion specifier. For example,%*f means skip the floating point value in the input data;%*d means skip the integer value in the input data.

Differences Between the MATLAB fscanf and the C fscanf Despite all the similarities between the MATLAB and C versions offscanf, there are some significant differences. For example, consider a file named moon.dat for which the contents are as follows.

```

3.654234533
2.71343142314
5.34134135678

```

The following code reads all three elements of this file into a matrix named MyData.

```
fid = fopen('moon.dat','r');
MyData = fscanf(fid,'%g');
status = fclose(fid);
```

Notice that this code does not use any loops. Instead, the fscanf function continues to read in text as long as the input format is compatible with the format specifier.

An optional size argument controls the number of matrix elements read. For example, if fid refers to an open file containing strings of integers, then this line reads 100 integer values into the column vector A.

```
A = fscanf(fid,'%5d',100);
```

This line reads 100 integer values into the 10-by-10 matrix A.

```
A = fscanf(fid,'%5d',[10 10]);
```

A related function, sscanf, takes its input from a string instead of a file. For example, this line returns a column vector containing 2 and its square root.

```
root2 = num2str([2, sqrt(2)]);
rootvalues = sscanf(root2,'%f');
```

Writing Formatted Text Files

The fprintf function converts data to character strings and outputs them to the screen or a file. A format control string containing conversion specifiers and any optional text specify the output format. The conversion specifiers control the output of array elements; fprintf copies text directly.

Common conversion specifiers include:

- %e for exponential notation
- %f for fixed point notation
- %g to automatically select the shorter of %e and %f

Optional fields in the format specifier control the minimum field width and precision. For example, this code creates a text file containing a short table of the exponential function.

```
x = 0:0.1:1;
y = [x; exp(x)];
```

The code below writes x and y into a newly created file named exptable.txt.

```
fid = fopen('exptable.txt','w');
fprintf(fid,'Exponential Function\n\n');
fprintf(fid,'%6.2f %12.8f\n',y);
status = fclose(fid);
```

The first call to fprintf outputs a title, followed by two carriage returns. The second call to fprintf outputs the table of numbers. The format control string specifies the format for each line of the table:

- A fixed-point value of six characters with two decimal places
- Two spaces

- A fixed-point value of twelve characters with eight decimal places

fprintf converts the elements of array in column order. The function uses the format string repeatedly until it converts all the array elements. Now usefscanf to read the exponential data file.

```
fid = fopen('exptable.txt','r');
title = fgetl(fid);
[table,count] = fscanf(fid,'%f %f',[2 11]);
table = table';
status = fclose(fid);
```

The second line reads the file title. The third line reads the table of values, two floating-point values on each line until it reaches end of file. count returns the number of values matched.

A function related to fprintf, sprintf, outputs its results to a string instead of a file or the screen. For example,

```
root2 = sprintf('The square root of %f is %10.8e.\n',2,sqrt(2));
```

Closing a File

When you finish reading or writing, usefclose to close the file. For example, this line closes the file associated with file identifier fid.

```
status = fclose(fid);
```

This line closes all open files.

```
status = fclose('all');
```

Both forms return 0 if the file or files were successfully closed or -1 if the attempt was unsuccessful.

MATLAB automatically closes all open files when you exit from MATLAB. It is still good practice, however, to close a file explicitly withfclose when you are finished using it. Not doing so can unnecessarily drain system resources.

Note Closing a file does not clear the file identifier variable fid. However, subsequent attempts to access a file through this file identifier variable will not work.

7

Editing and Debugging M-Files

Starting the Editor/Debugger	7-3
Creating a New M-File in the Editor/Debugger	7-4
Opening Existing M-Files in the Editor/Debugger.....	7-5
Opening the Editor Without Starting MATLAB.....	7-6
Closing the Editor/Debugger	7-6

Creating and Editing M-Files with the Editor/Debugger 7-8

Appearance of an M-File	7-8
-------------------------------	-----

Navigating in an M-File	7-11
Saving M-Files	7-15
Running M-Files from the Editor/Debugger.....	7-15
Printing an M-File.....	7-16
Closing M-Files	7-16
Debugging M-Files.....	7-17
Types of Errors.....	7-17
Finding Errors	7-17
Debugging Example – The Collatz Problem.....	7-18
Using Debugging Features.....	7-22
Preferences for the Editor/Debugger	7-35
General Preferences for the Editor/Debugger	7-36
Font & Colors Preferences for the Editor/Debugger.....	7-37
Display Preferences for the Editor/Debugger.....	7-38
Keyboard and Indenting Preferences for the Editor/Debugger	7-40
Printing Preferences for the Editor/Debugger.....	7-42
There are several methods for creating, editing, and debugging M-files, which are files containing MATLAB code.	

Task Option Instructions

Creating
and Editing M-files

MATLAB Editor “Starting the Editor/Debugger” on page 7-3

MATLAB Editor in stand-alone mode
(without running
MATLAB)

Any text editor, such as Emacs or vi
Debugging M-files
General debugging tips
MATLAB Debugger

MATLAB debugging functions
“Opening the Editor Without Starting MATLAB” on page 7-6

Specify the other editor as the default using preferences – see “Editor” on page 7-36

“Types of Errors” and “Finding Errors” on page 7-17

“Using Debugging Features” on page 7-22

“Using Debugging Features” on page 7-22

Use preferences to set up the editing and debugging environment to best meet your needs.

To learn more about writing M-files, see “Programming and Data Types”.

Starting the Editor / Debugger

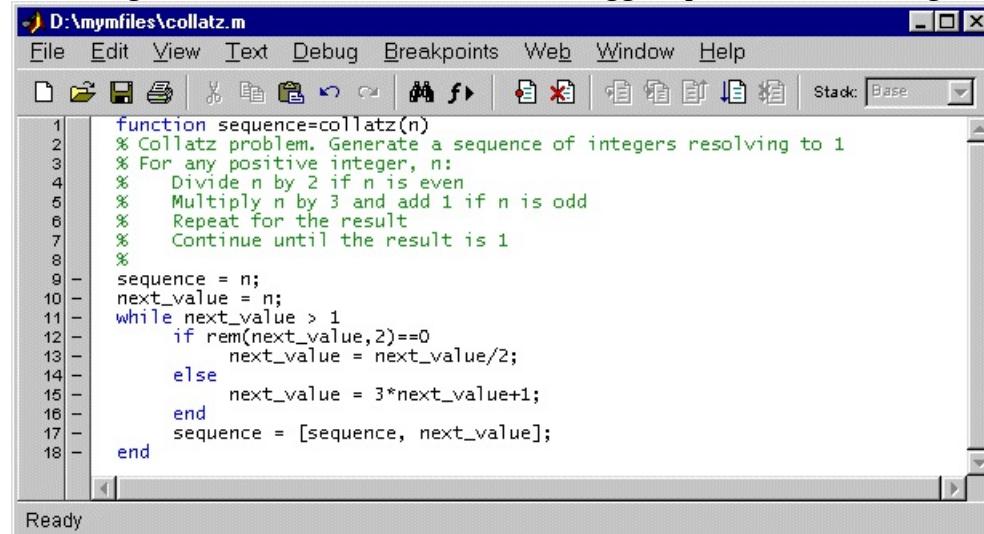
The MATLAB Editor/Debugger provides a graphical user interface for basic text editing features for any file type, as well as for M-file debugging. The Editor/Debugger is a single tool that you can use for editing, debugging, or both. There are various ways to start the Editor/Debugger – see the sections:

- “Creating a New M-File in the Editor/Debugger” on page 7-4
- “Opening Existing M-Files in the Editor/Debugger” on page 7-5
- “Opening the Editor Without Starting MATLAB” on page 7-6 (no Debugger)

After starting the Editor/Debugger, follow the instructions for:

- “Creating and Editing M-Files with the Editor/Debugger” on page 7-8
- “Debugging M-Files” on page 7-17
- “Closing the Editor/Debugger” on page 7-6

Following is an illustration of the Editor/Debugger opened to an existing M-file.



D:\mymfiles\collatz.m

```
function sequence=collatz(n)
% Collatz problem. Generate a sequence of integers resolving to 1
% For any positive integer, n:
%   Divide n by 2 if n is even
%   Multiply n by 3 and add 1 if n is odd
%   Repeat for the result
%   Continue until the result is 1
%
sequence = n;
next_value = n;
while next_value > 1
    if rem(next_value,2)==0
        next_value = next_value/2;
    else
        next_value = 3*next_value+1;
    end
    sequence = [sequence, next_value];
end
```

Ready

If the Editor/Debugger window is not wide enough, the toolbar buttons on the right will not be shown. The menu will wrap, and all toolbar functions are available from equivalent menu items. To see all toolbar buttons, make the Editor/Debugger window wider.

To dock the Editor/Debugger inside the MATLAB desktop, select **Dock M-File** from the **View** menu.

To change the default appearance and behavior of the Editor/Debugger, follow the instructions in “Preferences for the Editor/Debugger” on page 7-35.

Creating a New M-File in the Editor / Debugger

To create a new M-file in the Editor/Debugger, either click the new file button



on the MATLAB toolbar, or select **File -> New -> M-file** from the MATLAB desktop. You can also create a new M-file using the context menu in the

Current Directory browser – see “Creating New Files” on page 5-24. The Editor/Debugger opens, if it is not already open, with a blank file in which you can create an M-file.

If the Editor/Debugger is open, create more new files by using the new file button  on the toolbar, or select **File -> New -> M-file**.

Function Equivalent

Typeedit in the Command Window to create a new M-file in the Editor/ Debugger.

If you typeedit filename.m andfilename.m does not yet exist, a prompt appears asking if you want to create a new file titled filename.m.

- If you click **Yes**, the Editor/Debugger creates a blank file titledfilename.m. If you do not want the dialog to appear in this situation, either check that box in the dialog or specify it in preferences for “Prompt” on page 7-40. In that case, the next time you typeedit filename.m, the file will be created without first prompting you.
- If you click **No**, the Editor/Debugger does not create a new file. If you do not want the dialog to appear in this situation, either check that box in the dialog or specify it in preferences for “Prompt” on page 7-40. In that case, the next time you type edit filename.m, a “file not found” error will appear.

Opening Existing M-Files in the Editor / Debugger

To open an existing M-file in the Editor/Debugger, click the open button  on

the MATLAB or Editor/Debugger toolbar, or select **File -> Open**. Then, from the **Open** dialog box, select the M-file and click **Open**. You can also open files from the Current Directory browser – see “Opening Files” on page 5-26.

You can select a file to open from the most recently used files, which are listed at the bottom of the **File** menu in the desktop as well as in the Editor/Debugger. You can change the number of files appearing on the list – see “Preferences for the Editor/Debugger” on page 7-35.

If the Editor/Debugger is not already open, it opens with the file displayed. If it is already open, the file appears either in its own window or as a tab in the current window as specified in “Opening files in editor” on page 7-38. To make a document in the Editor/Debugger become the current document, click on it or use the **Window** menu or tabs.

You can set a preference that instructs MATLAB on startup to automatically open the files that were open when the previous MATLAB session ended. For instructions, see the **On restart** preference in “General Preferences for the Editor/Debugger” on page 7-36.

Function Equivalent

Use theedit oropen function to open an existing M-file in the Editor/ Debugger. For example,

type

edit collatz.m

to open the filecollatz.m in the Editor/Debugger.

Opening a Selection

Within a file in the Editor/Debugger, select a function, right-click, and select **Open Selection** from the context menu. The file opens in the Editor/Debugger.

Getting Help for a Function

Within a file in the Editor/Debugger, select a function, right-click, and select **Help on Selection** from the context menu. The reference page for that function opens in the Editor/Debugger, or if the reference page does not exist, the M-file help is shown instead.

Accessing Your Source Control System

If you use a source control system for M-files, you can access it from within the Editor/Debugger to check out files. See Chapter 9, “Interfacing with Source Control Systems.”

Opening the Editor Without Starting MATLAB

On Windows platforms, you can use the MATLAB Editor without starting MATLAB. To do so, double-click an M-file in Windows Explorer. The M-file opens in the MATLAB Editor. To open the Editor without a file, open \$matlabroot/bin/win32/meditor.exe. Regardless of the type of MATLAB license you have, you can open multiple instances of meditor because it is not considered an instance of MATLAB.

When you open the MATLAB Editor without starting MATLAB, the Editor is a stand-alone application. You cannot debug M-files from it, evaluate a selection, access source control features, dock the Editor in the MATLAB desktop, nor access help from it. It remains a stand-alone application, even if you subsequently open MATLAB. Other than these limitations, you can use the editing features as described in “Creating and Editing M-Files with the Editor/ Debugger” on page 7-8.

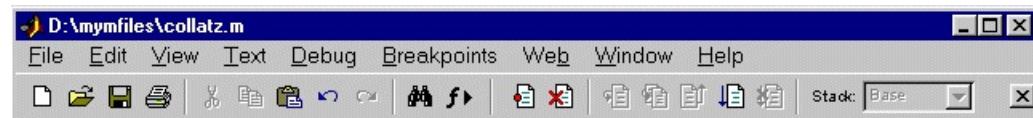
For Windows platforms, when MATLAB is installed, the stand-alone Editor is automatically associated with files having a.m extension. If you double-click on an M-file, the stand-alone Editor opens. You can change the association using Windows Explorer so that files with a.m extension will open in the Editor/Debugger in MATLAB.

Closing the Editor / Debugger

To close the Editor/Debugger, click the close box in the title bar of the Editor/ Debugger. This is different from the close box in the toolbar of the Editor/ Debugger, which closes the current file when multiple files are open in a single window.

CloseboxforEditor/Debugger

Closebox forcurrent file



When you close the current file and it is the only file open, then the Editor/ Debugger closes as well.

If multiple files are open with each in a separate Editor/Debugger window, close each window separately or close all of the files at once using the **Close All** item in the **Window** menu.

When you close the Editor/Debugger and any of the open files have unsaved changes, you will be prompted to save the files.

Creating and Editing M-Files with the Editor / Debugger

After opening an existing file or creating a new file, enter text in the Editor/ Debugger. Follow the same rules you would use for entering text in the Command Window as described in these sections:

- “Case and Space Sensitivity” on page 3-4
- “Entering Multiple Functions in a Line” on page 3-4
- “Entering Long Lines” on page 3-5
- “Suppressing Output” on page 3-9
- “Controlling the Format and Spacing of Numeric Output” on page 3-10

Use the editing features described in these sections:

- “Appearance of an M-File” on page 7-8, including syntax highlighting
- “Navigating in an M-File” on page 7-11, including go to and find features
- “Saving M-Files” on page 7-15
- “Running M-Files from the Editor/Debugger” on page 7-15
- “Printing an M-File” on page 7-16
- “Closing M-Files” on page 7-16

Appearance of an M-File

The following features make M-files more readable:

- “Syntax Highlighting” on page 7-8
- “Indenting” on page 7-9
- “Commenting” on page 7-9
- “Showing Balanced Delimiters” on page 7-10

You can specify the default behavior for many of them – see “Preferences for the Editor/Debugger” on page 7-35.

Syntax Highlighting

Some entries appear in different colors to help you better find matching elements, such as if/else statements. For more information, see “Syntax Highlighting” on page 3-5.

Indenting

Flow control entries are automatically indented to aid in reading the loops, such as while/end statements.

To move the current or selected lines further to the left, select **Decrease Indent** from the **Text** menu. To move the current or selected lines further to the right, select **Increase Indent** from the **Text** menu. If after using these features you want to apply automatic indenting to selected lines, select **Smart Indent** from the **Text** menu, or right-click and select it from the context menu. For more information about smart indenting, see the preference for smart indent.

Commenting

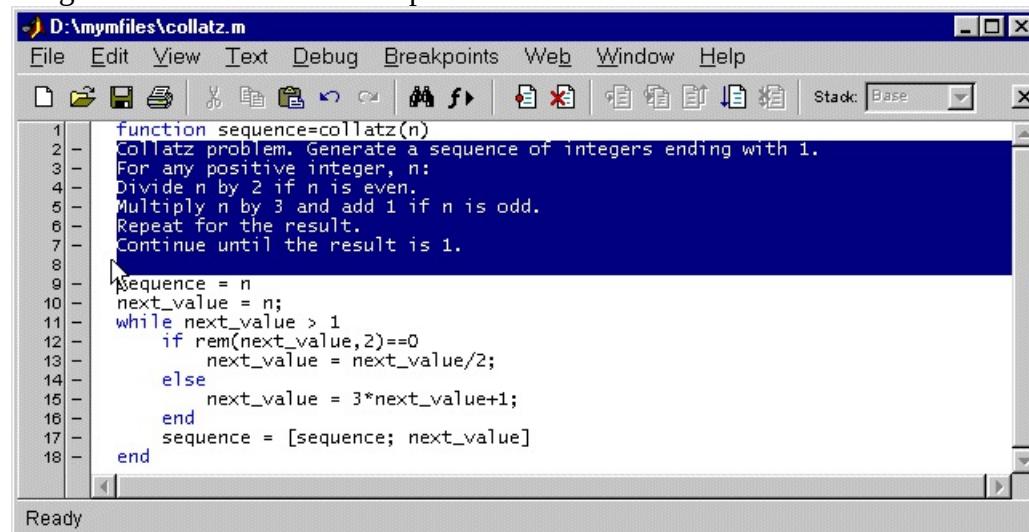
You can comment the current line or a selection of lines. To select a line, click just to the left of the line – the line becomes highlighted. Drag or **Shift** +click to select multiple lines. Then select **Comment** from the **Text** menu, or right-click and select it from the context menu. A comment symbol, %, is added at the start of the line, and the color of the text becomes green.

You can make any line a comment by typing a % at the beginning of it. To put a comment within a line, type % followed by the comment text; MATLAB treats all the information after the % on that line as a comment.

You can also uncomment a selected group of lines – select **Uncomment** from the **Text** menu, or right-click and select it from the context menu.

Click in the area to the left of a line to select that line.

Drag or Shift+click to select multiple lines as shown here.



```
D:\myfiles\collatz.m
File Edit View Text Debug Breakpoints Web Window Help
D:\myfiles\collatz.m | 1 function sequence=collatz(n)
2 % Collatz problem. Generate a sequence of integers ending with 1.
3 % For any positive integer, n:
4 % Divide n by 2 if n is even.
5 % Multiply n by 3 and add 1 if n is odd.
6 % Repeat for the result.
7 % Continue until the result is 1.
8 %
9 % Sequence = n
10 next_value = n;
11 while next_value > 1
12     if rem(next_value,2)==0
13         next_value = next_value/2;
14     else
15         next_value = 3*next_value+1;
16     end
17     sequence = [sequence; next_value]
18 end
Ready Showing
```

Balanced Delimiters

When you position the cursor inside a pair of delimiters, that is, inside (), [], or { }, and select **Balanced Delimiters** from the **Text** menu, the string inside the pair of delimiters is highlighted. In this example, when the cursor is positioned before /norm, as indicated here

`alfa = asin(T*v'./sqrt(diag(T*T')))/norm(v)` selecting **Balanced Delimiters** highlights the selection as shown here.

`alfa = asin(T*v'./sqrt(diag(T*T')))/norm(v)` **Navigating in an M-File**

There are several options for navigating in M-files:

- “Going to a Line Number”
- “Going to a Bookmark”
- “Going to a Function”
- “Finding a Selection”
- “Finding and Replacing a String”

Going to a Line Number

Line numbers are displayed along the left side of the Editor/Debugger window. Select **Go to**

Line from the **Edit** menu. Enter the line number in the edit box and click **OK**. The cursor moves to that line number in the current M-file.

Going to a Bookmark

You can set a bookmark at a line in an M-file so you can quickly go to the bookmarked line. This is particularly useful in long M-files. For example, if you're working on a line but need to look at another part of the file, set a bookmark at the current line, go to the other part of the file, and then go back to the bookmark.

To set a bookmark, position the cursor anywhere in the line and select **Set Bookmark** from the **Edit** menu. A bookmark icon appears to the left of the line.

 To go to a bookmark, select **Next Bookmark** or **Previous Bookmark** from the **Edit** menu.

To clear a bookmark, position the cursor anywhere in the line and select **Clear Bookmark** from the **Edit** menu.

Bookmarks are not saved when you close a file.

Going to a Function

To go to a function in an M-file (referred to as a subfunction), click the function

button

 on the toolbar. Select the function you want to go to from the list of

all functions in that M-file. The list does not include functions that are called from the M-file, but only lists lines in the current M-file that begin with a function statement.

Finding a Selection

Within the current M-file, select a string. From the **Edit** menu, select **Find Selection**. The next occurrence of that string becomes highlighted. Use **Find Selection** from the **Edit** menu to continue finding the next occurrences of the string.

To find the previous occurrence of the selected string (find backwards), press **Ctrl+Shift+F 3**.

Finding and Replacing a String

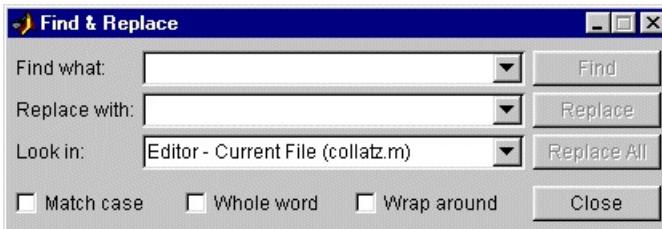
You can search for a specified string within multiple files, and replace the string within a file.

Finding a String. To search for a string in files:

1 Click the find button

 on the Editor/Debugger toolbar, or select **Find and Replace** from the **Edit** menu.

The **Find & Replace** dialog box appears. This provides the same features as the **Find & Replace** dialog box accessible from the Current Directory browser.

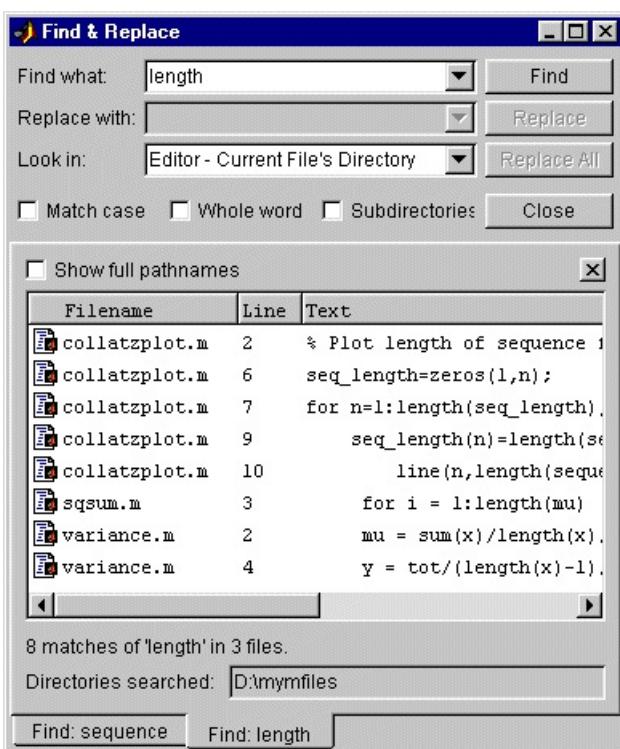


2 Complete the **Find & Replace** dialog box to find all occurrences of the string you specify.

- a** Type the string in the **Find w hat** field.
 - b** Select the files to search through from the **Look in** listbox.
 - c** Limit the search using **Match case**, **Whole w ord**, or **Wrap around**. These settings are remembered for your next MATLAB session.
- 3** Click **Find**.
- If you search for the current file, the next occurrence of the string is highlighted in the file.
 - If the search is through multiple files, results appear in the lower part of the **Find & Replace** dialog box and include the filename, M-file line number, and content of that line.

Double-click on a file in the results list to open it.

The results of previous find operations are available via tabs.



Close the results section.

- 4** Open any M-file(s) in the results list by doing one of the following:
- Double-clicking it.

- Selecting it and pressing the **Enter** or **Return** key.
- Right-clicking it and selecting **Open** from the context menu.

The M-file opens, scrolled to the line number shown in the results section of the **Find & Replace** dialog box.

5 If you perform another search, the results of each search are accessible via tabs just below the results list. Click a tab to see that results list as well as the search criteria.

Finding the Next or Previous Occurrence of the String. To find the next occurrence of the string you entered in the **Find & Replace** dialog box, select **Find Next** from the **Edit** menu. To find the previous occurrence of that string (find backwards), press **Shift+F 3**.

Function Equivalent. Use `lookfor` to search for the specified string in the first line of help in all M-files on the search path.

Replacing a String. After searching for a string within files, you can replace the specified content in the current file.

1 Open the file in the Editor if it's not already open. You can open it from the **Find & Replace** dialog box – see step 4 in “Finding a String” on page 7-12. Be sure that the file in which you want to replace the string is the current file in the Editor.

2 Be sure the **Look in** field in the **Find & Replace** dialog box shows the name of the file in which you want to replace the string. The **Replace** button in the **Find & Replace** dialog box becomes selectable.

3 In the **Replace with** field, type the text that is to replace the specified string. **4 Click Replace** to replace the string in the selected line, or click **Replace All** to replace all instances in the currently open file.

The text is replaced.

5 To save the changes, select **Save** from the **File** menu.
You can repeat this for multiple files.

Saving M-Files

After making changes to an M-file, you will see an asterisk (*) next to the file name in the title bar of the Editor/Debugger. This indicates there are unsaved changes to the file.

To save the changes, use one of the **Save** commands in the **File** menu:

- **Save**— Saves the file to its existing name. If the file was newly created, the **Save file as** dialog box opens, where you assign a name to the file and save it. Another way to save is by using the save button  on the toolbar.

- **Save As**— The **Save file as** dialog box opens, where you assign a name to the file and save it. You do not need to type the .m extension because MATLAB automatically assigns the .m

extension to the filename. If you do not want an extension, type a. (period) after the filename.

- **Save All**— Saves all named files to their existing filenames. For all newly created files, the **Save file as** dialog box opens, where you assign a name to each file and save it.

Running M-Files from the Editor / Debugger

You can run a script, that is, an M-file that doesn't require an input argument, directory from the Editor/Debugger by clicking the run button

on the toolbar, or by selecting **Run** from the **Debug** menu.

If the file is not in a directory on the search path or in the current directory, a dialog box appears presenting you with options that allow you to run the file. You can either change the current directory to the directory containing the file, or you can add to the search path the directory containing the file.

Note that if the file has unsaved changes, running it from the Editor/Debugger automatically saves the changes before running. In that event, the menu item becomes **Save and Run**.

See “Running an M-File with Breakpoints” on page 7-24 for additional information about running M-files while debugging.

Accessing Your Source Control System

If you use a source control system for M-files, you can access it from within the Editor/Debugger to check in files. See Chapter 9, “Interfacing with Source Control Systems.”

Printing an M-File

To print an entire M-file, select **Print** from the **File** menu, or click the print button

on the toolbar. To print the current selection, select **Print Selection** from the **File** menu. Complete the standard print dialog box that appears.

See also “Printing Preferences for the Editor/Debugger” on page 7-42 to specify a different font for printing, color or black and white printing, and other options.

Closing M-Files

To close the current M-file, select **Close filename** from the **File** menu. If there are multiple files open in a single Editor/Debugger window, click the close box in the Editor toolbar to close the current M-file. This is different from the close box in the titlebar of the Editor/Debugger, which closes the Editor/Debugger, including all open files.

CloseboxforEditor/Debugger

Closeboxforcurrent file



If each file is open in a separate Editor/Debugger window, close all of the files at once using the **Close All** item in the **Window** menu.

When you close the current file and it is the only file open, then the Editor/ Debugger closes as

well.

When you close a file which has unsaved changes, you are prompted to save the file.

Debugging M-Files

This section introduces general techniques for finding errors, and then illustrates MATLAB debugger features found in the Editor/Debugger and debugging functions using a simple example. It includes these topics:

- “Types of Errors” on page 7-17
- “Finding Errors” on page 7-17
- “Debugging Example – The Collatz Problem” on page 7-18
- “Trial Run for Example” on page 7-20
- “Using Debugging Features” on page 7-22

Types of Errors

Debugging is the process by which you isolate and fix problems with your code. Debugging helps to correct two kinds of errors:

- Syntax errors – For example, misspelling a function name or omitting a parenthesis. “Syntax Highlighting” on page 7-8 helps you identify these problems. MATLAB detects most syntax errors and displays an error message in the Command Window describing the error and showing its line number in the M-file. Position the cursor over the error message and press **Ctrl+Enter** to open the M-file at that line. Use `thePCODE` function to check for syntax errors in your M-file without running the M-file.
- Run-time errors – These errors are usually algorithmic in nature. For example, you might modify the wrong variable or perform a calculation incorrectly. Run-time errors are apparent when an M-file produces unexpected results.

Finding Errors

Usually, it’s easy to find syntax errors based on MATLAB’s error messages. Run-time errors are more difficult to track down because the function’s local workspace is lost when the error forces a return to the MATLAB base workspace. Use the following techniques to isolate the cause of run-time errors:

- Remove selected semicolons from the statements in your M-file. Semicolons suppress the display of intermediate calculations in the M-file. By removing the semicolons, you instruct MATLAB to display these results on your screen as the M-file executes.
- Add keyboard statements to the M-file. Keyboard statements stop M-file execution at the point where they appear and allow you to examine and change the function’s local workspace. This mode is indicated by a special prompt, “`K>>`.” Resume function execution by typing `return` and pressing the **Return** key.
- Comment out the leading function declaration and run the M-file as a script. This makes the intermediate results accessible in the base workspace.
- Use `thedefun` function to see the dependent functions.
- Use the MATLAB Editor/Debugger or debugging functions. They are useful for correcting run-time errors because you can access function workspaces and examine or change the values they

contain. You can set and clear *breakpoints*, lines in an M-file at which execution halts. You can change workspace contexts, view the function call stack, and execute the lines in an M-file one by one.

The remainder of this section on debugging M-files describes the use of the Editor/Debugger and debugging function using an example.

Debugging Example—The Collatz Problem

The example debugging session requires you to create two M-files, collatz.m and collatzplot.m, that produce data for the Collatz problem. For any given positive integer, n, the Collatz function produces a sequence of numbers that always resolves to 1. If n is even, divide it by 2 to get the next integer in the sequence. If n is odd, multiply it by 3 and add 1 to get the next integer in the sequence. Repeat the steps for the next integer in the sequence until the next integer is 1. The number of integers in the sequence varies, depending on the starting value, n.

The Collatz problem is to prove that the Collatz function will resolve to 1 for all positive integers. The M-files for this example are useful for studying the problem. The file collatz.m generates the sequence of integers for any given n. The file collatzplot.m calculates the number of integers in the sequence for any given n and plots the results. The plot shows patterns that can be further studied.

Following are the results when n is 1, 2, or 3.

n Sequence Number of Integers in the Sequence

```
1 1 1  
2 2 1 2  
3 3 10 5 16 8 4 2 1 8
```

M-Files for the Collatz Problem

Following are the two M-files you use for the debugging example. To create these files on your system, open two new M-files. Select and copy the following code from the Help browser and paste it into the M-files. Save and name the files collatz.m and collatzplot.m. Be sure to save them to your current directory or add to the search path the directory where you save them. One of the files has an embedded error for purposes of illustrating the debugging features.

Code for collatz.m.

```
function sequence=collatz(n)  
% Collatz problem. Generate a sequence of integers resolving to 1 % For any positive integer, n:  
% Divide n by 2 if n is even  
% Multiply n by 3 and add 1 if n is odd  
% Repeat for the result  
% Continue until the result is 1%  
sequence = n;  
next_value = n;  
while next_value > 1  
  
if rem(next_value,2)==0  
next_value = next_value/2;
```

```

else
next_value = 3*next_value+1;
end
sequence = [sequence, next_value]; end
Code for collatzplot.m.

function collatzplot(n)
% Plot length of sequence for Collatz problem
% Prepare figure
clf
set(gcf,'DoubleBuffer','on')
set(gca,'XScale','linear')
%
% Determine and plot sequence and sequence length for m = 1:n

plot_seq = collatz(m);
seq_length(m) = length(plot_seq);
line(m,plot_seq,'Marker','.', 'MarkerSize',9,'Color','blue') drawnow

end

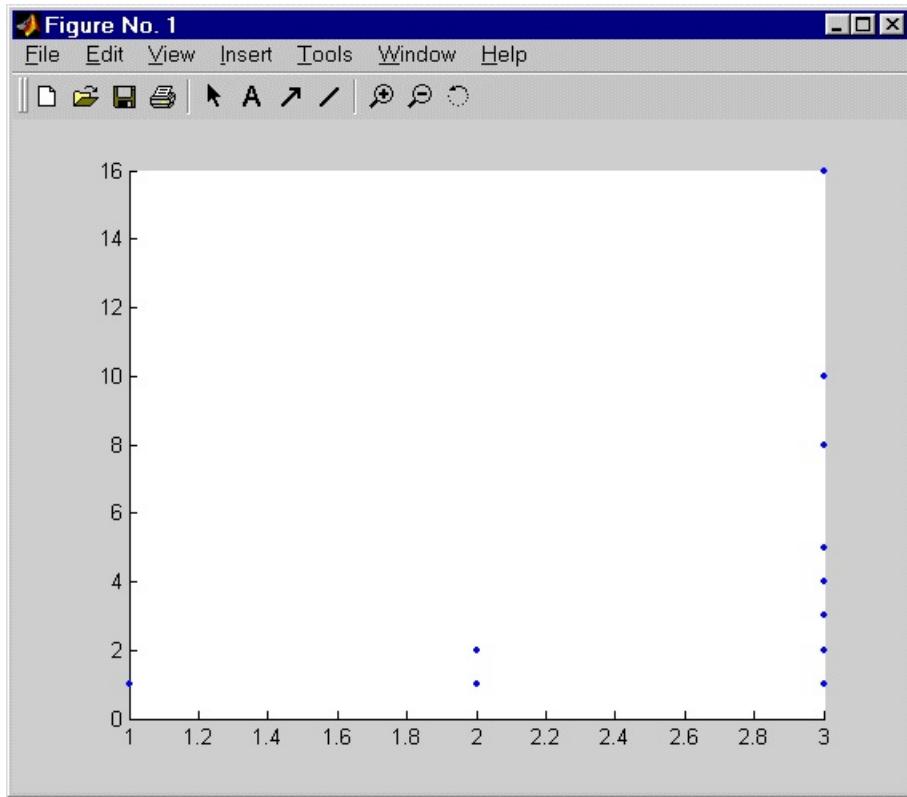
```

Trial Run for Example

Try out `collatzplot` to see if it works correctly. Use a simple input value, for example, 3, and compare the results to those shown in the preceding table. Typing

```
collatzplot(3)
```

produces the plot shown in the following figure.



The plot for 1 appears to be correct – when $n = 1$, the Collatz series is 1, and contains one integer. But for $n = 2$ and $n = 3$ it is wrong because there should be only one value plotted for each integer, the length of the sequences, which the preceding table shows to be 2 and 8 respectively. Instead, multiple values are plotted. Use MATLAB debugging features to isolate the problem.

Using Debugging Features

You can debug the M-files using the Editor/Debugger and debugging functions. You can use both methods interchangeably. The example describes both methods.

The debugging process consists of:

- “Preparing for Debugging” on page 7-22
- “Setting Breakpoints” on page 7-22
- “Running an M-File with Breakpoints” on page 7-24
- “Stepping Through an M-File” on page 7-25
- “Examining Values” on page 7-26
- “Correcting Problems and Ending Debugging” on page 7-29

Preparing for Debugging

Do the following to prepare for debugging:

- Open the file – To use the Editor/Debugger for debugging, open it with the file you will run, in this example, `collatzplot.m`.

- Save changes – If you are editing the file, save the changes before you begin debugging. If you try to run and debug a file with unsaved changes, the file is automatically saved before it runs.
- Add the file(s) to a directory on the search path or be sure it is in the current directory – Be sure the file you run and any files it calls are in directories that are on the search path. If all files to be used are in the same directory, you can instead make that directory be the current directory.

Setting Breakpoints

Set breakpoints to pause execution of the function so you can examine values where you think the problem might be. You can only set breakpoints at executable lines in saved files that are in the current directory or in directories on the search path. When you create a new M-file, save it before setting breakpoints. You cannot set breakpoints while MATLAB is busy, for example, running an M-file.

Breakpoints for the Example. It is unclear whether the problem in the example is in `collatzplot` or `collatz`. To start, set breakpoints `collatzplot.m` at lines 10, 11, and 12. The breakpoint at line 10 allows you to step into `collatz` to see if the problem might be there. The breakpoints at lines 11 and 12 stop the program where you can examine the interim results.

Setting Breakpoints Using the Editor/Debugger. To set a breakpoint using the Editor/Debugger, click in the breakpoint alley at the line where you want to set the breakpoint. The breakpoint alley is the column to the right of the line number. You can set breakpoints only at lines that are preceded by a - (dash). Lines not preceded by a dash, such as comments, are not executable. Other ways to set a breakpoint are to position the cursor in the line and then click the set/clear

breakpoint button

on the toolbar, or select **Set/Clear Breakpoint** from the **Breakpoints** menu or the context menu.

A breakpoint icon appears, as in the following illustration for line 10.

 `plot_seq = collatz(m);` **Function Equivalent.** To set a breakpoint using the debugging functions, use `dbstop`. For the example, type

```
dbstop in collatzplot at 10
dbstop in collatzplot at 11
dbstop in collatzplot at 12
```

Some useful related functions are:

- `dbtype`– Lists the M-file with line numbers in the Command Window.
- `dbstatus`– Lists breakpoints.

Setting Stops for Conditions. Use the items on the **Breakpoints** menu or the `dbstop` function equivalents to instruct the program to stop when it encounters a problem. For details, see `dbstop`.

- **Stop If Error**, `ordbstop if error`
- **Stop If Warning**, `ordbstop if warning`
- **Stop If NaN Or Inf** (for not-a-number or infinite value), `ordbstop if naninf` `ordbstop if infnan`
- **Stop If All Error**, `ordbstop if all error`

If the File Is Not on the Search Path or In the Current Directory. When you add a breakpoint to a file that is not in a directory on the search path or in the current directory, a dialog box appears presenting you with options that allow you to add the breakpoint. You can either change the current directory to the directory containing the file, or you can add to the search path the directory containing the file.

Running an M-File with Breakpoints

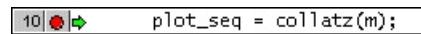
After setting breakpoints, run the M-file from the Command Window or the Editor/Debugger.

For the example, runcollatzplot for the simple input value, 3, by typing in the Command Window
collatzplot(3)

The example, collatzplot, requires an input argument and therefore runs only from the Command Window and not from the Editor/Debugger.

Results of Running an M-File Containing Breakpoints. Running the M-file results in the following:

- The prompt in the Command Window changes to `K>>`, indicating that MATLAB is in debug mode.
- The program is paused at the first breakpoint, which in the example is line 10. This means that line 10 will be executed when you continue. The pause is indicated in the Editor/Debugger by the green arrow just to the right of the breakpoint as shown here.



10 plot_seq = collatz(m);

If you use debugging functions and have the Debugger options preference for **Command Window w debugging** checked, the line at which you are paused is displayed in the Command Window. For the example, it would show

10 plot_seq = collatz(m);

- The function displayed in the **Stack** field on the toolbar changes to reflect the current function. If you use debugging functions, use `dbstack` to view the current call stack. The call stack includes subfunctions as well as called functions.
- If the file you're running is not in the current directory or in a directory on the search path, you're prompted to either add the directory to the path or change the current directory.

Stepping Through an M-File

While in debug mode, you can step through an M-file, pausing at points where you want examine values.

Use the step buttons or the step items in the **Debug** menu of the Editor/ Debugger, or use the equivalent functions.

Toolbar Debug Button Menu Item Description Function Equivalent

 **Continue**

None **Go Until Cursor**

 **Step**

 **Step In**

 **Step Out**

Continue execution of M-file until completion or until another breakpoint is encountered. The menu item says **Run** or **Save and Run** if the file is not running.

Continue execution of M-file until the line where the cursor is positioned. Also available on the context menu.

Execute the current line of the M-file.

Execute the current line of the M-file and, if the line is a call to another function, step into that function.

After stepping in, runs the rest of the called function or subfunction, leaves the called function, and pauses.

dbcont

None

dbstep

dbstep in

dbstep out

Stepping In. In the example, *collatzplot* is paused at line 10. Use the step-in button or type *dbstep in* in the Command Window to step into *collatz* and walk through that M-file. Stepping in takes you to line 9 of *collatz*.

The pause indicator at line 10 of *collatzplot* changes to a hollow arrow , indicating that MATLAB control is now in a function called from the main program, which in the example is *collatz*.

In the called function, you can do the same things you can do in the main (calling) function – set breakpoints, run, step through, and examine values.

Stepping Out. In the example, the program is paused at step 9 in *collatz*. Because the problem results are correct for *n = 1*, continue running the program since there is no need to examine values until *n = 2*. The fastest way to run through *collatz* is to step out, which runs the rest of *collatz* and returns to the next line in *collatzplot*, line 11. To step out, use the step-out button or type *dbstep out* in the Command Window.

Examining Values

While the program is paused, you can view the value of any variable currently in the workspace. Use the following methods to examine values:

- “Where to Examine Values” on page 7-26
- “Selecting the Workspace” on page 7-27
- “Viewing Datatips in the Editor/Debugger” on page 7-27
- “Viewing Values in the Command Window” on page 7-28
- “Viewing Values in the Array Editor” on page 7-28
- “Evaluating a Selection” on page 7-28

Many of these methods are used in “Examining Values in the Example” on page 7-28.

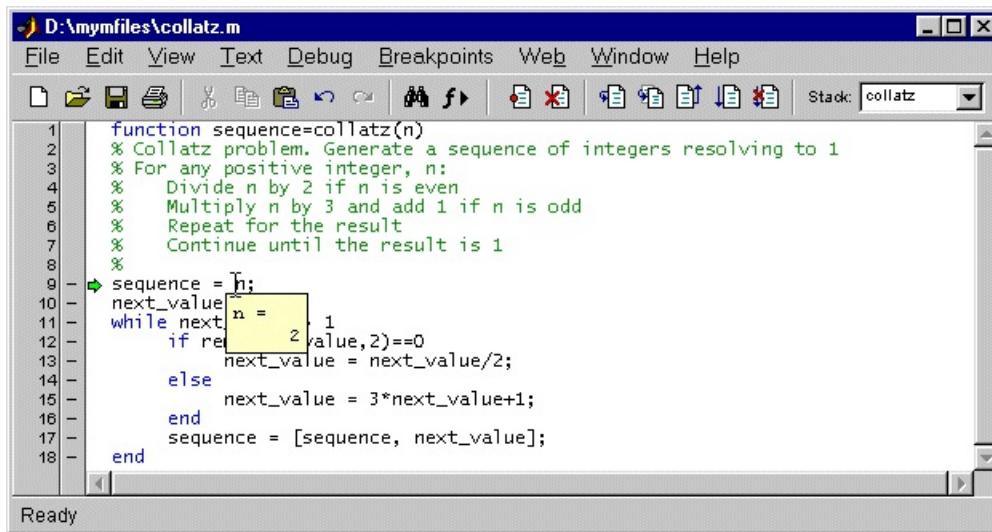
Where to Examine Values. When the program is paused, either at a breakpoint or at a line you have stepped to, you can examine values. Examine values when you want to see if a line of code has produced the expected result or not. If the result is as expected, continue running or step to the next line. If the result is not as expected, that line, or a previous line, contains an error.

In the example, because the results for $n = 1$ are correct, there is no need to examine values until $n = 2$. Therefore, continue or step through the first iteration of the loop in `collatzplot` when $m = 1$. When `collatzplot` stops at line 10 the next time ($whenm = 2$), step in to `thecollatz` function so you can examine values there.

Selecting the Workspace. Variables assigned through the Command Window are considered to be the base workspace. Variables created in each function have their own workspace. To examine a variable, you must first select its workspace. When you run a program, the current workspace is shown in the **Stack** field. To examine values that are part of another function workspace currently running or the base workspace, first select that workspace from the list in the **Stack** field.

Viewing Datatips in the Editor/Debugger. In the Editor/Debugger, position the cursor to the left of a variable. Its current value appears and stays in view until you move the cursor – this is called a datatip. In the example, position the cursor over `n` in `collatz` – the datatip shows that $n = 2$, as expected. Note that the **Stack** shows `collatz` as the current function.

Hold the cursor over
a variable to view its
current value.



Viewing Values in the Command Window. Type a variable name in the Command Window and MATLAB displays its current value. To see the variables currently in the workspace, use `whos`. To see the value of `n` for the example, type

`n`

and MATLAB returns the expected result

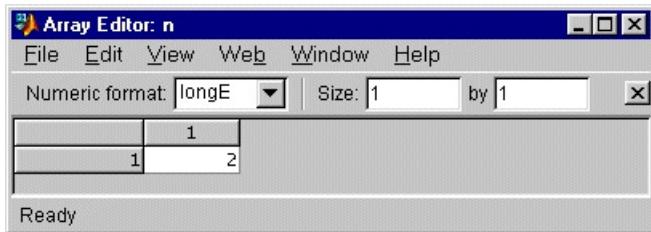
```
n =  
2
```

Viewing Values in the Array Editor. You can view the value of any variable in the Array Editor. To view the current variables, open the Workspace browser. In the Workspace browser, double-click a variable – the Array Editor opens, displaying the value for that variable. You can also open the Array Editor for a variable using `openvar`.

To see the value of `n` in the Array Editor for the example, type

```
openvar n
```

and the Array Editor opens, showing that `n = 2` as expected.



Evaluating a Selection. Select a variable or equation in an M-file in the Editor/ Debugger. Right-click and select **Evaluate Selection** from the context menu. MATLAB displays the value of the variable or equation in the Command Window. You cannot evaluate a selection while MATLAB is busy, for example, running an M-file.

Examining Values in the Example. In `collatz`, use the step button or the function `dbstep`. The program advances to line 10, where there is no need to examine values. Continue stepping until line 13.

When you step again, the pause indicator jumps to line 17, just after the if loop, as expected, given the code in line 13 `fornext_value = 2`. When you step again, you can check the value of `sequence` in line 17 and see that it is `[2 1]` as expected for `n = 2`. Stepping again takes you from line 17 to line 10. Because `next_value` is 1, the while loop ends. The pause indicator is at line 10 and

appears as a green down arrow

• This indicates that processing in the called function is complete and program control will return to the calling program, in this case, `collatzplot` line 10.

In `collatzplot`, step again to advance to line 11, then line 12. The variable `sequence_length` in line 11 is a vector with the elements `[1 2]`, which is correct.

Finally, step again to advance to line 13. Examining the values in line 12, `m = 2` as expected, but the second variable, `plot_seq`, has two values, where only one value is expected. While `plot_seq` has the value expected, `[2 1]`, it is the incorrect variable for plotting. Instead, the variable `seq_length` should be plotted.

Correcting Problems and Ending Debugging

These are some of the ways to correct problems and end the debugging session:

- “Changing Values and Checking Results” on page 7-29
- “Ending Debugging” on page 7-30
- “Clearing Breakpoints” on page 7-30
- “Correcting an M-File” on page 7-31

Many of these features are used in “Completing the Example” on page 7-31.

Changing Values and Checking Results. While debugging, you can change the value of a variable in the current workspace to see if the new value produces expected results. While the program is paused, assign a new value to the variable in the Command Window or in the Array Editor. Then continue running or stepping through the program. If the new value does not produce the expected results, the program has a different or another problem.

Ending Debugging. After identifying a problem, end the debugging session. You must end a debugging session if you want to change an M-file to correct a problem or if you want to run other functions in MATLAB.

Note Always quit debug mode before editing an M-file. If you edit an M-file while in debug mode, you can get unexpected results when you run the file.

To end debugging, click the exit debug mode icon

 or select **Exit Debug**

Mode from the **Debug** menu.

You can instead use the function `dbquit` to end debugging.

After quitting debugging, the pause indicators in the Editor/Debugger display no longer appear, and the normal prompt `>>` now appears in the Command Window instead of the debugging prompt `K>>`. You can no longer access the call stack.

Clearing Breakpoints. Breakpoints remain in a file until you clear them. Clear the breakpoints if you want the program to run uninterrupted, such as after identifying and correcting a problem.

To clear a breakpoint in the Editor/Debugger, click on the breakpoint icon for a line, or select **Set/Clear Breakpoint** from the **Breakpoints** or context menu. The breakpoint for that line is cleared.

To clear all breakpoints in all files, select **Clear All Breakpoints** from the **Breakpoints** menu, or click the

 equivalent button on the toolbar.

The function that clears breakpoints is `dbclear`. To clear all breakpoints, use `dbclear all`. For the example, clear all of the breakpoints in `collatzplot` by typing

`dbclear all in collatzplot`

Breakpoints are automatically cleared when you:

- End the MATLAB session

- Clear the M-file using `clear name` or `clear all`
- Edit the file if the changes impact line numbering
- Edit the file while in debug mode (although this does not always clear breakpoints)

Correcting an M-File. To correct a problem in an M-file:

1 Quit debugging.

Do not make changes to an M-file while MATLAB is in debug mode. It could produce unexpected debugging results when you run the M-file.

2 Clear all the breakpoints in the file.

The breakpoints become unreliable once the M-file is edited. The breakpoints will produce unexpected debugging results when you run the file.

3 Make changes to the M-file.

4 Save the M-file.

5 Set breakpoints, if desired.

6 Run the M-file again to be sure it produces the expected results.

Completing the Example. To correct the problem in the example, do the following:

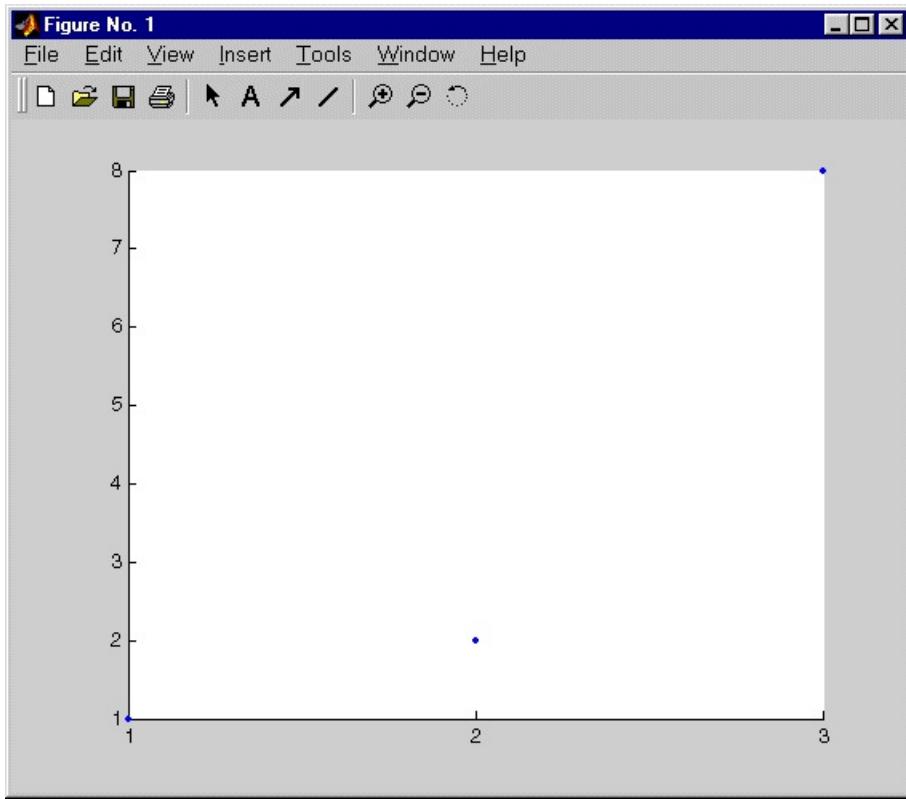
1 End the debugging session. One way to do this is to select **Exit Debug Mode** from the **Debug** menu.

2 Clear the breakpoints in `collatzplot.m`. One way to do this is by typing `dbclear all` in `collatzplot` in the Command Window.

3 In `collatzplot.m`, line 12, change the `stringplot_seqtoseq_length(m)` and save the file.

4 Run `collatzplot` for $n = 3$ by typing `collatzplot(3)` in the Command Window.

5 Verify the result. The figure shows that the length of the Collatz series is 1 when $n = 1$, 2 when $n = 2$, and 8 when $n = 3$, as expected.



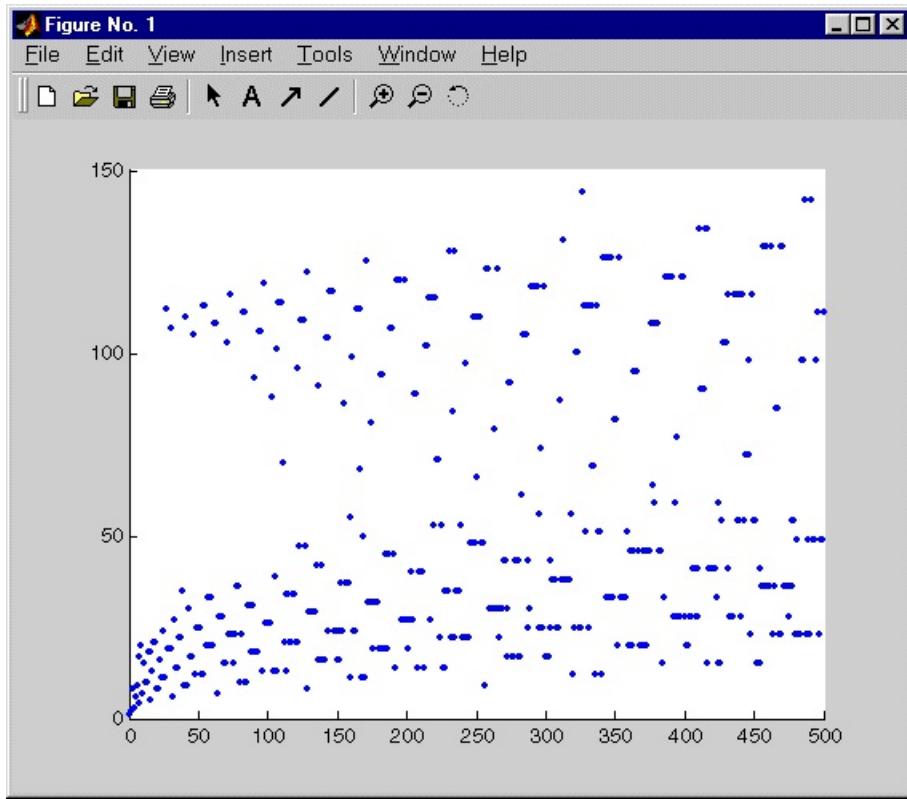
6 Test the function for a slightly larger value of n , such as 6, to be sure the results are still accurate. To make it easier to verify `collatzplot` for $n = 6$ as well as the results for `collatz`, add this line at the end of `collatz.m`

sequence which displays the series in the Command Window.
Then run `collatzplot` for $n = 6$ by typing `collatzplot(6)`

7 To make debugging easier, you ran `collatzplot` for a small value of n . Now that you know it works correctly, run `collatzplot` for a larger value to produce more interesting results. Before doing so, you might want to suppress output for the line you just added in step 6, line 18 of `collatz.m`, by adding a semicolon to the end of the line so it appears as

sequence; Then run `collatzplot(500)`

The following figure shows the lengths of the Collatz series for $n = 1$ through $n = 500$.

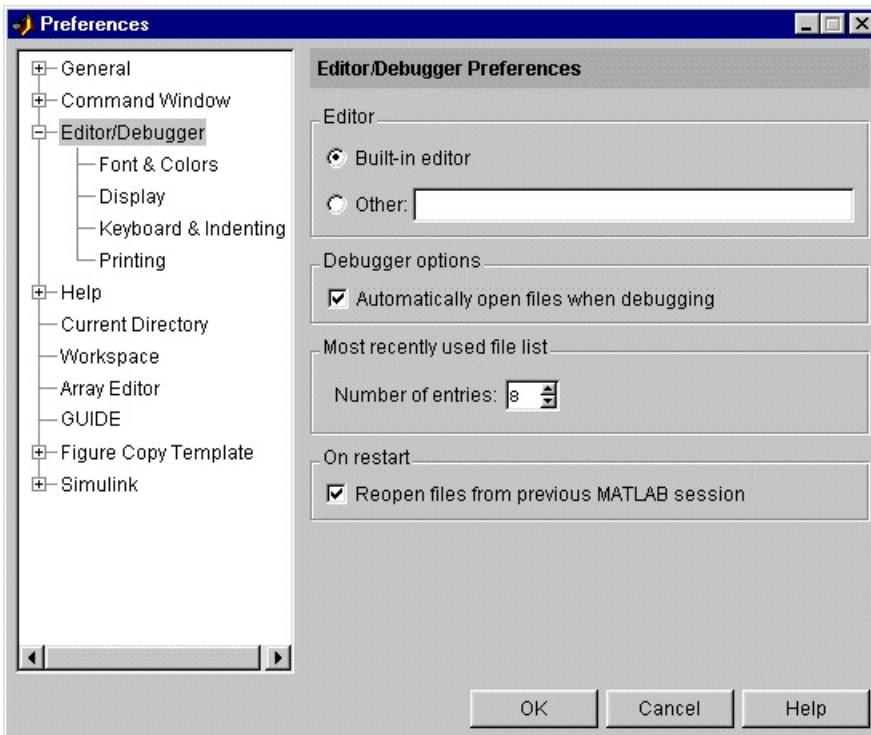


Preferences for the

Editor / Debugger

Using preferences, you can specify the default behavior for various aspects of the Editor/Debugger.

To set preferences for the Editor/Debugger, select **Preferences** from the **File** menu in the Editor/Debugger. The **Preferences** dialog box opens showing **Editor/Debugger Preferences**.



You can specify the

following Editor/Debugger preferences:

- “General Preferences for the Editor/Debugger” on page 7-36 (on the first panel, including the **Editor** preference)
- “Font & Colors Preferences for the Editor/Debugger” on page 7-37
- “Display Preferences for the Editor/Debugger” on page 7-38
- “Keyboard and Indenting Preferences for the Editor/Debugger” on page 7-40
- “Printing Preferences for the Editor/Debugger” on page 7-42

General Preferences for the Editor / Debugger

When you first access preferences for the Editor/Debugger, you can specify the general preferences described here.

Editor

By default, the **Built-in editor** option is selected, meaning that MATLAB uses its own Editor/Debugger.

To specify a text editor other than MATLAB’s Editor, such as Emacs or vi, to be used when you open an M-file from within MATLAB, select **Other**. In the **Other** field, type the path to the editor application you want to use.

For example, specify C:\Applications\Emacs.exe in the **Other** field, and then open a file using **Open** from the **File** menu the MATLAB desktop. The file opens in Emacs instead of in the MATLAB Editor/Debugger.

Debugger options

By default, the item **Automatically open files when debugging** is checked. The result is that when you run an M-file containing breakpoints, the MATLAB Editor/Debugger opens when it encounters a breakpoint.

If you use debugging functions, you might want to uncheck the item so that the Editor/Debugger does *not* open when a breakpoint is encountered.

Most recently used file list

Use this preference to specify the number of files that appear in the list of most recently used files in the **File** menu.

On restart

To start MATLAB and automatically open the files that were open when you last shut down MATLAB, check the item **Reopen files from previous MATLAB session**. If the item is unchecked and you close MATLAB when there are files open in the Editor/Debugger, the next time you start MATLAB, the Editor/Debugger is not opened upon startup.

Font & Colors Preferences for the Editor / Debugger

Use **Font & Colors** preferences to specify the font and colors used in files in the Editor/Debugger.

Font

Editor/Debugger font preferences specify the characteristics of the font used in files in the Editor/Debugger. Select **Use desktop font** if you want the font in the files to be the same as that specified under **General - Font & Colors**. If you want the font for Editor/Debugger files to be different, select **Use custom font** and specify the font characteristics:

- Type, for example, Lucida Console
- Style, for example, Plain
- Size in points, for example, 12 points

After you make a selection, the **Sample** area shows how the font will look. You can specify different font characteristics for printing files from the Editor – see “Printing Preferences for the Editor/Debugger” on page 7-42.

Colors

Specify the colors used in files in the Editor/Debugger:

- **Text color** – The color of nonspecial text; special text uses colors specified for

Syntax highlighting

- **Background color** – The color of background in the window.
- **Syntax highlighting** – The colors to use to highlight syntax. If checked, click

Set Colors to specify them. For a description of syntax highlighting, see “Syntax Highlighting” on page 3-5.

Files are tabbed within one window.

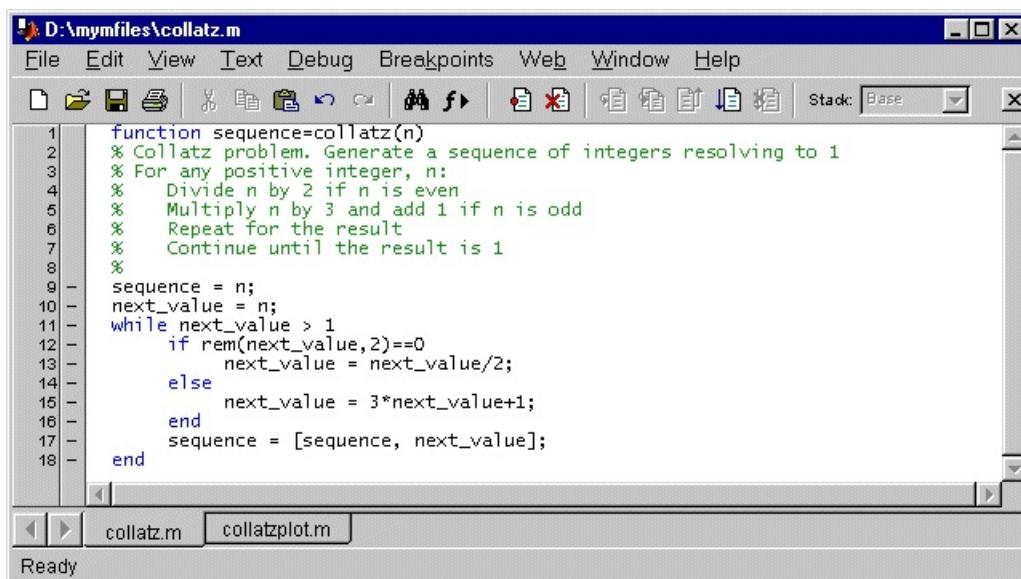
Display Preferences for the Editor / Debugger

Use **Display** preferences to specify how the Editor/Debugger window should look.

Opening files in editor

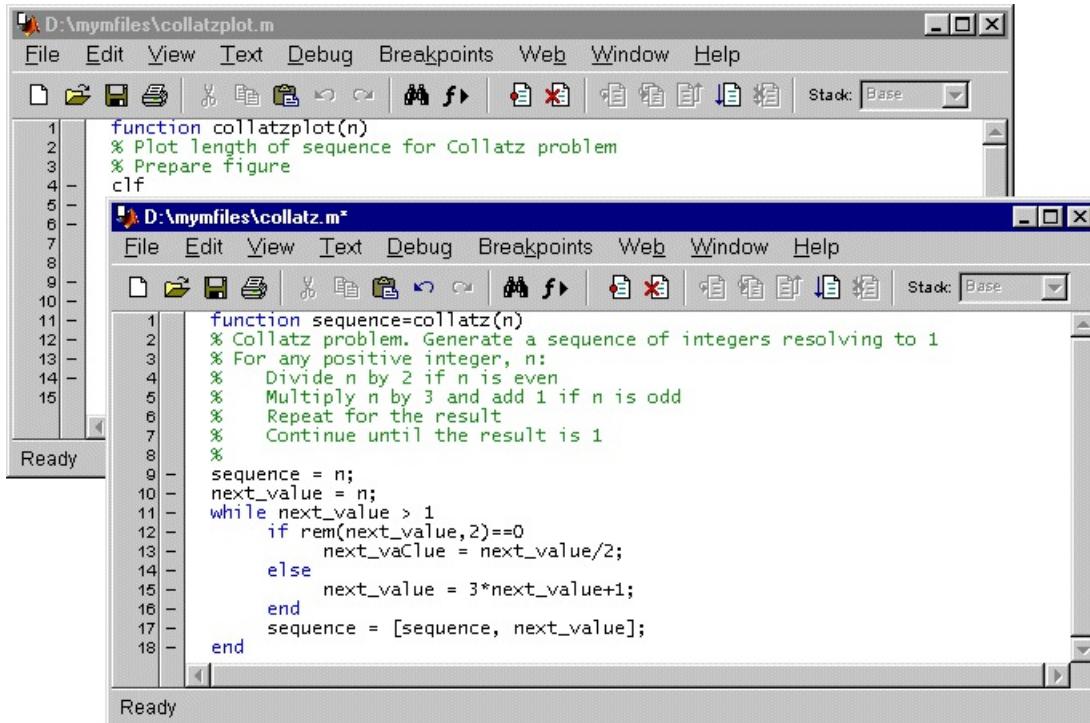
This preference controls how files are arranged when you open them in the Editor/Debugger. When you change this preference, it applies to files you open after making the change. Currently opened files are not rearranged to match the preference.

Check **Single window contains all files (tabbed style)** to have a single Editor/ Debugger window for all open files, as shown in the following illustration. Click the tab for a file to make it the current file.



Each file is its own window.

Check **Each file is displayed in its own window** to have a separate Editor/ Debugger window for each open file, as shown in the following illustration.



Display

Use display options to specify what is shown and what is hidden in the Editor/ Debugger.

Show toolbar. Check this item to display the toolbar. Uncheck it to hide the toolbar.

Show line numbers. Check this item to show line numbers. They appear along the left side of the window. When you uncheck this item, line numbers aren't shown.

Enable datatips in edit mode. Check this item to see datatips while in edit mode. Datatips are always enabled in debug mode.

Prompt

When you typeedit filename andfilename does not exist, MATLAB displays a prompt asking if you want to create a new file namedfilename.m. If you do not want to see this prompt, uncheck the preference **Show dialog prompt**

w hen editing files that do not exist . Then, the next time you typeedit filename, the file will be created without first prompting you.

If you do not generally use the edit command to create new files, then you might want to check the box to show the prompt. This alerts you to the possibility that you mistyped the filename you want to open.

Keyboard and Indenting Preferences for the Editor / Debugger

Use keyboard preferences to specify the key binding conventions MATLAB should follow. Use indenting preferences to specify how the Editor/Debugger indents lines.

Key bindings

Select **Windo ws** or **Emacs** depending on which convention you want the Editor/Debugger to follow for accelerators and shortcuts. The accelerators seen on the menus change after you change this option.

For example, when you select Windows key bindings, the shortcut to paste a selection is **Ctrl+V**. When you select Emacs key bindings, the shortcut to paste a selection is **Ctrl+Y**. You can see the accelerator on the **Edit** menu for the **P ast**e item.

M-file indenting for Enter key

Select the style of indenting you want the Editor/Debugger to use when you press the **Enter** key. Examples follow, illustrating the different styles.

- **No indent**– No lines are indented. Use this if you want lines to be aligned on the left or want to insert line indents manually.
- **Block indent** – Indents a line the same amount as the line above it.
- **Smart indent** – Automatically indents lines that start with keyword functions or that follow certain keyword functions. Smart indenting can help you to follow the code sequence.
The indenting style only applies to lines you enter after changing the preference; it does not affect the indenting of existing lines. To change the indenting for existing lines, use the **Text** menu entries for “Indenting” on page 7-9.

For any indenting style, you can manually insert tabs at the start of a line.

Example of No Indent Without Tabs.

```
sequence = n
next_value = n;
while next_value > 1
if rem(next_value,2)==0
next_value = next_value/2;
else
next_value = 3*next_value+1;
end
sequence = [sequence, next_value]
end
```

Example of No Indent with Tabs.

Created using **Noindent** preference. Did not manually insert any tabs.

```
sequence = n
next_value = n;
while next_value > 1
    if rem(next_value,2)==0
        next_value = next_value/2;
    else
        next_value = 3*next_value+1;
    end
    sequence = [sequence, next_value]
end
sequence = n
next_value = n;
while next_value > 1
    if rem(next_value,2)==0
        next_value = next_value/2;
    else
        next_value = 3*next_value+1;
    end
    sequence = [sequence, next_value]
end
```

Example of Block Indent.

Created using **No indent** preference. Created indentation by manually inserting a tab before each indented line.

Created using **Blockindent** preference. Inserted a tab before the if statement. Subsequent lines automatically indented on a tab.

Example of Smart Indent.

```
sequence = n
next_value = n;
while next_value > 1
    if rem(next_value,2)==0
        next_value = next_value/2;
    else
        next_value = 3*next_value+1;
    end
sequence = [sequence, next_value]
end
```

Indent

Created using **Smart indent** preference. Did not manually insert any tabs. Indented lines were automatically indented.

Indent size. Specify the indent size for smart indenting.

Emacs-style tab key smart indenting. This indenting convention is based on the style used by the Emacs editor. When you select the style, lines are not automatically indented. To indent a line(s) according to smart indenting practices, you can position the cursor in that line or select a group of lines and then press the **Tab** key.

Tab

Tab size. Specify the amount of space inserted when you press the **Tab** key. When you change the **Tab size**, it changes the tab space for existing lines in that file.

Tab key inserts spaces. Check this item if you want a series of spaces to be inserted when you press the **Tab** key. If the item is unchecked, a tab acts as one space whose length is determined by **Tab size**.

Printing Preferences for the Editor / Debugger

Use printing preferences to specify how printed M-files will look.

Syntax highlighting

This preference specifies how highlighted syntax is printed. The options are:

- **Print as black and white text**
- **Print as colored text**
- **Print as styled text**— Prints in black and white. Comments are italicized and keywords are bold.

Print options

- Check **Print header** to include a header on the printed page that lists the full pathname for the file, page numbers, and the date and time it is printed.
- Check **Print line numbers** to include line numbers when printing files.

Font

Use this preference to specify font characteristics for printed M-files. This is especially useful on Windows platforms because a given font size is displayed slightly smaller than the actual size, but is printed at the actual size. Therefore the font in the printed output might look larger than you expected.

To print documents using the same font characteristics as specified for the Editor/Debugger (see

“Font & Colors Preferences for the Editor/Debugger” on page 7-37), select **Use editor font**.

To print documents with different font characteristics than those specified for the Editor/Debugger, select **Use custom font**. After selecting this option, specify the font characteristics:

- Type, for example, Lucida Console
- Style, for example, Plain
- Size in points, for example, 12 points

8

Improving M-File Performance – the Profiler

What Is Profiling?	8-3
Using the Profiler	8-4
The profile Function.....	8-4
An Example Using the Profiler	8-6
Viewing Profiler Results	8-7
Viewing Profile Reports.....	8-7
Profile Plot	8-12
Saving Profile Reports.....	8-13

One way to improve the performance of your M-files is to profile them. MATLAB provides an M-file profiler that lets you see how much computation time each line of an M-file uses.

This section on profiling covers the following topics:

- “What Is Profiling?” on page 8-3
- “Using the Profiler” on page 8-4, including “The profile Function” and “An Example Using the Profiler”
- “Viewing Profiler Results” on page 8-7

There are numerous other techniques for improving the performance of your M-files. For a list of these types of functions, see Performance Tools and Techniques.

What Is Profiling?

Profiling is a way to measure where a program spends its time. Measuring is a much better method than guessing where the most execution time is spent. You probably deal with obvious speed issues at design time and can then discover unanticipated effects through measurement. One key to effective coding is to create an original implementation that is as simple as possible and then use a profiler to identify bottlenecks if speed is an issue. Premature optimization often increases code complexity unnecessarily without providing a real gain in performance.

Use the profiler to identify functions that are consuming the most time, then determine why you

are calling them and look for ways to minimize their use. It is often helpful to decide whether the number of times a particular function is called is reasonable. Because programs often have several layers, your code may not explicitly call the most expensive functions. Rather, functions within your code may be calling other time-consuming functions that can be several layers down in the code. In this case it's important to determine which of your functions are responsible for such calls.

The profiler often helps to uncover problems that you can solve by:

- Avoiding unnecessary computation, which can arise from oversight.
- Changing your algorithm to avoid costly functions.
- Avoiding recomputation by storing results for future use.

When you reach the point where most of the time is spent on calls to a small number of built-in functions, you have probably optimized the code as much as you can expect.

Using the Profiler

Use the `profile` function to generate and view statistics.

1 Start the profiler by typing `profile` on in the Command Window. Specify any options you want to use.

2 Execute your M-file.

The profiler counts how many seconds each line in the M-files use. The profiler works cumulatively, that is, adding to the count for each M-file you execute until you clear the statistics.

3 Use `profile report` to display the statistics gathered in an HTML-formatted report in your system's default Web browser.

The `profile` Function

Here is a summary of the main forms of `profile`. For details about these and other options, type `doc profile`.

Syntax Option Description `profile on` Starts the profiler, clearing previously recorded

statistics.

-detail level Specifies the level of function to be profiled.

-history Specifies that the exact sequence of function calls is to be recorded.

`profile off` Suspends the profiler. `profile report` Suspends the profiler, generates a profile report in HTML format, and displays the report in your system's default Web browser. `basename` Saves the report in the file `basename` in the current directory.

Using the Profiler

Syntax Option Description (Continued) `profile plot`

`profile resume`

`profile clear`

`s = profile('status')`

`stats = profile('info')`

Suspends the profiler and displays in a figure window a bar graph of the functions using the most

execution time.

Restarts the profiler without clearing previously recorded statistics.

Clears the statistics recorded by the profiler. Displays a structure containing the current profiler status.

Suspends the profiler and displays a structure containing profiler results.

Some people use `profile` simply to see the child functions; see also `depfun` for that purpose.

An Example Using the Profiler

This example demonstrates how to run the profiler.

1 To start the profiler, type in the Command Window
 `profile on -detail builtin -history`

The `-detail builtin` option instructs the profiler to gather statistics for built-in functions, in addition to the default M-functions, M-subfunctions, and MEX-functions.

The `-history` option instructs the profiler to track the exact sequence of entry and exit calls.

2 Execute an M-file. This example runs the Lotka-Volterra predator-prey population model. For more information about this model, type `lotkadem` to run a demonstration.

`[t,y] = ode23('lotka',[0 2],[20;20]);`

3 Generate the profile report and save the results to the file `lotkaprof`. profile report `lotkaprof`

This suspends the profiler, displays the profile report in your system's default Web browser, and saves the results. See "Viewing Profiler Results" on page 8-7 for more information.

4 Restart the profiler, without clearing the existing statistics. `profile resume`

The profiler is now ready to continue gathering statistics for any more M-files you run. It will add these new statistics to those generated in the previous steps.

5 Stop the profiler when you are finished gathering statistics. `profile off`

Viewing Profiler Results

There are two main ways to view the profiler results:

- "Viewing Profile Reports" on page 8-7
- "Profile Plot" on page 8-12

To save results, see "Saving Profile Reports" on page 8-13.

Viewing Profile Reports

To display profiler results, type

`profile report`

This suspends the profiler and produces three reports:

- "Summary Profile Report" on page 8-7

- "Function Details Profile Report" on page 8-10

- "Function Call History Profile Report" on page 8-11

The summary report appears in your system's default Web browser. Use the links at the top of

the report page to see the other reports.

Summary Profile Report

The summary report presents statistics about the overall execution and provides summary statistics for each function called. Values reported include:

- **Number of functions** – The numbers of built-in functions, M-functions, and M-subfunctions are reported.
- **Clock precision**– The precision of the profiler’s time measurement. When Time for a function is 0, it is actually a positive value, but smaller than the profiler can detect given the clock precision.
- **Time** columns – The total time spent in a function, including all child functions called. Because the time for a function includes time spent on child functions, the times do not add up to the **Total recorded time** and the percentages add up to more than 100%.
- **Self time** columns – The total time spent in a function, *not* including time for any child functions called. Adding the **Self time** values for all functions listed equals the **Total recorded time**. The **Self time** percentages for all functions add up to approximately 100%.

Note that the profiler itself uses some time, which is included in the profiler results.

Following is the summary report for the Lotka-Volterra model described in “An Example Using the Profiler” on page 8-6.

Viewsummaryreport(shown in this illustration).

View details for all functions.

View sequence of function calls.

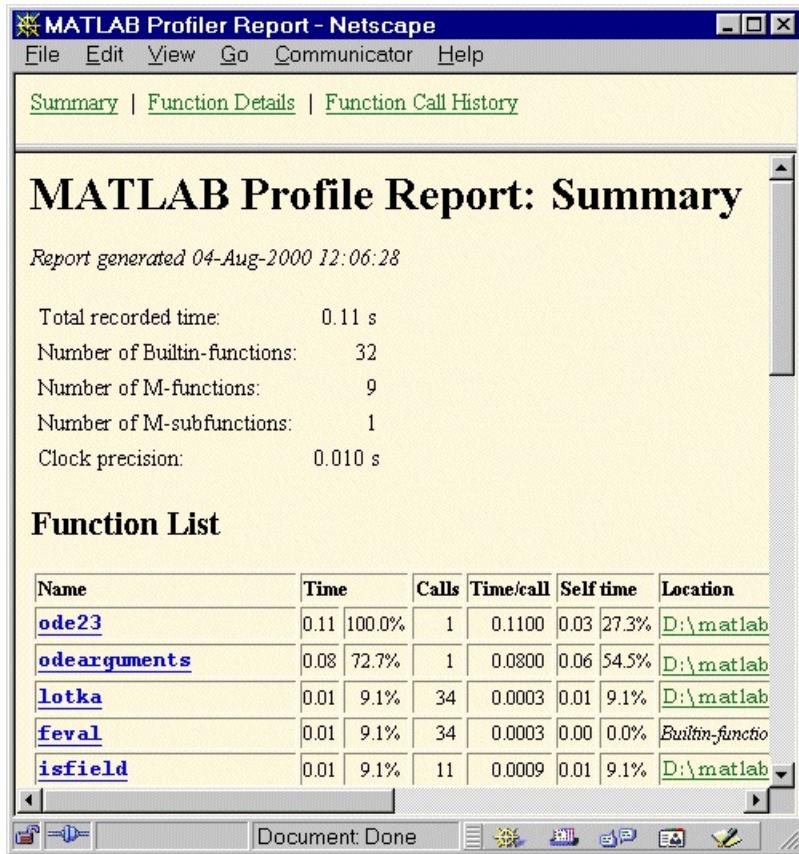
Total time profiler was recording.

Follow link to view details for each function.

Report includes built-in functions because

-detail builtin

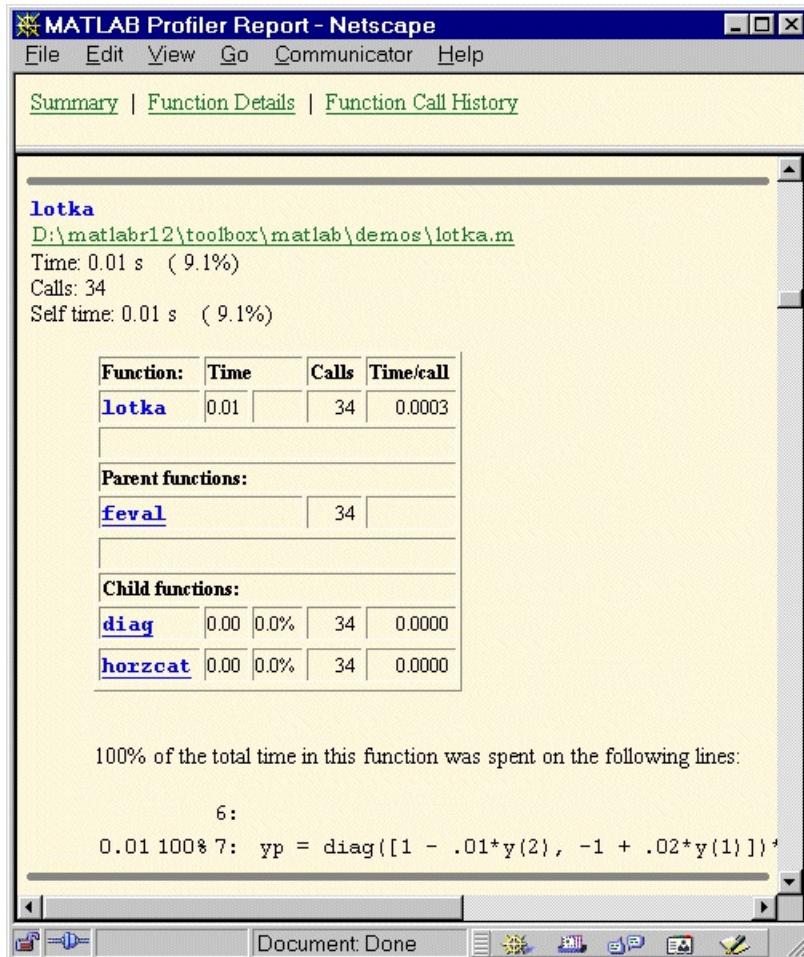
option was used.



Time per function includes time spent in child functions. Self time does NOT include time spent in child functions.

Function Details Profile Report

The function details report provides statistics for the parent and child functions of a function, and reports the line numbers on which the most time was spent. Following is detail report for the `lotka` function, which is one of the functions called in “An Example Using the Profiler” on page 8-6.



Time in seconds Percentage of the Line number
 function's time
 spent on that line

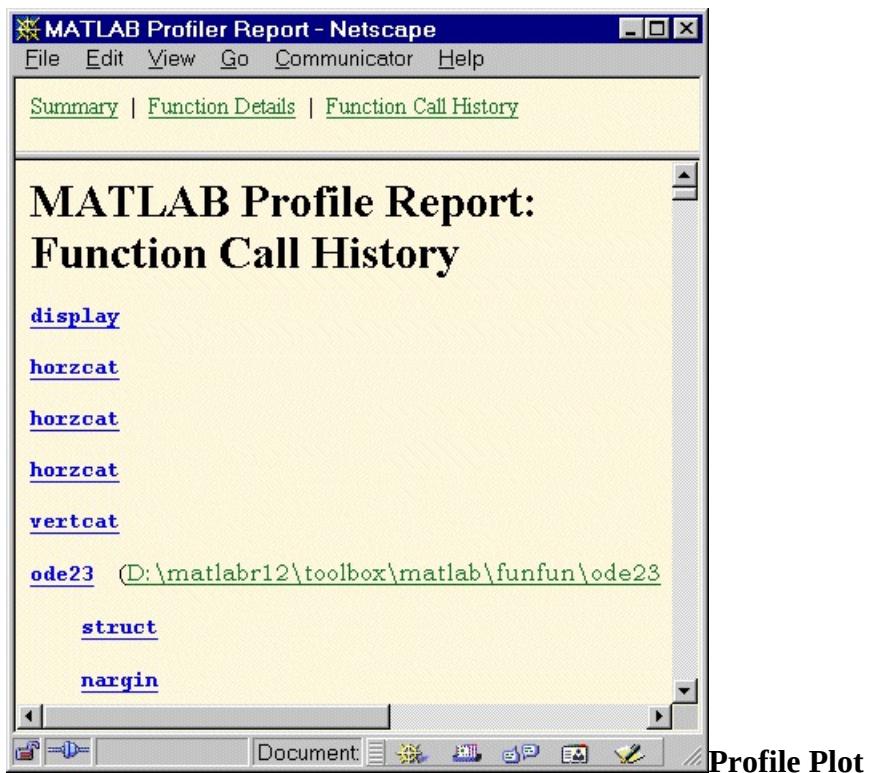
Function Call History Profile Report

The function call history displays the exact sequence of functions called. To view this report, you must have started the profiler using the -history option.

profile on -history

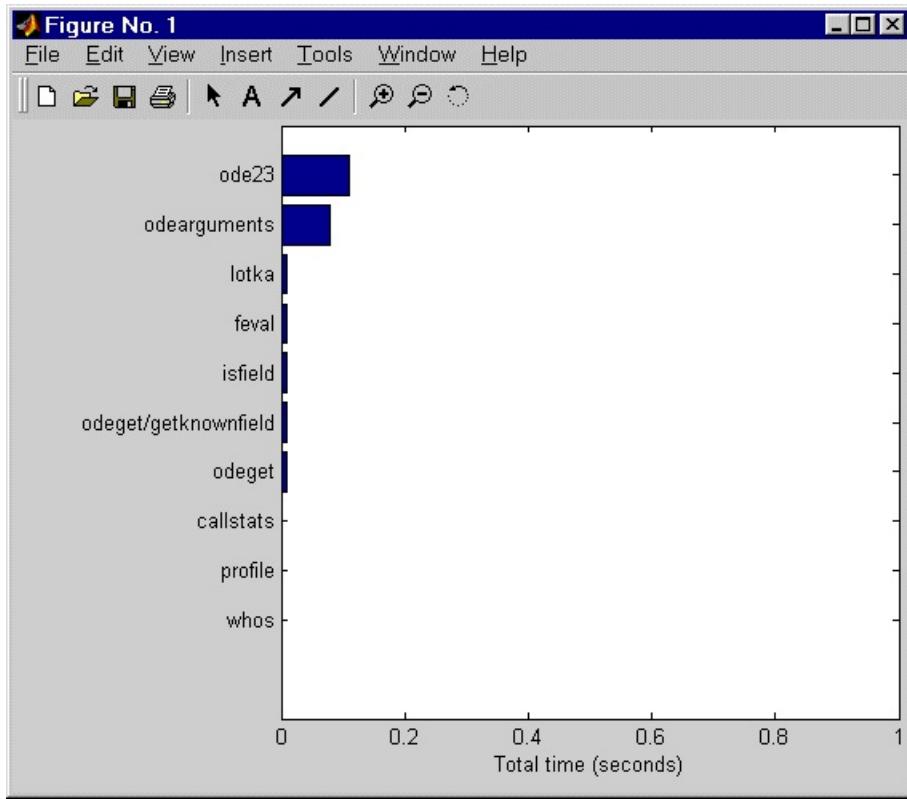
The profiler records up to 10,000 function entry and exit events. For more than 10,000 events, the profiler continues to record other profile statistics, but not the sequence of calls. Following is the history report generated from "An Example Using the Profiler" on page 8-6.

Exact sequence
 of calls



To view a bar graph for the functions using the most execution time, type profile plot

This suspends the profiler. The bar graph appears in a figure window. Following is the bar graph generated from “An Example Using the Profiler” on page 8-6.



Saving Profile Reports

When you generate the profile report, use the option to save it. For example, profile report basename

saves the profile report to the filebasename in the current directory. Later you can view the saved results using a Web browser.

Another way to save results is with the info = profile function, which displays a structure containing the profiler results. Save this structure so that later you can generate and view the profile report using profreport(info).

Example Using Structure of Profiler Results

The profiler results are stored in a structure that you can view or access. This example illustrates how you can view the results.

1 Run the profiler for code that computes the Lotka-Volterra predator-prey population model. profile on -detail builtin -history

```
[t,y] = ode23('lotka',[0 2],[20;20]);
```

2 To view the structure containing profiler results, type stats = profile('info') MATLAB returns

```
stats =
```

```
FunctionTable: [41x1 struct] FunctionHistory: [2x826 double] ClockPrecision: 0.0100
Name: 'MATLAB'
```

3 You can view and access the contents of the structure. For example, type stats.FunctionTable MATLAB displays theFunctionTable structure.

```
ans =  
41x1 struct array with fields:  
FunctionName  
FileName  
Type  
NumCalls  
TotalTime  
TotalRecursiveTime  
Children  
Parents  
ExecutedLines
```

4 To view the contents of an element in the FunctionTable structure, type, for example,
stats.FunctionTable(2)

MATLAB returns the second element in the structure.

```
ans =
```

```
FunctionName: 'horzcat'
```

```
FileName: ''
```

```
Type: 'Builtin-function'
```

```
NumCalls: 43
```

```
TotalTime: 0.0100
```

```
TotalRecursiveTime: 0.0100
```

```
Children: [0x1 struct] Parents: [2x1 struct] ExecutedLines: [0x3 double]
```

5 Save the results.

```
save profstats
```

6 In a later session, to generate the profiler report using the saved results, type

```
load profstats profreport(stats) MATLAB displays the profile report.
```

9

Interfacing with Source Control Systems

Process of Interfacing to an SCS	9-3
Vie w ing or Selecting the Source Control System.....	9-4
Function Alternative for Viewing the SCS.....	9-5
Setting Up the Source Control System	9-6
For SourceSafe Only – Mirroring MATLAB Hierarchy.....	9-6
For ClearCase on UNIX Only – Set a View and	
Check Out a Directory.....	9-6
Checking Files into the Source Control System	9-7
Function Alternative for Checking In Files.....	9-8
Checking Files Out of the SCS.....	9-9
Function Alternative for Checking Out Files.....	9-10

Undoing the Check-Out.....9-10

If you use a source control system (SCS) to manage your files, you can check M-files and Simulink and Stateflow files into and out of the source control system from within MATLAB, Simulink, and Stateflow.

MATLAB, Simulink, and Stateflow do not perform source control functions, but only provide an interface to your own source control system. This means, for example, that you can open a file in the MATLAB Editor and modify it without checking it out. However, the file will remain read-only so that you cannot accidentally overwrite the source control version of the file.

Four popular source control systems are supported, as well as a custom option:

- ClearCase from Rational Software
- PVCS from Merant
- RCS
- Visual SourceSafe from Microsoft
- Custom option – Allows you to build your own interface if you use a different source control system

Process of Interfacing to an SCS

Process of Interfacing to an SCS

You can interface to your source control system by using menus if you prefer a graphical user interface, or by using functions if you prefer to use the Command Window. There are some options you can perform using the functions that are not available with the menus – these are noted in the instructions.

Create M-files, Simulink files, or Stateflow files as you normally would and save the files. Then follow these steps to use MATLAB, Simulink, or Stateflow to interface with your source control system.

Steps Instructions

1 Select the source control system to use.

See “Viewing or Selecting the Source Control System” on page 9-4.

2 Set up your source control system to correctly include the files. This is only required for some source control systems.

“For SourceSafe Only – Mirroring MATLAB Hierarchy” on page 9-6

“For ClearCase on UNIX Only – Set a View and Check Out a Directory” on page 9-6

3 Check the files into the source control system. Note that for some source control systems, you must check out the files before you can check them in.

4 Next time you want to modify the files, open them in the MATLAB Editor, Simulink, or Stateflow, and check them out.

5 Undo a check-out if you want the files to remain checked in, without any of the changes you made since you checked them out.

“Checking Files into the Source Control System” on page 9-7

“Checking Files Out of the SCS” on page 9-9

“Undoing the Check-Out” on page 9-10

Viewing or Selecting the Source Control System

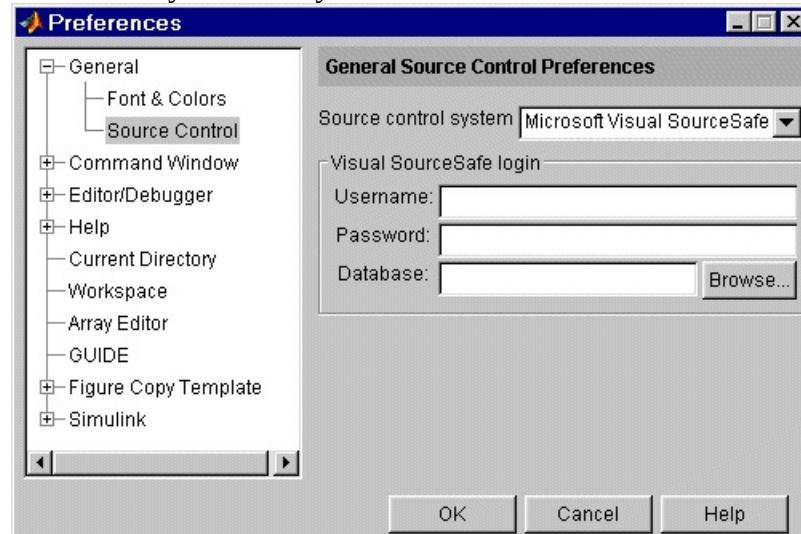
Specify the source control system for MATLAB to interface with:

1 From the MATLAB desktop, select **Preferences** from the **File** menu. You can also select this from the Simulink or Stateflow model or library windows.

The **Preferences** dialog box opens.

2 Click the + for **General** and then select **Source Control**.

The currently selected system is shown. The default selection isNone.



3 Select the system you want to use from the **Source control system** list. Viewing or Selecting the Source Control System

4 For Visual SourceSafe only, supply your **Username**, **P ass w ord**, and the **Database**.

- **P ass w ord** is not required if SourceSafe does not require it.

- **Database** is the full path for the srccafe.ini file associated with the

SourceSafe database you want to use. For example,

D:\Applications\Microsoft SourceSafe\srccafe.ini.

5 Click **OK**.

Function Alternative for Viewing the SCS

To view the currently selected system, typecmopts in the Command Window. MATLAB displays the current source control system.

Setting Up the Source Control System

For ClearCase on UNIX and for SourceSafe, set up your source control system as described here.

For SourceSafe Only– Mirroring MATLAB Hierarchy

If you use Visual SourceSafe, you must set up a project hierarchy in SourceSafe that mirrors the

hierarchy of your MATLAB, Simulink, and Stateflow files. For example, if you want to use the MATLAB Editor to interface to SourceSafe for the files in D:\matlabr12\mymfiles, configure a SourceSafe project as \$:\matlabr12\mymfiles.

In addition to setting up the hierarchy, you must supply your user name, password, and database in MATLAB general preferences. See “Viewing or Selecting the Source Control System” on page 9-4 for instructions.

For ClearCase on UNIX Only– Set a View and Check Out a Directory

If you use ClearCase on a UNIX platform, do the following using ClearCase:

1 Set a view.

2 Check out the directory in which you want to save files, check files into, or check files out of. You can now use the MATLAB, Simulink, or Stateflow interfaces to ClearCase to check files into and out of the directory you checked out in step 2. Checking Files into the Source Control System

Checking Files into the Source Control System

After creating or editing a file in the MATLAB Editor, Simulink, or Stateflow, save it, and then check in the file by following these steps:

1 From the MATLAB Editor, select **File -> Source Control -> Check In**. You can also select this from the Simulink or Stateflow model or library windows.

The **Check in** dialog box opens.



2 If you want to check in the file but keep it checked out so you can continue making changes, select **Keep checked out**.

3 If you have comments, type them in the **Comments** area.

Your comments will be submitted whether or not you select **Keep checked out**, yes.

4 Click **OK**.

The file is checked into the source control system. If you did not save the file before checking it in, it is automatically saved when it is checked in. If you did not keep the file checked out and you keep the file open, note that it is a read-only version.

Function Alternative for Checking In Files

Use `checkin` to check files into the source control system. The files can be open or closed when you use `checkin`. The `checkin` function has this form.

`checkin({'file1','filen'},'comments','string','option',... 'value')`

For `file`, use the complete path. You must supply the `comments` argument and a `comments` string with `checkin`.

Use the `option` argument to:

- Check in a file and keep it checked out – set thelock option to on.
- Check in a file even though it has not changed since the previous check in – set the force option to on.

The **comments** argument, and thelock andforce options apply to all files checked in.

After checking in the file, if you did not keep it checked out and have it open, note that it is a read-only version.

Example– Check in a File with Comments

To check in the fileclock.m with a commentAdjustment for Y2K, type
checkin('matlabr12\mymfiles\clock.m','comments','Adjustment ... for Y2K')

For other examples, see the reference page forcheckin.

Checking Files Out of the SCS

To check files out of the source control system using MATLAB, follow these steps:

1 Open the M-file, Simulink file, or Stateflow file you want to check out.

The file opens and the title bar indicates it is read-only.

2 From the MATLAB Editor, select **File -> Source Control -> Check Out**. You can also select this from the Simulink or Stateflow model or library windows.

The **Check-Out** dialog box opens.



3 To check out the version that was most recently checked in, select **Latest**. To check out a specific version of the file, select **Specific** and type the version number in the field

When you check out the latest version, you can use the Lock option. To prevent others from checking out the file while you have it checked out, select **Lock**. To check out a read-only version of the file, uncheck **Lock**.

4 Click OK.

The file is checked out from the source control system and is available to you for editing.

Function Alternative for Checking Out Files

Use checkout to check a file out of the source control system. You can check out multiple files at once and specify check-out options. The checkout function has this form.

`checkout({'file1','filen'},'option','value')`

For file, use the complete path.

Use the *option* argument to:

- Check out a read-only version of the file – set thelock option to off.
- Check out the file even if you already have it checked out – set theforce option to on.
- Check out a specific revision of the file – use therevisionoption, and assign the revision number

to the value argument.

The options apply to all files checked out. The file can be open or closed when you usecheckout.
Example— Check out a Specific Revision of a File

To check out the 1.1 revision of the file clock.m, type
checkout('matlab\mymfiles\clock.m', 'revision', '1.1')
For other examples, see the reference page forcheckout.

Undoing the Check- Out

You can undo the check-out for a file. The files remain checked in, without any of the changes you made since you checked them out. If you want to keep a local copy of your changes, use the **Save As** item from the **File** menu.

From the MATLAB Editor, select **File -> Source Control -> Undo Check Out**. You can also select this from the Simulink or Stateflow model or library windows.

Function Alternative for Undoing a Check-Out

Theundochekout function has this form.

undochekout({'file1','filen'})

Use the complete path forfile.

Example—Undo the Check-Out for Two Files. To undo the check-out for the files clock.m andcalendar.m, type

undochekout({'matlab\mymfiles\clock.m',
'matlab\mymfiles\calendar.m'})

10

Using Notebook

Notebook Basics.....	10-3
Creating an M-Book	10-3
Entering MATLAB Commands in an M-Book.....	10-6
Protecting the Integrity of Your Workspace.....	10-6
Ensuring Data Consistency.....	10-7
Defining MATLAB Commands as Input Cells	10-8
Defining Cell Groups.....	10-8
Defining Autoinit Input Cells.....	10-10
Defining Calc Zones	10-10
Converting an Input Cell to Text	10-11
Evaluating MATLAB Commands	10-12
Evaluating Cell Groups.....	10-13
Evaluating a Range of Input Cells	10-14
Evaluating a Calc Zone.....	10-14
Evaluating an Entire M-Book	10-15

Using a Loop to Evaluate Input Cells Repeatedly.....	10-15
Converting Output Cells to Text.....	10-16
Deleting Output Cells	10-17
Printing and Formatting an M-Book	10-18
Printing an M-Book.....	10-18
Modifying Styles in the M-Book Template.....	10-18
Choosing Loose or Compact Format.....	10-19
Controlling Numeric Output Format	10-20
Controlling Graphic Output.....	10-20
Configuring Notebook	10-24
Notebook Command Reference	10-26

Notebook allows you to access MATLAB's numeric computation and visualization software from within a word processing environment (Microsoft Word). Using Notebook, you can create a document, called an *M-book*, that contains text, MATLAB commands, and the output from MATLAB commands.

You can think of an M-book as a record of an interactive MATLAB session annotated with text or as a document embedded with live MATLAB commands and output. Notebook is useful for creating electronic or printed records of MATLAB sessions, class notes, textbooks or technical reports.

This section provides information about:

- “Notebook Basics” on page 10-3
- “Defining MATLAB Commands as Input Cells” on page 10-8
- “Evaluating MATLAB Commands” on page 10-12
- “Printing and Formatting an M-Book” on page 10-18
- “Configuring Notebook” on page 10-24
- “Notebook Command Reference” on page 10-26.

Notebook Basics

This section introduces basic Notebook capabilities, including:

- “Creating an M-Book” on page 10-3
- “Entering MATLAB Commands in an M-Book” on page 10-6
- “Protecting the Integrity of Your Workspace” on page 10-6
- “Ensuring Data Consistency” on page 10-7

Creating an M-Book

This section describes how to:

- Create an M-book from MATLAB
- Create an M-book while running Notebook
- Open an existing M-book
- Convert a Word document into an M-book

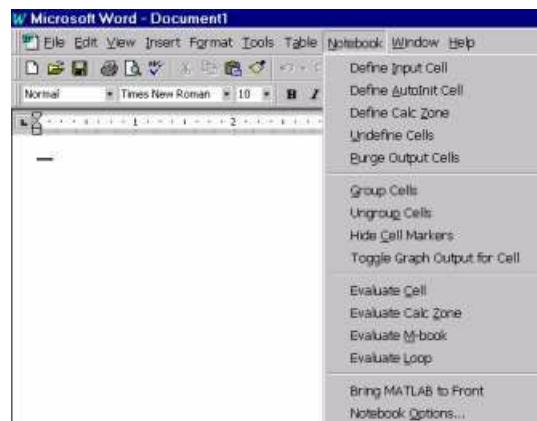
Creating an M-Book from MATLAB

To create a new M-book from within MATLAB, type notebook

at the prompt. If you are running Notebook for the first time, you may need to configure it. See “Configuring Notebook” on page 10-24 for more information.

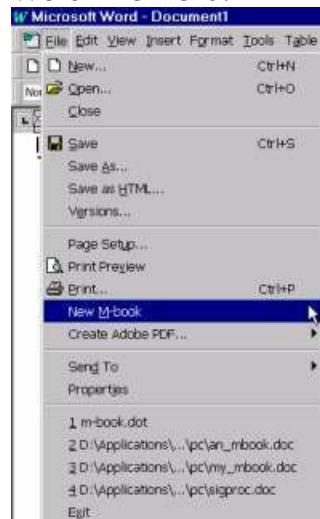
Notebook starts Microsoft Word on your system and creates a new M-book, called Document1. Notebook adds the **Notebook** menu to the Word menu bar. You use this menu, illustrated below, to access Notebook commands.

Note Notebook defines Microsoft Word macros that enable MATLAB to interpret the different types of cells that hold MATLAB commands and their output.



Creating an M-Book While Running Notebook

With Notebook running, you can also create a new M-book by selecting **New M-book** from the Word **File** menu.



Opening an Existing M-Book

You can also use the notebook command to open an existing M-book notebook filename

wherefilenameis the M-book you want to open, or you can simply double-click on an M-book file.

When you double-click on an M-book, Microsoft Word opens the M-book and starts MATLAB, if it is not already running. Notebook adds the **Notebook** menu to the Word menu bar.

Converting a Word Document to an M-Book

To convert a Word document to an M-book, follow these steps:

- 1** Create a new M-book.
- 2** From the **Insert** menu, select the **File** command.
- 3** Select the file you want to convert.
- 4** Click on the **OK** button.

Entering MATLAB Commands in an M-Book

Note A good way to learn how to use Notebook is to open the sample M-book, Readme.doc, and try out the various techniques described in this section. You can find this file in the \$MATLAB\notebook\pc directory, where \$MATLAB represents your installation directory.

You enter MATLAB commands in an M-book the same way you enter text in any other Word document. For example, you can enter the following text in a Word document. The example uses text in Courier Font but you can use any font.

Here is a sample M-book.

a = magic(3) To execute the MATLAB magic command in this document, you must:

- Define the command as an input cell
- Evaluate the input cell.

MATLAB displays the output of the command in the Word document in an output cell.

Protecting the Integrity of Your Workspace

When you work on more than one M-book in a single word processing session, note that:

- Each M-book uses the same “copy” of MATLAB.
- All M-books share the same workspace.

If you use the same variable names in more than one M-book, data used in one M-book can be affected by another M-book. You can protect the integrity of your workspace by specifying the clear command as the first autoinit cell in the M-book.

Ensuring Data Consistency

An M-book can be thought of as a sequential record of a MATLAB session. When executed in order, from the first MATLAB command to the last, the M-book accurately reflects the relationships among these commands.

If, however, you change an input cell or output cell as you refine your M-book, Notebook does not automatically recalculate input cells that depend on either the contents or the results of the changed cells. As a result, the M-book may contain inconsistent data.

When working on an M-book, you might find it useful to select the **Evaluate M-book** command periodically to ensure that your M-book data is consistent. You could also use calc zones to isolate related commands in a section of the M-book. You can then use the **Evaluate Calc Zone**

command to execute only those input cells contained in the calc zone.

Defining MATLAB Commands as Input Cells

To define a MATLAB command in a Word document as an input cell:

- 1 Type the command into the M-book as text. For example, This is a sample M-book.
`a = magic(3)`

- 2 Position the cursor anywhere in the command and choose the **Define Input Cell** command from the Notebook menu or press **Alt+D**. If the command is embedded in a line of text, use the mouse to select it. Notebook defines the MATLAB command as an input cell.

This is a sample M-book.

[`a = magic(3)`]

Note how Notebook changes the character font of the text in the input cell to a bold, dark green color and encloses it within *cell markers*. Cell markers are bold, gray brackets. They differ from the brackets used to enclose matrices by their size and weight. For information about changing these default formats, see “Modifying Styles in the M-Book Template” on page 10-18.

For information about defining other types of input cells, see:

- “Defining Cell Groups” on page 10-8
- “Defining Autoinit Input Cells” on page 10-10
- “Defining Calc Zones” on page 10-10
- “Converting an Input Cell to Text” on page 10-11

For information about evaluating the input cells you define, see “Evaluating MATLAB Commands” on page 10-12.

Defining Cell Groups

You can collect several input cells into a single input cell. This is called a *cell group*. Because all the output from a cell group appears in a single output cell that Notebook places immediately after the group, cell groups are useful when several MATLAB commands are needed to fully define a graphic. For example, if you define all the MATLAB commands that produce a graphic as a cell group and then evaluate the cell, Notebook generates a single graphic that includes all the graphic components defined in the commands. If instead you define all the MATLAB commands that generate the graphic as separate input cells, evaluating the cells generates multiple graphic output cells.

See “Evaluating Cell Groups” on page 10-13 for information about evaluating a cell group. For information about undefining a cell group, see “Ungroup Cells Command” on page 10-32.

Creating a Cell Group

To create a cell group:

- 1 Use the mouse to select the input cells that are to make up the group.

- 2 Select the **Group Cells** command from the Notebook menu or press **Alt+G**.

Notebook converts the selected cells into a cell group and replaces cell markers with a single pair that surrounds the group.

This is a sample cell group.

[date

a = magic(3)] Note the following:

- A cell group cannot contain text or output cells. If the selection includes output cells, Notebook deletes them.
- If the selection includes text, Notebook places the text after the cell group. However, if the text precedes the first input cell in the selection, Notebook leaves it where it is.
- If you select part or all of an output cell but not its input cell, Notebook includes the input cell in the cell group.

When you create a cell group, Notebook defines it as an input cell unless its first line is an autoinit cell, in which case Notebook defines the group as an autoinit cell.

Defining Autoinit Input Cells

You can use *autoinit cells* to specify MATLAB commands to be automatically evaluated each time an M-book is opened. This is a quick and easy way to initialize the workspace. *Autoinit cells* are simply input cells with the following additional characteristics:

- Notebook evaluates the autoinit cells when it opens the M-book.
 - Notebook displays the commands in autoinit cells using dark blue characters.
- Autoinit cells are otherwise identical to input cells.

Creating an Autoinit Cell

You can create an autoinit cell in two ways:

- Enter the MATLAB command as text, then convert the command to an autoinit cell by selecting the **Define AutoInit Cell** command from the Notebook menu.
- If you already entered the MATLAB command as an input cell, you can convert the input cell to an autoinit cell. Either select the input cell or position the cursor in the cell, then select the **Define AutoInit Cell** command from the Notebook menu.

See “Evaluating MATLAB Commands” on page 10-12 for information about evaluating autoinit cells.

Defining Calc Zones

You can partition an M-book into self-contained sections, called *calc zones*. A calc zone is a contiguous block of text, input cells, and output cells. Notebook inserts Microsoft Word section breaks before and after the section to define the calc zone. The section break indicators include bold, gray brackets to distinguish them from standard Word section breaks.

You can use calc zones to prepare problem sets, making each problem a separate calc zone that can be created and tested on its own. An M-book can contain any number of calc zones.

Note Using calc zones does not affect the scope of the variables in an M-book. Variables used in one calc zone are accessible to all calc zones.

Creating a Calc Zone

After you create the text and cells you want to include in the calc zone, you define the calc zone by following these steps:

1 Select the input cells and text to be included in the calc zone.

2 Choose the **Define Calc Zone** command from the Notebook menu.

Note You must select an input cell and its output cell in their entirety to include them in the calc zone.

See “Evaluating a Calc Zone” on page 10-14 for information about evaluating a calc zone.

Converting an Input Cell to Text

To convert an input cell (or an autoinit cell or a cell group) to text:

1 Select the input cell with the mouse or position the cursor in the input cell.

2 Select the **Undefine Cells** command from the Notebook menu or press **Alt+U**.

When Notebook converts the cell to text, it reformats the cell contents according to the Microsoft Word Normal style. For more information about M-book styles, see “Modifying Styles in the M-Book Template” on page 10-18. When you convert an input cell to text, Notebook also converts the corresponding output cell to text.

Evaluating MATLAB Commands

After you define a MATLAB command as an input cell, or as an autoinit cell, you can evaluate it in your M-book. Use the following steps to define and evaluate a MATLAB command:

1 Type the command into the M-book as text. For example, This is a sample M-book
`a = magic(3)`

2 Position the cursor anywhere in the command and choose the **Define Input Cell** command from the Notebook menu or press **Alt+D**. If the command is embedded in a line of text, use the mouse to select it. Notebook defines the MATLAB command as an input cell. For example,

This is a sample M-book

[a = magic(3)]

3 Choose the **Evaluate Cell** command from the Notebook menu or press **Ctrl+Enter**. You can specify the input cell to be evaluated by selecting it with the mouse, or placing the cursor in it.

Notebook evaluates the input cell and displays the results in a output cell immediately following the input cell. If there is already an output cell, Notebook replaces its contents, wherever it is in the M-book. For example,

This is a sample M-book.

[a = magic(3)]

**[a =
8 1 6
3 5 7
4 9 2]**

The text in the output cell is blue and is enclosed within cell markers. Cell markers are bold, gray

brackets. They differ from the brackets used to enclose matrices by their size and weight. Error messages appear in red. For information about changing these default formats, see “Modifying Styles in the M-Book Template” on page 10-18.

For more information about evaluating MATLAB commands in an M-book, see:

- “Evaluating Cell Groups” on page 10-13

- “Evaluating a Range of Input Cells” on page 10-14
- “Evaluating a Calc Zone” on page 10-14
- “Evaluating an Entire M-Book” on page 10-15
- “Using a Loop to Evaluate Input Cells Repeatedly” on page 10-15.
- “Converting Output Cells to Text” on page 10-16
- “Deleting Output Cells” on page 10-17

Evaluating Cell Groups

You evaluate a cell group the same way you evaluate an input cell (because a cell group is an input cell):

1 Position the cursor anywhere in the cell or in its output cell.

2 Choose the **Evaluate Cell** command from the Notebook menu or press **Ctrl+Enter**.

For information about creating a cell group, see “Defining Cell Groups” on page 10-8.

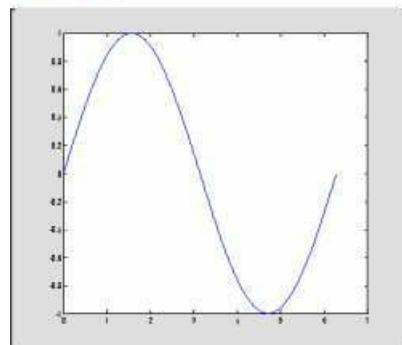
When MATLAB evaluates a cell group, the output for all commands in the group appears in a single output cell. By default, Notebook places the output cell immediately after the cell group the first time the cell group is evaluated. If you evaluate a cell group with an existing output cell, Notebook places the results in the output cell wherever it is located in the M-book.

Note Text or numeric output always comes first, regardless of the order of the commands in the group.

The illustration shows a cell group and the figure created when you evaluate the cell group.

This is a sample M-book with a cell group.

```
t = 0:pi/100:2*pi;
y = sin(t);
plot(t,y)
```



Evaluating a Range of Input Cells

To evaluate more than one MATLAB command contained in different but contiguous input cells:

1 Select the range of cells that includes the input cells you want to evaluate. You can include text that surrounds input cells in your selection.

2 Choose the **Evaluate Cell** command from the Notebook menu or press **Ctrl+Enter**.

Notebook evaluates each input cell in the selection, inserting new output cells or replacing

existing ones.

Evaluating a Calc Zone

To evaluate a calc zone:

1 Position the cursor anywhere in the calc zone.

2 Select the **Evaluate Calc Zone** command from the Notebook menu or press **Alt+Enter**.

For information about creating a calc zone, see “Defining Calc Zones” on page 10-10.

By default, Notebook places the output cell immediately after the calc zone the first time the calc zone is evaluated. If you evaluate a calc zone with an existing output cell, Notebook places the results in the output cell wherever it is located in the M-book.

Evaluating an Entire M-Book

To evaluate the entire M-book, either select the **Evaluate M-book** command or press **Alt+R**.

Notebook begins at the top of the M-book regardless of the cursor position and evaluates each input cell in the M-book. As it evaluates the M-book, Notebook inserts new output cells or replaces existing output cells.

Controlling Execution of Multiple Commands

When you evaluate an entire M-book, and an error occurs, evaluation continues. If you want to stop evaluation if an error occurs, follow this procedure:

1 Choose **Notebook Options** from the Notebook menu.

2 Click the **Stop evaluating on error** check box.

Using a Loop to Evaluate Input Cells Repeatedly

To evaluate a sequence of MATLAB commands repeatedly:

1 Use the mouse to select the input cells, including any text or output cells located between them.

2 Choose the **Evaluate Loop** command or press **Alt+L**. Notebook displays the **Evaluate Loop** dialog box.



3 Enter the number of times you want MATLAB to evaluate the selected commands in the **Stop After** field, then click on the **Start** button. The label on the button changes to **Stop**. Notebook begins evaluating the commands and indicates the number of completed iterations in the **Loop Count** field.

You can increase or decrease the delay at the end of each iteration by clicking on the **Slow er** or **Faster** button. Slower increases the delay. Faster decreases the delay.

To suspend evaluation of the commands, click on the **Pause** button. The label on the button changes to **Resume**. Click on the **Resume** button to continue evaluation.

To stop processing the commands, click on the **Stop** button. To close the **Evaluate Loop** dialog box, click on the **Close** button.

Converting Output Cells to Text

You can convert an output cell to text using the **Undefine Cells** command. If the output is numeric or textual, Notebook removes the cell markers and converts the cell contents to text according to the Microsoft Word Normal style. If the output is graphical, Notebook removes the cell markers and dissociates the graphic from its input cell, but does not alter its contents.

Note Undefining an output cell does not affect the associated input cell.

To undefine an output cell:

1 Select the output cell you want to undefine.

2 Choose the **Undefine Cells** command from the Notebook menu or press **Alt+U**.

Deleting Output Cells

To delete output cells:

1 Select an output cell, using the mouse, or place the cursor in the output cell.

2 Choose the **Purge Output Cells** command from the Notebook menu or press **Alt+P**.

If you select a range of input cells, each with output cells, Notebook deletes all the associated output cells.

Printing and Formatting an M-Book

This section describes:

- “Printing an M-Book” on page 10-18
- “Modifying Styles in the M-Book Template” on page 10-18
- “Choosing Loose or Compact Format” on page 10-19
- “Controlling Numeric Output Format” on page 10-20
- “Controlling Graphic Output” on page 10-20

Printing an M-Book

You can print all or part of an M-book by selecting the **Print** command from the **File** menu. Word follows these rules when printing M-book cells and graphics:

- Cell markers are not printed.
- Input cells, autoinit cells, and output cells (including error messages) are printed according to their defined styles. If you prefer to print these cells using black type instead of colors or shades of gray, you can modify the styles.

Modifying Styles in the M-Book Template

You can control the appearance of the text in your M-book by modifying the predefined styles stored in the M-book template. These styles control the appearance of text and cells. By default, M-books use the Word Normal style for all other text.

For example, if you print an M-book on a color printer, input cells appear dark green, output and autoinit cells appear dark blue, and error messages appear red. If you print the M-book on a grayscale printer, these cells appear as shades of gray. To print these cells using black type, you

need to modify the color of the Input, Output, AutoInit, and Error styles in the M-book template.

The table below describes the default styles used by Notebook. If you modify styles, you can use the information in the tables below to help you return the styles to their original settings. For general information about using styles in Word documents, see the Word documentation.

Style	Font	Size	Weight	Color
Normal	Times New Roman	10 points	Black	AutoInit
Courier	New	10 points	Bold	Dark Blue
Error	Courier	New	10 points	Bold Red
Input	Courier	New	10 points	Dark Green
Output	Courier	New	10 points	Blue

When you change a style, Word applies the change to all characters in the M-book that use that style and gives you the option to change the template. Be cautious about making changes to the template. If you choose to apply the changes to the template, you will affect all new M-books you create using the template. See the Word documentation for more information.

Choosing Loose or Compact Form at

You can specify whether a blank line appears between the input and output cells by selecting Loose or Compact format.

1 Click on the Notebook menu and select **Notebook Options**.

2 In the **Notebook Options** dialog box, select either the Loose or Compact check box. Loose format adds an empty line. Compact format does not.



3 Click **OK**.

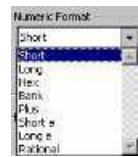
Note Changes you make using the **Notebook Options** dialog box take effect for output generated *after* you click on the **OK** button. To affect existing input or output cells, you must re-evaluate the cells.

Controlling Numeric Output Form at

To change how Notebook displays numeric output:

1 Click on the Notebook menu and select **Notebook Options**.

2 In the **Notebook Options** dialog box, click on the **Numeric Format** menu to view a list of available formats. These settings correspond to the choices available with the MATLABformat command. The figure below shows the available numeric formats.



3 Click **OK**.

Note Changes you make using the **Notebook Options** dialog box take effect for output generated *after* you click on the **OK** button. To affect existing input or output cells, you must re-evaluate the cells.

Controlling Graphic Output

This section describes how to control several aspects of the graphic output produced by MATLAB commands in an M-book, including:

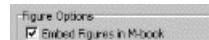
- “Embedding Graphic Output in the M-Book” on page 10-21
- “Suppressing Graphic Output for Individual Input Cells” on page 10-21
- “Sizing Graphic Output” on page 10-22
- “Cropping Graphic Output” on page 10-22
- “Adding White Space Around Graphic Output” on page 10-22
- “Specifying Color Mode” on page 10-23

Embedding Graphic Output in the M-Book

By default, graphic output is embedded in an M-book. To display graphic output in a separate figure window:

1 Click on the Notebook menu and choose **Notebook Options**.

2 In the **Notebook Options** dialog box, deselect the **Embed Figures in M-book** check box.

 **3** Click **OK**.

Note Embedded figures do not include Handle Graphics® objects generated by the uicontrol and uimenu functions.

Notebook determines whether to embed a figure in the M-book by examining the value of the figure object's Visible property. If the value of the property is off, Notebook embeds the figure. If the value of this property is on, all graphic output is directed to the current figure window.

Suppressing Graphic Output for Individual Input Cells

If an input or autoinit cell generates figure output that you want to suppress:

1 Place the cursor in the input cell.

2 Click on the Notebook menu and choose the **Toggle Graph Output for Cell** command.

Notebook suppresses graphic output from the cell, inserting the string (no graph) after the input cell.

To allow graphic output for a cell, repeat the same procedure. Notebook allows graphic output from the cell and removes the (no graph) marker.

Note The **Toggle Graph Output for Cell** command overrides the **Embed Figures in M-book** option, if that option is set.

Sizing Graphic Output

To set the default size of embedded graphics in an M-book:

1 Click on the Notebook menu and choose **Notebook Options**.

2 In the **Notebook Options** dialog box, use the **Height** and **Width** fields to set the size of graphics generated by the M-book.

3 Click **OK**.

Note Changes you make using the **Notebook Options** dialog box take effect for graphic output generated *after* you click on the **OK** button. To affect existing input or output cells, you must re-evaluate the cells.

You change the size of an existing embedded figure by selecting the figure, clicking the left mouse button anywhere in the figure, and dragging the resize handles of the figure. If you resize an embedded figure using its handles and then regenerate the figure, its size reverts to its original size.

Cropping Graphic Output

To crop an embedded figure to cut off areas you do not want to show:

- 1 Select the graphic, by clicking the left mouse button anywhere in the figure.
- 2 Hold down the **Shift** key.
- 3 Drag a sizing handle toward the center of the graphic.

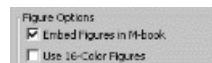
Adding White Space Around Graphic Output

You can add white space around an embedded figure by moving the boundaries of a graphic outward. Select the graphic, then hold down the **Shift** key and drag a sizing handle away from the graphic.

Specifying Color Mode

If you print graphic output that includes surfaces or patches, the output uses 16-color mode by default. To use 256-color mode:

- 1 Click on the Notebook menu and choose **Notebook Options**.
- 2 Deselect the **Use 16-Color Figures** check box in the **Notebook Options** dialog box.



- 3 Click **OK**.

Note Changes you make using the **Notebook Options** dialog box take effect for graphic output generated *after* you click on the **OK** button. To affect existing input or output cells, you must re-evaluate the cells.

Configuring Notebook

After you install Notebook, you must configure it. (Notebook is installed as part of the MATLAB installation process. See the *MATLAB Installation Guide for PC* for more information.)

To configure Notebook, type
notebook -setup

in the MATLAB command window. Notebook prompts you to specify which version of Microsoft Word you are using.

Choose your version of Microsoft Word:

- [1] Microsoft Word for Windows 95 (Version 7.0)
- [2] Microsoft Word 97
- [3] Microsoft Word 2000
- [4] Exit, making no changes

Microsoft Word Version: 2

Next Notebook prompts you to specify the location of the Microsoft Word executable (winword.exe) on your system.

You will be presented with a dialog box. Please use it to select your copy of the Microsoft Word 97 executable (winword.exe). Press any key to continue...

Use this dialog box to specify the location of your Microsoft Word program.



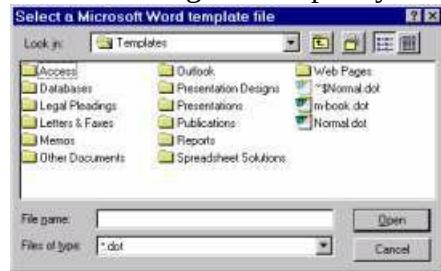
Configuring Notebook

When you click **Open**, Notebook prompts you to specify the name of a Microsoft Word template file.

You will be presented with a dialog box. Please use it to select a Microsoft Word template (.dot) file in one of your Microsoft Word template directories. We suggest that you specify your normal.dot file.

Press any key to continue...

Use this dialog box to specify the name of your Microsoft Word template file.



When configuration is completed, Notebook outputs this message. Notebook setup is complete.

Notebook Command Reference

This section provides reference information about each of the Notebook commands, listed alphabetically. To use these commands, select them from the Notebook menu.

- “Bring MATLAB to Front Command” on page 10-26
- “Define Autoinit Cell Command” on page 10-26
- “Define Calc Zone Command” on page 10-27
- “Define Input Cell Command” on page 10-27
- “Evaluate Calc Zone Command” on page 10-28
- “Evaluate Cell Command” on page 10-28
- “Evaluate Loop Command” on page 10-29
- “Evaluate M-Book Command” on page 10-30
- “Group Cells Command” on page 10-30
- “Hide Cell Markers Command” on page 10-31
- “Notebook Options Command” on page 10-31
- “Purge Output Cells Command” on page 10-31
- “Toggle Graph Output for Cell Command” on page 10-31
- “Undefine Cells Command” on page 10-32
- “Ungroup Cells Command” on page 10-32

Bring M ATLAB to Front Comm and

The **Bring MATLAB to Front** command brings the MATLAB command window to the foreground.

Define Autoinit Cell Comm and

The **Define AutoInit Cell** command creates an autoinit cell by converting the current paragraph, selected text, or input cell. An autoinit cell is an input cell that is automatically evaluated whenever you open an M-book.

Action

If you select this command while the cursor is in a paragraph of text, Notebook converts the entire paragraph to an autoinit cell. If you select this command while text is selected, Notebook converts the text to an autoinit cell. If you select this command while the cursor is in an input cell, Notebook converts the input cell to an autoinit cell.

Format

Notebook formats the autoinit cell using the AutoInit style, defined as bold, dark blue, 10-point Courier New.

See Also

For more information about autoinit cells, see “Defining Autoinit Input Cells” on page 10-10.

Define Calc Zone Comm and

The **Define Calc Zone** command defines the selected text, input cells, and output cells as a calc zone. A calc zone is a contiguous block of related text, input cells, and output cells that describes a specific operation or problem.

Action

Notebook defines a calc zone as a Word document section, placing section breaks before and after the calc zone. However, Word does not display section breaks at the beginning or end of a document.

See Also

For information about evaluating calc zones, see “Evaluating a Calc Zone” on page 10-14. For more information about document sections, see the Microsoft Word documentation.

Define Input Cell Comm and

The **Define Input Cell** command creates an input cell by converting the current paragraph, selected text, or autoinit cell. An input cell contains a MATLAB command.

Action

If you select this command while the cursor is in a paragraph of text, Notebook converts the entire paragraph to an input cell. If you select this command while text is selected, Notebook converts the text to an input cell.

If you select this command while the cursor is in an autoinit cell, Notebook converts the autoinit cell to an input cell.

Format

Notebook encloses the text in cell markers and formats the cell using the Input style, defined as bold, dark green, 10-point Courier New.

See Also

For more information about creating input cells, see “Defining MATLAB Commands as Input Cells” on page 10-8. For information about evaluating input cells, see “Evaluating MATLAB Commands” on page 10-12.

Evaluate Calc Zone Comm and

The **Evaluate Calc Zone** command sends the input cells in the current calc zone to MATLAB to be evaluated. A calc zone is a contiguous block of related text, input cells, and output cells that describes a specific operation or problem.

The current calc zone is the Word section that contains the cursor.

Action

As Notebook evaluates each input cell, it generates an output cell. When you evaluate an input cell for which there is no output cell, Notebook places the output cell immediately after the input cell that generated it. If you evaluate an input cell for which there is an output cell, Notebook replaces the results in the output cell wherever it is in the M-book.

See Also

For more information, see “Evaluating a Calc Zone” on page 10-14.

Evaluate Cell Comm and

The **Evaluate Cell** command sends the current input cell or cell group to MATLAB to be evaluated. An input cell contains a MATLAB command. A cell group is a single, multiline input cell that contains more than one MATLAB command. Notebook displays the output or an error message in an output cell.

Action

If you evaluate an input cell for which there is no output cell, Notebook places the output cell immediately after the input cell that generated it. If you evaluate an input cell for which there is an output cell, Notebook replaces the results in the output cell wherever it is in the M-book. If you evaluate a cell group, all output for the cell appears in a single output cell.

An input cell or cell group is the current input cell or cell group if:

- The cursor is in the input cell or cell group.
- The cursor is at the end of the line that contains the closing cell marker for the input cell or cell group.
- The cursor is in the output cell for the input cell or cell group.
- The input cell or cell group is selected.

Note Evaluating a cell that involves a lengthy operation may cause a time-out. If this happens, Word displays a time-out message and asks whether you want to continue waiting for a response or terminate the request. If you choose to continue, Word resets the time-out value and continues waiting for a response. Word sets the time-out value; you cannot change it.

See Also

For more information, see “Evaluating MATLAB Commands” on page 10-12. For information about evaluating the entire M-book, see “Evaluating an Entire M-Book” on page 10-15.

Evaluate Loop Comm and

The **Evaluate Loop** command evaluates the selected input cells repeatedly. For more information, see “Using a Loop to Evaluate Input Cells Repeatedly” on page 10-15.

Evaluate M-Book Comm and

The **Evaluate M-book** command evaluates the entire M-book, sending all input cells to MATLAB to be evaluated. Notebook begins at the top of the M-book regardless of the cursor position.

Action

As Notebook evaluates each input cell, it generates an output cell. When you evaluate an input cell for which there is no output cell, Notebook places the output cell immediately after the input cell that generated it. If you evaluate an input cell for which there is an output cell, Notebook replaces the results in the output cell wherever it is in the M-book.

See Also

For more information, see “Evaluating an Entire M-Book” on page 10-15.

Group Cells Comm and

The **Group Cells** command converts the input cells in the selection into a single multiline input cell called a cell group. You evaluate a cell group using the **Evaluate Cell** command. When you evaluate a cell group, all of its output follows the group and appears in a single output cell.

Action

If you include text in the selection, Notebook moves it after the cell group. However, if text precedes the first input cell in the group, the text will remain before the group.

If you include output cells in the selection, Notebook deletes them. If you select all or part of an output cell before selecting this command, Notebook includes its input cell in the cell group.

If the first line in the cell group is an autoinit cell, the entire group acts as a sequence of autoinit cells. Otherwise, the group acts as a sequence of input cells. You can convert an entire cell group to an autoinit cell with the **Define AutoInit Cell** command.

See Also

For more information, see “Defining Cell Groups” on page 10-8. For information about converting a cell group to individual input cells, see the description of the “Ungroup Cells Command” on page 10-32.

Hide Cell Markers Comm and

The **Hide Cell Markers** command hides cell markers in the M-book. When you select this command, it changes to **Show Cell Markers**.

Note Notebook does not print cell markers whether you choose to hide them or show on the screen.

Notebook Options Comm and

The **Notebook Options** command allows you to examine and modify display options for numeric and graphic output.

See Also

See “Printing and Formatting an M-Book” on page 10-18 for more information.

Purge Output Cells Comm and

The **Purge Output Cells** command deletes all output cells from the current selection.

See Also

For more information, see “Deleting Output Cells” on page 10-17.

Toggle Graph Output for Cell Comm and

The **Toggle Graph Output for Cell** command suppresses or allows graphic output from an input cell.

If an input or autoinit cell generates figure output that you want to suppress, place the cursor in the input cell and choose this command. The string(no graph) will be placed after the input cell to indicate that graph output for that cell will be suppressed.

To allow graphic output for that cell, place the cursor inside the input cell and choose **Toggle Graph Output for Cell** again. The (no graph) marker will be removed. This command overrides the Embed Graphic Output in the M-Book option, if that option is set in the **Notebook Options** dialog.

See Also

See “Embedding Graphic Output in the M-Book” on page 10-21 and “Suppressing Graphic Output for Individual Input Cells” on page 10-21 for more information.

Undefine Cells Comm and

The **Undefine Cells** command converts the selected cells to text. If no cells are selected but the cursor is in a cell, Notebook undefines that cell. Notebook removes the cell markers and reformats the cell according to the Normal style.

If you undefine an input cell, Notebook automatically undefines its output cell. However, if you undefine an output cell, Notebook does not undefine its input cell. If you undefine an output cell containing an embedded graphic, the graphic remains in the M-book but is no longer associated with an input cell.

See Also

For information about the Normal style, see “Modifying Styles in the M-Book Template” on page 10-18. For information about deleting output cells, see the description of the “Purge Output Cells Command” on page 10-31.

Ungroup Cells Comm and

The **Ungroup Cells** command converts the current cell group into a sequence of individual input cells or autoinit cells. If the cell group is an input cell, Notebook converts the cell group to input cells. If the cell group is an autoinit cell, Notebook converts the cell group to autoinit cells. Notebook deletes the output cell for the cell group.

A cell group is the current cell group if:

- The cursor is in the cell group.
- The cursor is at the end of a line that contains the closing cell marker for the cell group.
- The cursor is in the output cell for the cell group.
- The cell group is selected.

See Also

For information about creating cell groups, see the description of the “Defining Cell Groups” on page 10-8.

M athematics

MATLAB provides many functions for performing mathematical operations and analyzing data. The following list summarizes the contents of this collection:

- Matrices and Linear Algebra — describes matrix creation and matrix operations that are directly supported by MATLAB. Topics covered include matrix arithmetic, linear equations, eigenvalues, singular values, and matrix factorizations.
- Polynomials and Interpolation — describes functions for standard polynomial operations such as polynomial roots, evaluation, and differentiation. Additional topics covered include curve fitting and partial fraction expansion.
- Data Analysis and Statistics — describes how to organize arrays for data analysis, how to use simple descriptive statistics functions, and how to perform data pre-processing tasks in MATLAB. Additional topics covered include regression, curve fitting, data filtering, and fast Fourier transforms (FFTs).
- Function Functions — describes MATLAB functions that work with mathematical functions instead of numeric arrays. These function functions include plotting, optimization, zero finding, and numerical integration (quadrature).
- Differential Equations — describes the solution, in MATLAB, of initial value problems for ordinary differential equations (ODEs) and

differential-algebraic equations (DAEs), and the solution of boundary value problems for ODEs. It also describes the solution of initial-boundary value problems for systems of parabolic and elliptic partial differential equations (PDEs). Topics covered include representing problems in MATLAB, solver syntax, and using integration parameters.

- Sparse Matrices — describes how to create sparse matrices in MATLAB, and how to use them in both specialized and general mathematical operations.

Matrices and Linear Algebra

Function Summary	11-3
Matrices in MATLAB.....	11-5
Creation	11-5
Addition and Subtraction	11-7
Vector Productsand Transpose.....	11-7
Matrix Multiplication.....	11-9
The Identity Matrix.....	11-11
The Kronecker Tensor Product	11-11
Vector and Matrix Norms	11-12
Solving Linear Systems of Equations	11-13
Overview	11-13
Square Systems.....	11-15
Overdetermined Systems	11-15
Underdetermined Systems.....	11-18
Inverses and Determinants	11-21
Overview	11-21
Pseudoinverses.....	11-22
Cholesky, LU, and QR Factorizations	11-25
Cholesky Factorization.....	11-25
LU Factorization	11-26
QR Factorization	11-28
Matrix P o w ers and Exponentials	11-32
Eigenvalues	11-35
Singular Value Decomposition	11-39

MATLAB supports many matrix operations that are commonly used in *linear algebra*, including matrix arithmetic, linear equations, eigenvalues, singular values, and matrix factorizations. MATLAB enables you to work with entire matrices quickly and easily.

This chapter describes basic matrix operations in MATLAB and explains their use in solving problems. It includes:

Function Summary

Summarizes the MATLAB linear algebra functions

Matrices in MATLAB

Explains the use of matrices and basic matrix operations in MATLAB

Solving Linear Systems of Equations

Discusses the solution of simultaneous linear equations in MATLAB, including square systems, overdetermined systems, and underdetermined systems

Inverses and Determinants

Explains the use in MATLAB of inverses, determinants, and pseudoinverses in the solution of systems of linear equations

Cholesky, LU, and QR Factorizations

Discusses the solution in MATLAB of systems of linear equations that involve triangular matrices, using Cholesky factorization, Gaussian elimination, and orthogonalization

Matrix Powers and Exponentials

Explains the use of MATLAB notation to obtain various matrix powers and exponentials

Eigenvalues

Explains eigenvalues and describes eigenvalue decomposition in MATLAB

Singular Value Decomposition

Describes singular value decomposition of a rectangular matrix in MATLAB Function Summary

Function Summary

The linear algebra functions are located in the MATLABmatfun directory.

Function Summary Category Function Description Matrix analysis norm Matrix or vector norm. normest Estimate the matrix 2-norm. rank Matrix rank. det Determinant. trace Sum of diagonal elements. null Null space. orth Orthogonalization. rref Reduced row echelon form. subspace Angle between two subspaces. Linear equations \ and / Linear equation solution. inv Matrix inverse. cond Condition number for inversion. condest 1-norm condition number estimate. chol Cholesky factorization. cholinc Incomplete Cholesky factorization. lu LU factorization. luinc Incomplete LU factorization. qr Orthogonal-triangular decomposition. lsqnonneg Nonnegative least-squares.

Function Summary (Continued) Category Function Description pinv Pseudoinverse. lscov Least squares with known covariance. Eigenvalues and singular values eig Eigenvalues and eigenvectors.

svd Singular value decomposition. eigs A few eigenvalues. svds A few singular values. poly Characteristic polynomial. polyeig Polynomial eigenvalue problem. condeig Condition number for eigenvalues. hess Hessenberg form. qz QZ factorization. schur Schur decomposition. Matrix functions expm Matrix exponential. logm Matrix logarithm. sqrtm Matrix square root. funm Evaluate general matrix function.

M atrices in MATLAB

A *matrix* is a two-dimensional array of real or complex numbers. *Linear algebra* defines many matrix operations that are directly supported by MATLAB. Linear algebra includes matrix arithmetic, linear equations, eigenvalues, singular values, and matrix factorizations.

This section describes these matrix operations:

- Creation
- Addition and subtraction
- Vector products
- Matrix multiplication

It also describes the MATLAB functions you use to produce:

- An identity matrix
- The Kronecker Tensor product of two matrices
- Vector and matrix norms

Creation

Informally, the terms matrix and array are often used interchangeably. More precisely, a matrix is a two-dimensional rectangular array of real or complex numbers that represents a linear transformation. The linear algebraic operations defined on matrices have found applications in a wide variety of technical fields. (The optional Symbolic Math Toolbox extends MATLAB's capabilities to operations on various types of nonnumeric matrices.)

MATLAB has dozens of functions that create different kinds of matrices. Two of them can be used to create a pair of 3-by-3 example matrices for use throughout this chapter. The first example is symmetric.

```
A = pascal(3)
```

```
A =
1 1 1
1 2 3
1 3 6
```

The second example is not symmetric.

```
B = magic(3)
```

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

Another example is a 3-by-2 rectangular matrix of random integers.

```
C = fix(10*rand(3,2))
```

```
C =
9 4
2 8
6 7
```

A *column vector* is an m -by-1 matrix, a *row vector* is a 1-by- n matrix and a *scalar* is a 1-by-1 matrix. The statements

```
u = [3; 1; 4]
v = [2 0 -1]
```

$s = 7$ produce a column vector, a row vector, and a scalar.

```
u =
3
```

1
4

v = 2 0 -1
s =

Addition and Subtraction

Addition and subtraction of matrices is defined just as it is for arrays, element-by-element. Adding A to B and then subtracting A from the result recovers B.

X = A + B

X =
9 2 7
4 7 10
5 12 8

Y = X - A

Y =
8 1 6
3 5 7
4 9 2

Addition and subtraction require both matrices to have the same dimension, or one of them be a scalar. If the dimensions are incompatible, an error results.

X = A + C
Error using ==> +
Matrix dimensions must agree.

w = v + s
w = 9 7 6

Vector Products and Transpose

A row vector and a column vector of the same length can be multiplied in either order. The result is either a scalar, the *inner* product, or a matrix, the *outer* product.

x = v*u
x = X = u*v

X =
6 0 -3
2 0 -1
8 0 -4

For real matrices, the *transpose* operation interchanges a_{ij} and a_{ji} . MATLAB uses the apostrophe (or single quote) to denote transpose. Our example matrix A is *symmetric*, so A' is equal to A. But B is not symmetric.

X = B'

X =
8 3 4
1 5 9
6 7 2

Transposition turns a row vector into a column vector.

x = v'

x =
2
0
-1

If x and y are both real column vectors, the product x*y is not defined, but the two products

x'*y
and
y'*x

are the same scalar. This quantity is used so frequently, it has three different names: *inner* product, *scalar* product, or *dot* product.

For a complex vector or matrix, z, the quantity z' denotes the *complex conjugate transpose*, where the sign of the complex part of each element is reversed. The unconjugated complex transpose, where the complex part of each element retains its sign, is denoted by z.'. So if

z = [1+2i 3+4i] then z' is

1-2i
3-4i
while z.' is
1+2i
3+4i

For complex vectors, the two scalar products x'*y and y'*x are complex conjugates of each other and the scalar product x'*x of a complex vector with itself is real.

M atrix Multiplication

Multiplication of matrices is defined in a way that reflects composition of the underlying linear transformations and allows compact representation of systems of simultaneous linear equations. The matrix product $C = AB$ is defined when the column dimension of A is equal to the row

dimension of B , or when one of them is a scalar. If A is m -by- p and B is p -by- n , their product C is m -by- n . The product can actually be defined using MATLAB for loops, colon notation, and vector dot products.

```
for i = 1:m  
for j = 1:n  
C(i,j) = A(i,:)*B(:,j);  
end  
end
```

MATLAB uses a single asterisk to denote matrix multiplication. The next two examples illustrate the fact that matrix multiplication is not commutative; AB is usually not equal to BA .

$X = A * B$

$X =$
15 15 15
26 38 26
41 70 39
 $Y = B * A$

$Y =$
15 28 47
15 34 60
15 28 43

A matrix can be multiplied on the right by a column vector and on the left by a row vector.
 $x = A * u$

$x =$
8
17
30

$y = v * B$
 $y =$ 12 -7 10

Rectangular matrix multiplications must satisfy the dimension compatibility conditions.
 $X = A * C$

$X =$
17 19
31 41
51 70

$Y = C * A$
Error using ==> *
Inner matrix dimensions must agree.
Anything can be multiplied by a scalar. $w = s * v$
 $w =$ 14 0 -7

The Identity M atrix

Generally accepted mathematical notation uses the capital letter I to denote *identity* matrices, matrices of various sizes with ones on the main diagonal and zeros elsewhere. These matrices have the property that $AI = A$ and $IA = A$ whenever the dimensions are compatible. The original version of MATLAB could not use I for this purpose because it did not distinguish between upper and lowercase letters and i already served double duty as a subscript and as the complex unit. So an English language pun was introduced. The function

`eye(m,n)`

returns an m -byn rectangular identity matrix and `eye(n)` returns an n -byn square identity matrix.

The Kronecker Tensor Product

The Kronecker product, `kron(X,Y)`, of two matrices is the larger matrix formed from all possible products of the elements of X with those of Y . If X is m -byn and Y is p -by q , then `kron(X,Y)` is mp -by nq . The elements are arranged in the order

$[X(1,1)*Y \ X(1,2)*Y \ \dots \ X(1,n)*Y$

\dots

$X(m,1)*Y \ X(m,2)*Y \ \dots \ X(m,n)*Y]$

The Kronecker product is often used with matrices of zeros and ones to build up repeated copies of small matrices. For example, if X is the 2-by-2 matrix

$X =$
1 2
3 4

and $I = \text{eye}(2,2)$ is the 2-by-2 identity matrix, then the two matrices `kron(X,I)`

and

`kron(I,X)`

are

1 0 2 0
0 1 0 2
3 0 4 0
0 3 0 4

and

1 2 0 0
3 4 0 0
0 0 1 2
0 0 3 4

Vector and M atrix N orms

The p -norm of a vector x

p

$\}$
 $1 \frac{d}{c} p$
 $x_p = \star \circ x_i \backslash$
 \star

is computed by $\text{norm}(x,p)$. This is defined by any value of $p > 1$, but the most common values of p are 1, 2, and ∞ . The default value is $p = 2$, which corresponds to *Euclidean length*.

[$\text{norm}(v,1)$ $\text{norm}(v)$ $\text{norm}(v,\infty)$]

ans = 3.0000 2.2361 2.0000 The p -norm of a matrix A , $\max_{pA} x_p = x$ can be computed for $p = 1, 2$, and ∞ by $\text{norm}(A,p)$. Again, the default value is $p = 2$.

[$\text{norm}(C,1)$ $\text{norm}(C)$ $\text{norm}(C,\infty)$]

ans = 19.0000 14.8015 13.0000

Solving Linear Systems of Equations

This section describes:

- The general solution of systems of linear equations
- The solution of square systems
- The solution of overdetermined systems
- The solution of underdetermined systems

Overview

One of the most important problems in technical computing is the solution of simultaneous linear equations. In matrix notation, this problem can be stated as follows.

Given two matrices A and B , does there exist a unique matrix X so that $AX = B$ or $XA = B$? It is instructive to consider a 1-by-1 example.

Does the equation

$$7x = 21$$

have a unique solution?

The answer, of course, is yes. The equation has the unique solution $x = 3$. The solution is easily obtained by *division*.

$$x = 21 \frac{d}{c} 7 = 3$$

The solution is *not* ordinarily obtained by computing the inverse of 7, that is $7^{-1} = 0.142857\dots$, and then multiplying 7^{-1} by 21. This would be more work and, if 7^{-1} is represented to a finite number of digits, less accurate. Similar considerations apply to sets of linear equations with more than one unknown; MATLAB solves such equations without computing the inverse of the matrix.

Although it is not standard mathematical notation, MATLAB uses the division terminology familiar in the scalar case to describe the solution of a general system of simultaneous equations. The two division symbols, *slash*, $/$, and *backslash*, \backslash , are used for the two situations where the unknown matrix appears on the left or right of the coefficient matrix.

$X = A \setminus B$ Denotes the solution to the matrix equation $AX = B$. $X = B / A$ Denotes the solution to the matrix equation $XA = B$.

You can think of “dividing” both sides of the equation $AX = B$ or $XA = B$ by A . The coefficient matrix A is always in the “denominator.”

The dimension compatibility conditions for $X = A \setminus B$ require the two matrices A and B to have the same number of rows. The solution X then has the same number of columns as B and its row dimension is equal to the column dimension of A . For $X = B / A$, the roles of rows and columns are interchanged.

In practice, linear equations of the form $AX = B$ occur more frequently than those of the form $XA = B$. Consequently, backslash is used far more frequently than slash. The remainder of this section concentrates on the backslash operator; the corresponding properties of the slash operator can be inferred from the identity

$$(B/A)' = (A' \setminus B')$$

The coefficient matrix A need not be square. If A is m -by- n , there are three cases.

$m = n$ Square system. Seek an exact solution.

$m > n$ Overdetermined system. Find a least squares solution.

$m < n$ Underdetermined system. Find a basic solution with at most m nonzero components.

The backslash operator employs different algorithms to handle different kinds of coefficient matrices. The various cases, which are diagnosed automatically by examining the coefficient matrix, include:

- Permutations of triangular matrices
- Symmetric, positive definite matrices
- Square, nonsingular matrices
- Rectangular, overdetermined systems
- Rectangular, underdetermined systems

Square Systems

The most common situation involves a square coefficient matrix A and a single right-hand side column vector b . The solution, $x = A \setminus b$, is then the same size as b . For example,

$$x = A \setminus u$$

$$\begin{matrix} x = 10 \\ -12 \\ 5 \end{matrix}$$

It can be confirmed that A^*x is exactly equal to u .

If A and B are square and the same size, then $X = A \setminus B$ is also that size.

$$X = A \setminus B$$

```
X = 19 -3 -1
-17 4 13
6 0 -6
```

It can be confirmed that A^*X is exactly equal to B .

Both of these examples have exact, integer solutions. This is because the coefficient matrix was chosen to be `pascal(3)`, which has a determinant equal to one. A later section considers the effects of roundoff error inherent in more realistic computation.

A square matrix A is *singular* if it does not have linearly independent columns. If A is singular, the solution to $AX = B$ either does not exist, or is not unique. The backslash operator, `A\B`, issues a warning if A is nearly singular and raises an error condition if exact singularity is detected.

Overdetermined Systems

Overdetermined systems of simultaneous linear equations are often encountered in various kinds of curve fitting to experimental data. Here is a hypothetical example. A quantity y is measured at several different values of time, t , to produce the following observations.

t	y
0.0	0.82
0.3	0.72
0.8	0.63
1.1	0.60
1.6	0.55
2.3	0.50

This data can be entered into MATLAB with the statements `t = [0 .3 .8 1.1 1.6 2.3]';
y = [.82 .72 .63 .60 .55 .50]'`;

The data can be modeled with a decaying exponential function.

$$-t \mathbf{y} \approx c_1 + c_2 e^{-t}$$

This equation says that the vector y should be approximated by a linear combination of two other vectors, one the constant vector containing all ones and the other the vector with components e^{-t} . The unknown coefficients, c_1 and c_2 , can be computed by doing a *least squares fit*, which minimizes the sum of the squares of the deviations of the data from the model. There are six equations in two unknowns, represented by the 6-by-2 matrix.

$$E = [\text{ones}(\text{size}(t)) \exp(-t)]$$

```
E =
1.0000 1.0000
1.0000 0.7408
1.0000 0.4493
1.0000 0.3329
1.0000 0.2019
```

1.0000 0.1003

The least squares solution is found with the backslash operator.

$c = E \setminus y$

$c =$
0.4760
0.3413

In other words, the least squares fit to the data is

$y \approx 0.4760 + 0.3413 e^{-t}$ The following statements evaluate the model at regularly spaced increments in t , and then plot the result, together with the original data.

```
T = (0:0.1:2.5)';
Y = [ones(size(T)) exp(-T)]*c;
plot(T,Y,'-',t,y,'o')
```

You can see that $E*c$ is not exactly equal to y , but that the difference might well be less than measurement errors in the original data.

A rectangular matrix A is *rank deficient* if it does not have linearly independent columns. If A is rank deficient, the least squares solution to $AX = B$ is not unique. The backslash operator, $A \setminus B$, issues a warning if A is rank deficient and produces a *basic* solution that has as few nonzero elements as possible.

0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.50 0.5 1 1.5 2 2.5

Underdetermined Systems

Underdetermined linear systems involve more unknowns than equations. When they are accompanied by additional constraints, they are the purview of *linear programming*. By itself, the backslash operator deals only with the unconstrained system. The solution is never unique. MATLAB finds a *basic* solution, which has at most m nonzero components, but even this may not be unique. The particular solution actually computed is determined by the QR factorization with column pivoting (see a later section on the QR factorization). Here is a small, random example.

$R = fix(10 * rand(2,4))$

$R =$
6 8 7 3
3 5 4 1

```
b = fix(10*rand(2,1))
```

```
b =  
1  
2
```

The linear system $Rx = b$ involves two equations in four unknowns. Since the coefficient matrix contains small integers, it is appropriate to use the format command to display the solution in *rational* format. The particular solution is obtained with

```
format rat
```

```
p = R\b
```

```
p = 0 5/7 0
```

```
-11/7
```

One of the nonzero components is $p(2)$ because $R(:,2)$ is the column of R with largest norm. The other nonzero component is $p(4)$ because $R(:,4)$ dominates after $R(:,2)$ is eliminated.

The complete solution to the overdetermined ([Overdetermined?](#)) system can be characterized by adding an arbitrary vector from the null space, which can be found using the `null` function with an option requesting a “rational” basis.

```
Z = null(R,'r')
```

```
Z =  
-1/2 -7/6  
-1/2 1/2  
1 0 0 1
```

It can be confirmed that R^*Z is zero and that any vector x where $x = p + Z^*q$ for an arbitrary vector q satisfies $R^*x = b$.

Inverses and Determinants

This section provides:

- An overview of the use of inverses and determinants for solving square nonsingular systems of linear equations
- A discussion of the Moore-Penrose pseudoinverse for solving rectangular systems of linear equations

Overview

If A is square and nonsingular, the equations $AX = I$ and $XA = I$ have the same solution, X . This solution is called the *inverse* of A , is denoted by A^{-1} , and is computed by the function `inv`. The *determinant* of a matrix is useful in theoretical considerations and some types of symbolic computation, but its scaling and roundoff error properties make it far less satisfactory for

numeric computation. Nevertheless, the function `det` computes the determinant of a square matrix.

```
A = pascal(3)
```

```
A =  
1 1 1  
1 2 3  
1 3 6
```

```
d = det(A) X = inv(A)  
d = 1
```

```
X = 3 -3 1  
-3 5 -2  
1 -2 1
```

Again, because `A` is symmetric, has integer elements, and has determinant equal to one, so does its inverse. On the other hand,

```
B = magic(3)  
B = 8 1 6 3 5 7 4 9 2
```

```
d = det(B) X = inv(B)  
d =  
-360
```

```
X = 0.1472 -0.1444 0.0639  
-0.0611 0.0222 0.1056  
-0.0194 0.1889 -0.1028
```

Closer examination of the elements of `X`, or use `format rat`, would reveal that they are integers divided by 360.

If `A` is square and nonsingular, then without roundoff error, `X = inv(A)*B` would theoretically be the same as `X = A\B` and `Y = B*inv(A)` would theoretically be the same as `Y = B/A`. But the computations involving the backslash and slash operators are preferable because they require less computer time, less memory, and have better error detection properties.

Pseudoinverses

Rectangular matrices do not have inverses or determinants. At least one of the equations $AX = I$ and $XA = I$ does not have a solution. A partial replacement for the inverse is provided by the *Moore-Penrose pseudoinverse*, which is computed by the `pinv` function.

```
X = pinv(C)
```

```
X = 0.1159 -0.0729 0.0171
```

-0.0534 0.1152 0.0418

The matrix

$$Q = X * C$$

$Q =$

1.0000 0.0000
0.0000 1.0000

is the 2-by-2 identity, but the matrix

$$P = C * X$$

P = 0.8293 -0.1958 0.3213
-0.1958 0.7754 0.3685
0.3213 0.3685 0.3952

is not the 3-by-3 identity. However, P acts like an identity on a portion of the space in the sense that P is symmetric, P^*C is equal to C and X^*P is equal to X. If A is m-by-n with $m > n$ and full rank n, then each of the three statements

$x = A \backslash b$
 $x = \text{pinv}(A)^* b$
 $x = \text{inv}(A'^* A)^* A'^* b$

theoretically computes the same least squares solution x, although the backslash operator does it faster.

However, if A does not have full rank, the solution to the least squares problem is not unique.

There are many vectors x that minimize

$$\text{norm}(A^*x - b)$$

The solution computed by $x = A \backslash b$ is a *basic* solution; it has at most r nonzero components, where r is the rank of A. The solution computed by $x = \text{pinv}(A)^* b$ is the *minimal norm* solution because it minimizes $\text{norm}(x)$. An attempt to compute a solution with $x = \text{inv}(A'^* A)^* A'^* b$ fails because $A'^* A$ is singular.

Here is an example to illustrates the various solutions.

A =
[1 2 3
4 5 6
7 8 9
10 11 12]

does not have full rank. Its second column is the average of the first and third columns. If $b = A(:,2)$

is the second column, then an obvious solution to $A^*x = b$ is $x = [0 1 0]'$. But none of the approaches computes that x. The backslash operator gives

$$x = A \backslash b$$

Warning: Rank deficient, rank = 2.

```
x =
0.5000
0
0.5000
```

This solution has two nonzero components. The pseudoinverse approach gives
 $y = \text{pinv}(A)*b$

```
y =
0.3333
0.3333
0.3333
```

There is no warning about rank deficiency. But $\text{norm}(y) = 0.5774$ is less than $\text{norm}(x) = 0.7071$.
Finally

```
z = inv(A'*A)*A'*b
fails completely.
Warning: Matrix is singular to working precision.
```

```
z = Inf Inf Inf
```

Cholesky, LU, and QR Factorizations

MATLAB's linear equation capabilities are based on three basic matrix factorizations:

- Cholesky factorization for symmetric, positive definite matrices
- LU factorization (Gaussian elimination) for general square matrices
- QR (orthogonal) for rectangular matrices

These three factorizations are available through the `chol`, `lu`, and `qr` functions.

All three of these factorizations make use of *triangular* matrices where all the elements either above or below the diagonal are zero. Systems of linear equations involving triangular matrices are easily and quickly solved using either *forward* or *back substitution*.

Cholesky Factorization

The Cholesky factorization expresses a symmetric matrix as the product of a triangular matrix and its transpose

$$A = R^T R$$

where R is an upper triangular matrix.

Not all symmetric matrices can be factored in this way; the matrices that have such a factorization are said to be *positive definite*. This implies that all the diagonal elements of A are positive and that the offdiagonal elements are “not too big.” The Pascal matrices provide an interesting example. Throughout this chapter, our example matrix A has been the 3-by-3 Pascal

matrix. Let's temporarily switch to the 6-by-6.

```
A = pascal(6)
```

```
A =
1 1 1 1 1 1
1 2 3 4 5 6
1 3 6 10 15 21
1 4 10 20 35 56
1 5 15 35 70 126
1 6 21 56 126 252
```

The elements of A are binomial coefficients. Each element is the sum of its north and west neighbors. The Cholesky factorization is

```
R = chol(A)
```

```
R =
1 1 1 1 1 1
0 1 2 3 4 5
0 0 1 3 6 10
0 0 0 1 4 10
0 0 0 0 1 5
0 0 0 0 0 1
```

The elements are again binomial coefficients. The fact that R^*R is equal to A demonstrates an identity involving sums of products of binomial coefficients.

Note The Cholesky factorization also applies to complex matrices. Any complex matrix which has a Cholesky factorization satisfies $A' = A$ and is said to be *Hermitian positive definite*.

The Cholesky factorization allows the linear system $Ax = b$ to be replaced by $R^T R x = b$

Because the backslash operator recognizes triangular systems, this can be solved in MATLAB quickly with

```
x = R\(\b)
```

If A is n -by- n , the computational complexity of $\text{chol}(A)$ is $O(n^3)$, but the complexity of the subsequent backslash solutions is only $O(n^2)$.

LU Factorization

LU factorization, or Gaussian elimination, expresses any square matrix A as the product of a permutation of a lower triangular matrix and an upper triangular matrix

```
A = LU
```

where L is a permutation of a lower triangular matrix with ones on its diagonal and U is an upper triangular matrix.

The permutations are necessary for both theoretical and computational reasons. The matrix $\begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ cannot be expressed as the product of triangular matrices without interchanging its two

rows. Although the matrix

$$\begin{pmatrix} & 1 & 1 & 0 \end{pmatrix}$$

can be expressed as the product of triangular matrices, when ω is small the elements in the factors are large and magnify errors, so even though the permutations are not strictly necessary, they are desirable. *Partial pivoting* ensures that the elements of L are bounded by one in magnitude and that the elements of U are not much larger than those of A .

For example

$$[L, U] = lu(B)$$

$L =$

$$\begin{pmatrix} 1.0000 & 0 & 0 \\ 0.3750 & 0.5441 & 1.0000 \\ 0.5000 & 1.0000 & 0 \end{pmatrix}$$

$U =$ 8.0000 1.0000 6.0000

0 8.5000 -1.0000

0 0 5.2941

The LU factorization of A allows the linear system

$$A^*x = b$$

to be solved quickly with

$$x = U \setminus (L \setminus b)$$

Determinants and inverses are computed from the LU factorization using $\det(A) = \det(L)^*\det(U)$ and

$$\text{inv}(A) = \text{inv}(U)^*\text{inv}(L)$$

You can also compute the determinants using $\det(A) = \text{prod}(\text{diag}(U))$, though the signs of the determinants may be reversed.

QR Factorization

An *orthogonal* matrix, or a matrix with *orthonormal columns*, is a real matrix whose columns all have unit length and are perpendicular to each other. If Q is orthogonal, then

$$Q^T Q = I$$

The simplest orthogonal matrices are two-dimensional coordinate rotations.

$$\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

For complex matrices, the corresponding term is *unitary*. Orthogonal and unitary matrices are desirable for numerical computation because they preserve length, preserve angles, and do not magnify errors.

The orthogonal, or QR, factorization expresses any rectangular matrix as the product of an orthogonal or unitary matrix and an upper triangular matrix. A column permutation may also be involved.

$$A = Q R$$

or

$$A P = Q R$$

where Q is orthogonal or unitary, R is upper triangular, and P is a permutation.

There are four variants of the QR factorization— full or economy size, and with or without column permutation.

Overdetermined linear systems involve a rectangular matrix with more rows than columns, that is m -by- n with $m > n$. The *full* size QR factorization produces a square, m -by- m orthogonal Q and a rectangular m -by- n upper triangular R .

$$[Q, R] = qr(C)$$

$$Q =$$

```
-0.8182 0.3999 -0.4131  
-0.1818 -0.8616 -0.4739  
-0.5455 -0.3126 0.7777
```

$$R =$$

```
-11.0000 -8.5455  
0 -7.4817  
0 0
```

In many cases, the last $m - n$ columns of Q are not needed because they are multiplied by the zeros in the bottom portion of R . So the *economy* size QR factorization produces a rectangular, m -by- n Q with orthonormal columns and a square n -by- n upper triangular R . For our 3-by-2 example, this is not much of a saving, but for larger, highly rectangular matrices, the savings in both time and memory can be quite important.

$$[Q, R] = qr(C, 0)$$

$$Q =$$

```
-0.8182 0.3999  
-0.1818 -0.8616  
-0.5455 -0.3126
```

$$R =$$

```
-11.0000 -8.5455  
0 -7.4817
```

In contrast to the LU factorization, the QR factorization does not require any pivoting or permutations. But an optional column permutation, triggered by the presence of a third output argument, is useful for detecting singularity or rank deficiency. At each step of the factorization, the column of the remaining unfactored matrix with largest norm is used as the basis for that step. This ensures that the diagonal elements of R occur in decreasing order and that any linear dependence among the columns is almost certainly be revealed by examining these elements. For our small example, the second column of C has a larger norm than the first, so the two columns

are exchanged.

$[Q, R, P] = qr(C)$

$Q =$
-0.3522 0.8398 -0.4131
-0.7044 -0.5285 -0.4739
-0.6163 0.1241 0.7777

$R =$
-11.3578 -8.2762
0 7.2460
0 0

$P =$
0 1
1 0

When the economy size and column permutations are combined, the third output argument is a permutation vector, rather than a permutation matrix.

$[Q, R, p] = qr(C, 0)$

$Q =$
-0.3522 0.8398
-0.7044 -0.5285
-0.6163 0.1241

$R =$
-11.3578 -8.2762
0 7.2460
 $p = 2 \ 1$

The QR factorization transforms an overdetermined linear system into an equivalent triangular system. The expression

$\text{norm}(A^*x - b)$
is equal to
 $\text{norm}(Q^*R^*x - b)$

Multiplication by orthogonal matrices preserves the Euclidean norm, so this expression is also equal to

$\text{norm}(R^*x - y)$

where $y = Q^*b$. Since the last $m-n$ rows of R are zero, this expression breaks into two pieces

$\text{norm}(R(1:n, 1:n)^*x - y(1:n))$

and

$\text{norm}(y(n+1:m))$

When A has full rank, it is possible to solve for x so that the first of these expressions is zero. Then the second expression gives the norm of the residual. When A does not have full rank, the triangular structure of R makes it possible to find a basic solution to the least squares problem.

M atrix Po w ers and Exponentials

This section tells you how to obtain the following matrix powers and exponentials in MATLAB:

- Positive integer
- Inverse and fractional
- Element-by-element
- Exponentials

Positive Integer Powers

If A is a square matrix and p is a positive integer, then A^p effectively multiplies A by itself $p-1$ times.

$$X = A^2$$

$$\begin{matrix} X = \\ 3 & 6 & 10 \\ 6 & 14 & 25 \\ 10 & 25 & 46 \end{matrix}$$

Inverse and Fractional Powers

If A is square and nonsingular, then $A^{(-p)}$ effectively multiplies $\text{inv}(A)$ by itself $p-1$ times.

$$Y = B^{(-3)}$$

$$\begin{matrix} Y = 0.0053 & -0.0068 & 0.0018 \\ -0.0034 & 0.0001 & 0.0036 \\ -0.0016 & 0.0070 & -0.0051 \end{matrix}$$

Fractional powers, like $A^{(2/3)}$, are also permitted; the results depend upon the distribution of the eigenvalues of the matrix.

Element-by-Element Powers

The $.^{\wedge}$ operator produces element-by-element powers. For example, $X = A.^{\wedge}2$

Matrix Powers and Exponentials

$$\begin{matrix} A = \\ 1 & 1 & 1 \\ 1 & 4 & 9 \\ 1 & 9 & 36 \end{matrix}$$

Exponentials

The function
`sqrtm(A)`

computes $A^{(1/2)}$ by a more accurate algorithm. The function `sqrtm` distinguishes this function from `sqrt(A)` which, like $A.^{\wedge}(1/2)$, does its job

element-by-element.

A system of linear, constant coefficient, ordinary differential equations can be written
$$dx_c^{\frac{d}{dt}} d t = Ax$$

where $x = x(t)$ is a vector of functions of t and A is a matrix independent of t . The solution can be expressed in terms of the *matrix exponential* ,
$$x(t) = e^{tA}x(0)$$

The function
`expm(A)`

computes the matrix exponential. An example is provided by the 3-by-3 coefficient matrix

$A =$
0 -6 -1
6 2 -16
-5 20 -10

and the initial condition, $x(0)$

$x0 =$
1
1
1

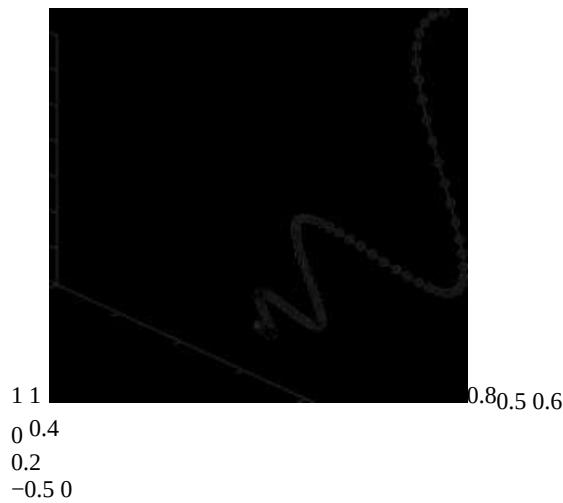
The matrix exponential is used to compute the solution, $x(t)$, to the differential equation at 101 points on the interval $0 \leq t \leq 1$ with

```
X = [];
for t = 0:.01:1
X = [X expm(t*A)*x0];
end
```

A three-dimensional phase plane plot obtained with
`plot3(X(1,:),X(2,:),X(3,:),'-o')`

shows the solution spiraling in towards the origin. This behavior is related to the eigenvalues of the coefficient matrix, which are discussed in the next section.

1.2
1
0.8
0.6
0.4
0.2
0
-0.2
1.5



Eigenvalues

An *eigenvalue* and *eigenvector* of a square matrix A are a scalar λ and a nonzero vector v that satisfy

$$Av = \lambda v$$

This section explains:

- Eigenvalue decomposition
- Problems associated with defective (not diagonalizable) matrices
- The use of Schur decomposition to avoid problems associated with eigenvalue decomposition

Eigenvalue Decomposition

With the eigenvalues on the diagonal of a diagonal matrix Λ and the corresponding eigenvectors forming the columns of a matrix V , we have $AV = V\Lambda$

If V is nonsingular, this becomes the *eigenvalue decomposition*

$$A^{-1} = V\Lambda V^{-1}$$

A good example is provided by the coefficient matrix of the ordinary differential equation in the previous section.

$$\begin{aligned} A = \\ 0 & -6 & -1 \\ 6 & 2 & -16 \\ -5 & 20 & -10 \end{aligned}$$

The statement

$$\text{lambda} = \text{eig}(A)$$

produces a column vector containing the eigenvalues. For this matrix, the eigenvalues are complex.

$$\begin{aligned} \text{lambda} = \\ -3.0710 \\ -2.4645 + 17.6008i \\ -2.4645 - 17.6008i \end{aligned}$$

The real part of each of the eigenvalues is negative, so $e^{\frac{-}{\lambda}t}$ approaches zero as t increases. The nonzero imaginary part of two of the eigenvalues, $\frac{238}{93}\sqrt{-1}$, contributes the oscillatory component, $\sin(\frac{238}{93}\sqrt{-1}t)$, to the solution of the differential equation.

With two output arguments, eig computes the eigenvectors and stores the eigenvalues in a diagonal matrix.

$[V,D] = \text{eig}(A)$

$V =$

```
-0.8326 0.2003 - 0.1394i 0.2003 + 0.1394i
-0.3553 -0.2110 - 0.6447i -0.2110 + 0.6447i
-0.4248 -0.6930 -0.6930
```

$D =$

```
-3.0710 0 0
0 -2.4645+17.6008i 0
0 0 -2.4645-17.6008i
```

The first eigenvector is real and the other two vectors are complex conjugates of each other. All three vectors are normalized to have Euclidean length, norm(v,2), equal to one.

The matrix $V*D*inv(V)$, which can be written more succinctly as $V*D/V$, is within roundoff error of A. And, $inv(V)*A*V$, or $V\backslash A*V$, is within roundoff error of D.

Defective Matrices

Some matrices do not have an eigenvector decomposition. These matrices are *defective*, or *not diagonalizable*. For example,

$A =$

```
6 12 19
-9 -20 -33
4 9 15
```

For this matrix

$[V,D] = \text{eig}(A)$
produces

$V =$

```
-0.4741 -0.4082 -0.4082 0.8127 0.8165 0.8165
-0.3386 -0.4082 -0.4082
```

$D =$

```
-1.0000 0 0 0 1.0000 0 0 0 1.0000
```

There is a double eigenvalue at $\lambda = 1$. The second and third columns of V are the same. For this matrix, a full set of linearly independent eigenvectors does not exist.

The optional Symbolic Math Toolbox extends MATLAB's capabilities by connecting to Maple, a powerful computer algebra system. One of the functions provided by the toolbox computes the Jordan Canonical Form. This is appropriate for matrices like our example, which is 3-by-3 and has exactly known, integer elements.

```
[X,J] = jordan(A)
```

```
X =
-1.7500 1.5000 2.7500
3.0000 -3.0000 -3.0000
-1.2500 1.5000 1.2500
```

```
J =
-1 0 0
0 1 1
0 0 1
```

The Jordan Canonical Form is an important theoretical concept, but it is not a reliable computational tool for larger matrices, or for matrices whose elements are subject to roundoff errors and other uncertainties.

Schur Decomposition in MATLAB Matrix Computations MATLAB's advanced matrix computations do not require eigenvalue decompositions. They are based, instead, on the *Schur decomposition*

```
A = U S UT
```

where U is an orthogonal matrix and S is a block upper triangular matrix with 1-by-1 and 2-by-2 blocks on the diagonal. The eigenvalues are revealed by the diagonal elements and blocks of S , while the columns of U provide a basis with much better numerical properties than a set of eigenvectors. The Schur decomposition of our defective example is

```
[U,S] = schur(A)
```

```
U = 0.4741 -0.6571 0.5861
-0.8127 -0.0706 0.5783
0.3386 0.7505 0.5675
```

```
S =
-1.0000 21.3737 44.4161
0 1.0081 0.6095
0 -0.0001 0.9919
```

The double eigenvalue is contained in the lower 2-by-2 block of S .

Note If A is complex, $schur$ returns the complex Schur form, which is upper triangular with the eigenvalues of A on the diagonal.

Singular Value Decomposition

Singular Value Decomposition

A *singular value* and corresponding *singular vectors* of a rectangular matrix A are a scalar σ and a pair of vectors u and v that satisfy

$$Av = \sigma u$$

$$A^T u = \sigma v$$

With the singular values on the diagonal of a diagonal matrix Σ and the corresponding singular vectors forming the columns of two orthogonal matrices U and V , we have

$$AV = U\Sigma$$

$$A^T U = V\Sigma$$

Since U and V are orthogonal, this becomes the *singular value decomposition*

$$A = U\Sigma V^T$$

The full singular value decomposition of an m -by- n matrix involves an m -by- m U , an m -by- n Σ , and an n -by- n V . In other words, U and V are both square and Σ is the same size as A . If A has many more rows than columns, the resulting U can be quite large, but most of its columns are multiplied by zeros in Σ . In this situation, the *economy* sized decomposition saves both time and storage by producing an m -by- n U , an n -by- n Σ and the same V .

The eigenvalue decomposition is the appropriate tool for analyzing a matrix when it represents a mapping from a vector space into itself, as it does for an ordinary differential equation. On the other hand, the singular value decomposition is the appropriate tool for analyzing a mapping from one vector space into another vector space, possibly with a different dimension. Most systems of simultaneous linear equations fall into this second category.

If A is square, symmetric, and positive definite, then its eigenvalue and singular value decompositions are the same. But, as A departs from symmetry and positive definiteness, the difference between the two decompositions increases. In particular, the singular value decomposition of a real matrix is always real, but the eigenvalue decomposition of a real, nonsymmetric matrix might be complex.

For the example matrix

$$A =$$

$$\begin{matrix} 9 & 4 \\ 6 & 8 \end{matrix}$$

$$\begin{matrix} 2 & 7 \end{matrix}$$

the full singular value decomposition is

$$[U, S, V] = \text{svd}(A)$$

$$U =$$

$$\begin{matrix} -0.6105 & 0.7174 & 0.3355 \end{matrix}$$

$$\begin{matrix} -0.6646 & -0.2336 & -0.7098 \end{matrix}$$

$$\begin{matrix} -0.4308 & -0.6563 & 0.6194 \end{matrix}$$

$$S = 14.9359 \ 0$$

$$0 \ 5.1883$$

0 0

V =
-0.6925 0.7214
-0.7214 -0.6925

You can verify that U^*S^*V' is equal to A to within roundoff error. For this small problem, the economy size decomposition is only slightly smaller.

[U,S,V] = svd(A,0)

U =
-0.6105 0.7174
-0.6646 -0.2336
-0.4308 -0.6563

S = 14.9359 0
0 5.1883

V =
-0.6925 0.7214
-0.7214 -0.6925

Again, U^*S^*V' is equal to A to within roundoff error.

12

Polynomials and Interpolation

P olynomials.....	12-3
Polynomial Function Summary.....	12-3
Representing Polynomials.....	12-4
Polynomial Roots.....	12-4
Characteristic Polynomials	12-5
Polynomial Evaluation.....	12-5
Convolution and Deconvolution.....	12-6
Polynomial Derivatives	12-6
Polynomial Curve Fitting.....	12-7
Partial Fraction Expansion	12-8
 I nterpolation	 12-10
Interpolation Function Summary	12-10
One-Dimensional Interpolation.....	12-11
Two-Dimensional Interpolation.....	12-13
Comparing Interpolation Methods.....	12-14
Interpolation and Multidimensional Arrays	12-16
Triangulation and Interpolation of Scattered Data.....	12-19
Tessellation and Interpolation of Scattered Data in Higher Dimensions.....	12-27

Selected Bibliography12-38

This chapter introduces MATLAB functions that enable you to work with polynomials and interpolate one-, two-, and multi-dimensional data. It includes:

Polynomials

Functions for standard polynomial operations. Additional topics include curve fitting and partial fraction expansion.

Interpolation

Two- and multi-dimensional interpolation techniques, taking into account speed, memory, and smoothness considerations.

Polynomials

This section provides:

- A summary of the MATLAB polynomial functions
- Instructions for representing polynomials in MATLAB

It also describes the MATLAB polynomial functions that:

- Calculate the roots of a polynomial
- Calculate the coefficients of the characteristic polynomial of a matrix
- Evaluate a polynomial at a specified value
- Convolve (multiply) and deconvolve (divide) polynomials
- Compute the derivative of a polynomial
- Fit a polynomial to a set of data
- Convert between partial fraction expansion and polynomial coefficients

Polynomial Function Summary

MATLAB provides functions for standard polynomial operations, such as polynomial roots, evaluation, and differentiation. In addition, there are functions for more advanced applications, such as curve fitting and partial fraction expansion.

The polynomial functions reside in the MATLAB polyfun directory.

Polynomial Function Summary Function Description conv Multiply polynomials. deconv Divide polynomials. poly Polynomial with specified roots. polyder Polynomial derivative. polyfit Polynomial curve fitting. polyval Polynomial evaluation.

Polynomial Function Summary (Continued) Function Description polyvalm Matrix polynomial evaluation. residue Partial-fraction expansion (residues). roots Find polynomial roots.

The Symbolic Math Toolbox contains additional specialized support for polynomial operations.

Representing Polynomials

MATLAB represents polynomials as row vectors containing coefficients ordered by descending powers. For example, consider the equation

$p(x) = x^3 - 2x - 5$ This is the celebrated example Wallis used when he first represented Newton's method to the French Academy. To enter this polynomial into MATLAB, use $p = [1 0 -2 -5]$;

Polynomial Roots

The roots function calculates the roots of a polynomial. $r = \text{roots}(p)$

$r = 2.0946$

$-1.0473 + 1.1359i$

$-1.0473 - 1.1359i$ By convention, MATLAB stores roots in column vectors. The function poly returns to the polynomial coefficients.

$p2 = \text{poly}(r)$

$p2 = 1.8818e-16 - 2.5$ poly and roots are inverse functions, up to ordering, scaling, and roundoff error.

Characteristic Polynomials

The poly function also computes the coefficients of the characteristic polynomial of a matrix.

$A = [1.2 \ 3 \ -0.9; \ 5 \ 1.75 \ 6; \ 9 \ 0 \ 1];$

$\text{poly}(A)$

$\text{ans} =$

$1.0000 \ -3.9500 \ -1.8500 \ -163.2750$

The roots of this polynomial, computed with roots, are the *characteristic roots*, or eigenvalues, of the matrix A. (Use eig to compute the eigenvalues of a matrix directly.)

Polynomial Evaluation

The polyval function evaluates a polynomial at a specified value. To evaluate p at $s = 5$, use

$\text{polyval}(p, 5)$

$\text{ans} =$

110

It is also possible to evaluate a polynomial in a matrix sense. In this case $p(s) = s^3 - 2s - 5$ becomes $p(X) = X^3 - 2X - 5I$, where X is a square

matrix and I is the identity matrix. For example, create a square matrix X and evaluate the polynomial p at X.

$X = [2 \ 4 \ 5; \ -1 \ 0 \ 3; \ 7 \ 1 \ 5];$

$Y = \text{polyvalm}(p, X)$

$Y =$

$377 \ 179 \ 439$

$111 \ 81 \ 136$

$490 \ 253 \ 639$

Convolution and Deconvolution

Polynomial multiplication and division correspond to the operations convolution and deconvolution. The functions conv and deconv implement these operations.

Consider the polynomials $a(s) = s^2 + 2s + 3$ and $b(s) = 4s^2 + 5s + 6$. To compute their product,

$a = [1 \ 2 \ 3]; b = [4 \ 5 \ 6];$

$c = \text{conv}(a, b)$

$c = 4 \ 13 \ 28 \ 27 \ 18$ Use deconvolution to divide a(s) back out of the product. $[q, r] = \text{deconv}(c, a)$

$q = 4 \ 5 \ 6$

$r = 0 \ 0 \ 0 \ 0 \ 0$

Polynomial Derivatives

The `polyder` function computes the derivative of any polynomial. To obtain the derivative of the polynomial $p = [1 \ 0 \ -2 \ -5]$,

$q = \text{polyder}(p)$

$q = 3 \ 0 \ -2$ `polyder` also computes the derivative of the product or quotient of two polynomials.

For example, create two polynomials a and b . $a = [1 \ 3 \ 5]$;

$b = [2 \ 4 \ 6]$;

Calculate the derivative of the product $a * b$ by calling `polyder` with a single output argument.

$c = \text{polyder}(a,b)$

$c = 8 \ 30 \ 56 \ 38$ Calculate the derivative of the quotient a/b by calling `polyder` with two output arguments.

$[q,d] = \text{polyder}(a,b)$

$q =$

$-2 \ -8 \ -2$

$d = 4 \ 16 \ 40 \ 48 \ 36$ q/d is the result of the operation.

Polynomial Curve Fitting

`polyfit` finds the coefficients of a polynomial that fits a set of data in a least-squares sense.

$p = \text{polyfit}(x,y,n)$ x and y are vectors containing the x and y data to be fitted, and n is the order of the polynomial to return. For example, consider the x - y test data. $x = [1 \ 2 \ 3 \ 4 \ 5]$; $y = [5.5 \ 43.1 \ 128 \ 290.7 \ 498.4]$;

A third order polynomial that approximately fits the data is

$p = \text{polyfit}(x,y,3)$

$p =$

$-0.1917 \ 31.5821 \ -60.3262 \ 35.3400$ Compute the values of the `polyfit` estimate over a finer range, and plot the estimate over the real data values for comparison.

$x2 = 1:1:5;$

$y2 = \text{polyval}(p,x2);$

`plot(x,y,'o',x2,y2)`

`grid on`

500

450

400

350

300

250

200

150

100

50

0 1 1.5 2 2.5 3 3.5 4 4.5 5

To use these functions in an application example, see the “Data Analysis and Statistics” chapter.

Partial Fraction Expansion

`residue` finds the partial fraction expansion of the ratio of two polynomials. This is particularly useful for applications that represent systems in transfer function form. For polynomials b and a , if there are no multiple roots,

b
F
s
f
r r r

$$=^1 + ^2 + ^4 + ^n + k_{sa} \int s - p_1 s - p_2 s - p_n$$

where *r* is a column vector of residues, *p* is a column vector of pole locations, and *k* is a row vector of direct terms. Consider the transfer function

$$-4s + 8$$

$$s^2 + 6s + 8$$

```
b = [-4 8];  
a = [1 6 8];  
[r,p,k] = residue(b,a)
```

```
r =  
-12  
8
```

```
p =  
-4  
-2
```

k = []

Given three input arguments (*r*,*p*, and *k*), *residue* converts back to polynomial form.

```
[b2,a2] = residue(r,p,k)
```

```
b2 =  
-4 8  
a2 = 1 6 8
```

Interpolation

Interpolation is a process for estimating values that lie between known data points. It has important applications in areas such as signal and image processing.

This section:

- Provides a summary of the MATLAB interpolation functions
- Discusses one-dimensional interpolation
- Discusses two-dimensional interpolation
- Uses an example to compare nearest neighbor, bilinear, and bicubic interpolation methods
- Discusses interpolation of multidimensional data
- Discusses triangulation and interpolation of scattered data

Interpolation Function Summary

MATLAB provides a number of interpolation techniques that let you balance the smoothness of the data fit with speed of execution and memory usage. The interpolation functions reside in the

MATLABpolyfun directory.

Interpolation Function Summary Function Description griddata Data gridding and surface fitting. griddata3 Data gridding and hypersurface fitting for three-dimensional data.

griddatan

interp1

interp2

interp3

interpft

Data gridding and hypersurface fitting (dimension ≥ 3). One-dimensional interpolation (table lookup). Two-dimensional interpolation (table lookup). Three-dimensional interpolation (table lookup). One-dimensional interpolation using FFT method.

Interpolation Function Summary (Continued) Function Description

interpN N-D interpolation (table lookup). mkpp

pchip

ppval

spline

unmkpp

Make a piecewise polynomial

Piecewise Cubic Hermite Interpolating Polynomial (PCHIP).

Piecewise polynomial evaluation

Cubic spline data interpolation

Piecewise polynomial details

One-Dimensional Interpolation

There are two kinds of one-dimensional interpolation in MATLAB:

- Polynomial interpolation
- FFT-based interpolation

Polynomial Interpolation

The function interp1 performs one-dimensional interpolation, an important operation for data analysis and curve fitting. This function uses polynomial techniques, fitting the supplied data with polynomial functions between data points and evaluating the appropriate function at the desired interpolation points. Its most general form is

$yi = \text{interp1}(x, y, xi, \text{method})$

y is a vector containing the values of a function, and x is a vector of the same length containing the points for which the values in y are given. xi is a vector containing the points at which to interpolate. method is an optional string specifying an interpolation method:

- *Nearest neighbor interpolation* ($\text{method} = \text{'nearest'}$). This method sets the value of an interpolated point to the value of the nearest existing data point.
- *Linear interpolation* ($\text{method} = \text{'linear'}$). This method fits a different linear function between each pair of existing data points, and returns the value of the relevant function at the points

specified byxi. This is the default method for the interp1 function.

- *Cubic spline interpolation* (method = 'spline'). This method fits a different cubic function between each pair of existing data points, and uses the spline function to perform cubic spline interpolation at the data points.
- *Cubic interpolation* (method = 'pchip' or 'cubic'). These methods are identical. They use the pchip function to perform piecewise cubic Hermite interpolation within the vectors x and y. These methods preserve monotonicity and the shape of the data.

If any element of xi is outside the interval spanned by x, the specified interpolation method is used for extrapolation. Alternatively,
yi = interp1(x,Y,xi,method,extrapval) replaces extrapolated values with extrapval. NaN is often used for extrapval.

All methods work with nonuniformly spaced data.

Speed, Memory, and Smoothness Considerations

When choosing an interpolation method, keep in mind that some require more memory or longer computation time than others. However, you may need to trade off these resources to achieve the desired smoothness in the result.

- Nearest neighbor interpolation is the fastest method. However, it provides the worst results in terms of smoothness.
- Linear interpolation uses more memory than the nearest neighbor method, and requires slightly more execution time. Unlike nearest neighbor interpolation its results are continuous, but the slope changes at the vertex points.
- Cubic spline interpolation has the longest relative execution time, although it requires less memory than cubic interpolation. It produces the smoothest results of all the interpolation methods. You may obtain unexpected results, however, if your input data is non-uniform and some points are much closer together than others.
- Cubic interpolation requires more memory and execution time than either the nearest neighbor or linear methods. However, both the interpolated data and its derivative are continuous.

The relative performance of each method holds true even for interpolation of two-dimensional or multidimensional data. For a graphical comparison of interpolation methods, see the section “Comparing Interpolation Methods” on page 12-14.

FFT-Based Interpolation

The function interpft performs one-dimensional interpolation using an FFT-based method. This method calculates the Fourier transform of a vector that contains the values of a periodic function. It then calculates the inverse Fourier transform using more points. Its form is

y = interpft(x,n)

x is a vector containing the values of a periodic function, sampled at equally spaced points. n is the number of equally spaced points to return.

Two-Dimensional Interpolation

The function interp2 performs two-dimensional interpolation, an important operation for image

processing and data visualization. Its most general form is $ZI = \text{interp2}(X, Y, Z, XI, YI, \text{method})$

Z is a rectangular array containing the values of a two-dimensional function, and X and Y are arrays of the same size containing the points for which the values in Z are given. XI and YI are matrices containing the points at which to interpolate the data. method is an optional string specifying an interpolation method.

There are three different interpolation methods for two-dimensional data:

- *Nearest neighbor interpolation* ($\text{method} = \text{'nearest'}$). This method fits a piecewise constant surface through the data values. The value of an interpolated point is the value of the nearest point.
- *Bilinear interpolation* ($\text{method} = \text{'linear'}$). This method fits a bilinear surface through existing data points. The value of an interpolated point is a combination of the values of the four closest points. This method is piecewise bilinear, and is faster and less memory-intensive than bicubic interpolation.
- *Bicubic interpolation* ($\text{method} = \text{'cubic'}$). This method fits a bicubic surface through existing data points. The value of an interpolated point is a combination of the values of the sixteen closest points. This method is piecewise bicubic, and produces a much smoother surface than bilinear interpolation. This can be a key advantage for applications like image processing. Use bicubic interpolation when the interpolated data and its derivative must be continuous.

All of these methods require that X and Y be monotonic, that is, either always increasing or always decreasing from point to point. You should prepare these matrices using the `meshgrid` function, or else be sure that the “pattern” of the points emulates the output of `meshgrid`. In addition, each method automatically maps the input to an equally spaced domain before interpolating. If X and Y are already equally spaced, you can speed execution time by prepending an asterisk to the `method` string, for example, `*cubic`.

Comparing Interpolation Methods

This example compares two-dimensional interpolation methods on a 7-by-7 matrix of data.

1 Generate the peaks function at low resolution.

```
[x,y] = meshgrid(-3:1:3);
z = peaks(x,y);
surf(x,y,z)
```

```
6
4
2
0
-2
-4
-6 3
231 2
```

```
0 1 -1^0
-1 -2 -2
```

-3 -3

- 2 Generate a finer mesh for interpolation. $[xi, yi] = \text{meshgrid}(-3:0.25:3);$
- 3 Interpolate using nearest neighbor interpolation. $zi1 = \text{interp2}(x, y, z, xi, yi, \text{'nearest'});$
- 4 Interpolate using bilinear interpolation: $zi2 = \text{interp2}(x, y, z, xi, yi, \text{'bilinear'});$
- 5 Interpolate using bicubic interpolation. $zi3 = \text{interp2}(x, y, z, xi, yi, \text{'bicubic'});$
- 6 Compare the surface plots for the different interpolation methods.

-2

-4

-6

6

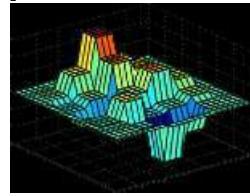
4

2

0

3

2



$^1 2 0 1 -1 0$

-1

-2 -2

-3 -3

6

4

2

0

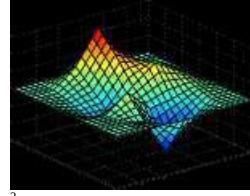
-2

-4

-6

3

2



$^1 2 0 1 -1 0$

-1

-2 -2

-3 -3

-2

-4

-6

3

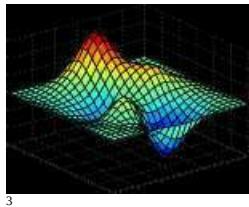
6

4

2

0

2



3

1 2 0 1 -1 0

-1
-2 -2
-3 -3

```
surf(xi,yi,zi1) % nearest
surf(xi,yi,zi2) % bilinear
surf(xi,yi,zi3) % bicubic
```

7 Compare the contour plots for the different interpolation methods.

```
3 3 3
2 2 2
1 1 1
0 0 0
-1 -1 -1
-2 -2 -2
-3 -3 -2 -1 0 1 2 3 -3
-3 -2 -1 0 1 2 3 -3
-3 -2 -1 0 1 2 3
```

```
contour(xi,yi,zi1) % nearest
contour(xi,yi,zi2) % bilinear
contour(xi,yi,zi3) % bicubic
```

Notice that the bicubic method, in particular, produces smoother contours. This is not always the primary concern, however. For some applications, such as medical image processing, a method like nearest neighbor may be preferred because it doesn't generate any "new" data values.

Interpolation and Multidimensional Arrays

Several interpolation functions operate specifically on multidimensional data.

Interpolation Functions for Multidimensional Data Function Description

interp3
interpн
ndgrid

Three-dimensional data interpolation.

Multidimensional data interpolation.

Multidimensional data gridding (ndfun directory).

This section discusses:

Interpolation of three-dimensional data•

- Interpolation of higher dimensional data
- Multidimensional data gridding

Interpolation of Three-Dimensional Data

The function `interp3` performs three-dimensional interpolation, finding interpolated values between points of a three-dimensional set of samples V . You must specify a set of known data points:

- X , Y , and Z matrices specify the points for which values of V are given.
- A matrix V contains values corresponding to the points in X , Y , and Z .

The most general form for `interp3` is

$VI = \text{interp3}(X, Y, Z, V, XI, YI, ZI, \text{method})$

XI , YI , and ZI are the points at which `interp3` interpolates values of V . For out-of-range values, `interp3` returns `NaN`.

There are three different interpolation methods for three-dimensional data:

- *Nearest neighbor interpolation* (`method = 'nearest'`). This method chooses the value of the nearest point.
- *Trilinear interpolation* (`method = 'linear'`). This method uses piecewise linear interpolation based on the values of the nearest eight points.
- *Tricubic interpolation* (`method = 'cubic'`). This method uses piecewise cubic interpolation based on the values of the nearest sixty-four points.

All of these methods require that X , Y , and Z be *monotonic*, that is, either always increasing or always decreasing in a particular direction. In addition, you should prepare these matrices using the `meshgrid` function, or else be sure that the “pattern” of the points emulates the output of `meshgrid`.

Each method automatically maps the input to an equally spaced domain before interpolating. If x is already equally spaced, you can speed execution time by prepending an asterisk to the `method` string, for example, `*cubic`.

Interpolation of Higher Dimensional Data

The function `interpN` performs multidimensional interpolation, finding interpolated values between points of a multidimensional set of samples V . The most general form for `interpN` is

$VI = \text{interpN}(X1, X2, X3, \dots, V, Y1, Y2, Y3, \dots, \text{method})$

$1, 2, 3, \dots$ are matrices that specify the points for which values of V are given.

V is a matrix that contains the values corresponding to these points. $1, 2, 3, \dots$ are the points for which `interpN` returns interpolated values of V . For out-of-range values, `interpN` returns `NaN`.

$Y1, Y2, Y3, \dots$ must be either arrays of the same size, or vectors. If they are vectors of different sizes, `interpN` passes them to `ndgrid` and then uses the resulting arrays.

There are three different interpolation methods for multidimensional data: (`method = 'nearest'`). This method chooses

- *Nearest neighbor interpolation* the value of the nearest point.

- *Linear interpolation* (`method = 'linear'`). This method uses piecewise linear interpolation based

on the values of the nearest two points in each dimension.

- *Cubic interpolation* (method = 'cubic'). This method uses piecewise cubic interpolation based on the values of the nearest four points in each dimension.

All of these methods require that X1, X2, X3 be monotonic. In addition, you should prepare these matrices using the ndgrid function, or else be sure that the “pattern” of the points emulates the output of ndgrid.

Each method automatically maps the input to an equally spaced domain before interpolating. If X is already equally spaced, you can speed execution time by prepending an asterisk to the method string; for example, '*cubic'.

Multidimensional Data Gridding

The ndgrid function generates arrays of data for multidimensional function evaluation and interpolation. ndgrid transforms the domain specified by a series of input vectors into a series of output arrays. The i-th dimension of these output arrays are copies of the elements of input vector x_i .

The syntax for ndgrid is

$$[X_1, X_2, X_3, \dots] = \text{ndgrid}(x_1, x_2, x_3, \dots)$$

For example, assume that you want to evaluate a function of three variables over a given range. Consider the function

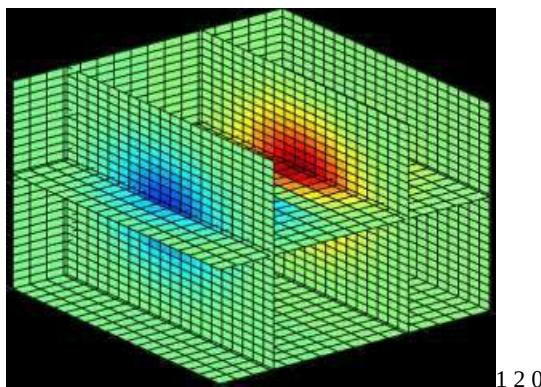
$$f = -x_1^2 - x_2^2 - x_3^2$$

$$z = x_2 e^{x_1^2 + x_2^2 + x_3^2}$$

for $-2 \leq x_1 \leq 0$, $2 \leq x_2 \leq 4$, and $0 \leq x_3 \leq 2$. To evaluate and plot this function:

```
x1 = -2:0.2:2;
x2 = -2:0.25:2;
x3 = -2:0.16:2;
[X1,X2,X3] = ndgrid(x1,x2,x3);
z = X2.*exp(-X1.^2 -X2.^2 -X3.^2);
slice(X2,X1,X3,z,[-1.2 8 2],2,[ -2 0.2])
```

2
1
0
-1
-2
2



1
0-1 -1
-2 -2

Triangulation and Interpolation of Scattered Data

MATLAB provides routines that aid in the analysis of closest-point problems and geometric analysis.

Functions for Analysis of Closest-Point Problems and Geometric Analysis Function

Description convhull Convex hull. delaunay Delaunay triangulation.

Functions for Analysis of Closest-Point Problems and Geometric Analysis Function

Description delaunay3 3-D Delaunay tessellation. dsearch Nearest point search of Delaunay triangulation. inpolygon True for points inside polygonal region. polyarea Area of polygon. rectint Area of intersection for two or more rectangles. tsearch Closest triangle search. voronoi Voronoi diagram.

This section applies the following techniques to theseamount data set supplied with MATLAB:

- Convex hulls
- Delaunay triangulation
- Voronoi diagrams

See also “Tessellation and Interpolation of Scattered Data in Higher Dimensions” on page 12-27.

Note Examples in this section use the MATLABseamount data set. Seamounts are underwater mountains. They are valuable sources of information about marine geology. The seamount data set represents the surface, in 1984, of the seamount designated LR148.8W located at 48.2°S, 148.8°W on the Louisville Ridge in the South Pacific. For more information about the data and its use, see Parker [2].

Theseamount data set provides longitude (x), latitude (y) and depth-in-feet (z) data for 294 points on the seamount LR148.8W.

Convex Hulls

The convhull function returns the indices of the points in a data set that comprise the convex hull for the set. Use theplot function to plot the output of convhull.

This example loads the seamount data and plots the longitudinal (x) and latitudinal (y) data as a scatter plot. It then generates the convex hull and uses plot to plot the convex hull.

```
load seamount
```

```

plot(x,y,'','markersize',10)
k = convhull(x,y);
hold on, plot(x(k),y(k),'-r'), hold off
grid on

-47.95
-48
-48.05
-48.1
-48.15
-48.2
-48.25
-48.3
-48.35
-48.4
-48.45
210.8 211 211.2 211.4 211.6 211.8

```

Delaunay Triangulation

Given a set of coplanar data points, *Delaunay triangulation* is a set of lines connecting each point to its natural neighbors. The delaunay function returns

a Delaunay triangulation as a set of triangles such that no data points are contained in any triangle's circumcircle.

You can use triplot to print the resulting triangles in two-dimensional space. You can also add data for a third dimension to the output of delaunay and plot the result as a surface with trisurf, or as a mesh with trimesh.

Plotting a Delaunay Triangulation. To try delaunay, load the seamount data set and view the longitude (x) and latitude (y) data as a scatter plot.

```

load seamount
plot(x,y,'','markersize',12)
xlabel('Longitude'), ylabel('Latitude')
grid on

```

```

-47.95
-48
-48.05
-48.1
-48.15
-48.2
-48.25
-48.3
-48.35
-48.4
-48.45
210.8 211 211.2 211.4 211.6 211.8 Longitude

```

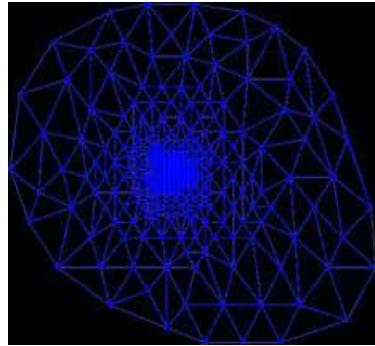
Apply Delaunay triangulation and use triplot to overplot the resulting triangles on the scatter plot.

```

tri = delaunay(x,y);
hold on, triplot(tri,x,y), hold off
-47.95
-48

```

```
-48.05  
-48.1  
-48.15  
-48.2  
-48.25  
-48.3  
-48.35  
-48.4
```

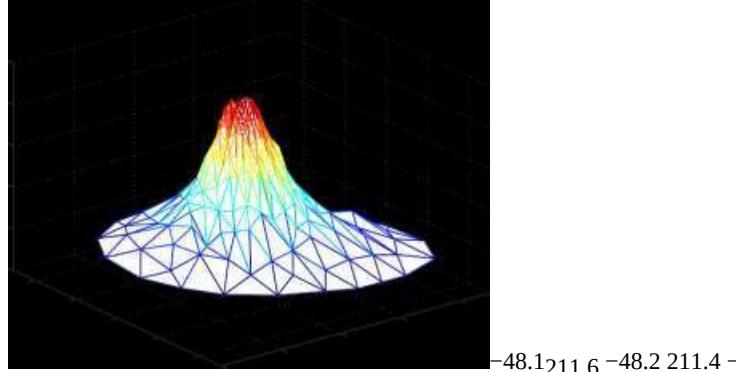


```
- 48.45  
210.8 211 211.2 211.4 211.6 211.8  
Longitude
```

Mesh and Surface Plots. Add the depth data (z) from seamount, to the Delaunay triangulation, and usetrimesh to produce a mesh in three-dimensional space. Similarly, you can use trisurf to produce a surface.

```
figure  
hidden on  
trimesh(tri,x,y,z)  
grid on  
xlabel('Longitude'); ylabel('Latitude'); zlabel('Depth in Feet')
```

```
-1000  
-2000  
-3000  
-4000  
  
- 5000  
-48 0
```



```
48.3  
211.2
```

```
-48.1 211.6 -48.2 211.4 -
```

211-48.4
Latitude 210.8 Longitude

Contour Plots. This code uses meshgrid, griddata, and contour to produce a contour plot of theseamount data.

```
figure  
[xi,yi] = meshgrid(210.8:01:211.8,-48.5:01:-47.9); zi = griddata(x,y,z,xi,yi,'cubic');  
[c,h] = contour(xi,yi,zi,'b-');  
clabel(c,h)
```

-47.95
-48
-48.05
-48.1
-48.15

-48.2
-48.25
-48.3
-48.35
-48.4
-
48.45
210.8 211 211.2 211.4 211.6 211.8 Longitude

The arguments for meshgrid encompass the largest and smallest x and y values in the originalseamount data. To obtain these values, use min(x), max(x), min(y), and max(y).

Closest-Point Searches. You can search through the Delaunay triangulation data with two functions:

- dsearch finds the indices of the (x,y) points in a Delaunay triangulation closest to the points you specify. This code searches for the point closest to (211.32, -48.35) in the triangulation of theseamount data.

```
xi = 211.32; yi = -48.35;  
p = dsearch(x,y,tri,xi,yi);  
[x(p), y(p)]
```

```
ans =  
211.3400 -48.3700
```

- tsearch, finds the indices into the delaunay output that specify the enclosing triangles of the points you specify. This example uses the index of the enclosing triangle for the point (211.32, -48.35) to obtain the coordinates of the vertices of the triangle.

```
xi = 211.32; yi = -48.35; t = tsearch(x,y,tri,xi,yi); r = tri(t,:);  
A = [x(r) y(r)]
```

```
A =  
211.3000 -48.3000  
211.3400 -48.3700
```

211.2800 -48.3200

Voronoi Diagrams

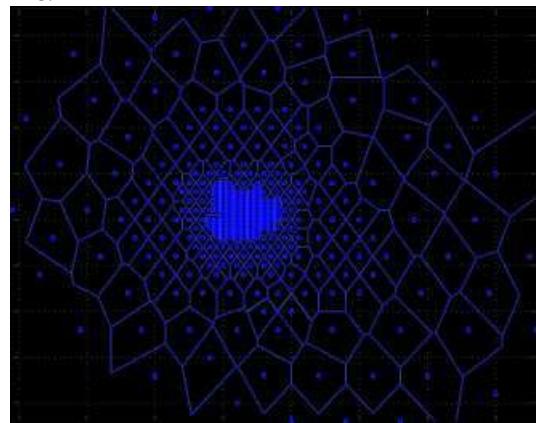
Voronoi diagrams are a closest-point plotting technique related to Delaunay triangulation.

For each point in a set of coplanar points, you can draw a polygon that encloses all the intermediate points that are closer to that point than to any other point in the set. Such a polygon is called a *Voronoi polygon*, and the set of all Voronoi polygons for a given point set is called a *Voronoi diagram*.

The voronoi function can plot the cells of the Voronoi diagram, or return the vertices of the edges of the diagram. This example loads the seamount data, then uses the voronoi function to produce the Voronoi diagram for the longitudinal (x) and latitudinal (y) dimensions. Note that voronoi plots only the bounded cells of the Voronoi diagram.

```
load seamount  
voronoi(x,y)  
grid on
```

-48
-48.05
-48.1
-48.15
-48.2
-48.25
-48.3
-48.35
-48.4



210.9 211 211.1 211.2 211.3 211.4 211.5 211.6

Longitude

Note See the voronoi function for an example that uses the vertices of the edges to plot a Voronoi diagram.

Tessellation and Interpolation of Scattered Data in Higher Dimensions

Many applications in science, engineering, statistics, and mathematics require structures like convex hulls, Voronoi diagrams, and Delaunay tessellations. Using Qhull [1], MATLAB functions enable you to geometrically analyze data sets in any dimension.

Functions for Multidimensional Geometrical Analysis Function Description

convhulln n-D convex hull. delaunayn n-D Delaunay tessellation. dsearchn n-D nearest point search. griddatan n-D data gridding and hypersurface fitting. tsearchn n-D closest simplex search. voronoin n-D Voronoi diagrams.

This section demonstrates these geometric analysis techniques:

- Convex hulls
- Delaunay triangulations
- Voronoi diagrams
- Interpolation of scattered multidimensional data

Convex Hulls

The convex hull of a data set in n-dimensional space is defined as the smallest convex region that contains the data set.

Computing a Convex Hull. The convhulln function returns the indices of the points in a data set that comprise the facets of the convex hull for the set. For example, suppose X is an 8-by-3 matrix that consists of the 8 vertices of a cube. The convex hull of X then consists of 12 facets.

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d);
X = [x(:),y(:),z(:)]; % 8 corner points of a cube
C = convhulln(X)
```

```
C =
3 1 5
1 2 5
2 1 3
7 3 5
8 7 5
7 8 3
6 8 5
2 6 5
6 2 8
8 4 3
4 2 3
2 4 8
```

Because the data is three-dimensional, the facets that make up the convex hull are triangles. The 12 rows of C represent 12 triangles. The elements of C are indices of points in X. For example, the first row, 3 1 5, means that the first triangle has X(3,:), X(1,:), and X(5,:) as its vertices.

For three-dimensional convex hulls, you can use trisurf to plot the output. However, using patch to plot the output gives you more control over the color of the facets. Note that you cannot plot convhulln output for n > 3.

This code plots the convex hull by drawing the triangles as three-dimensional patches.

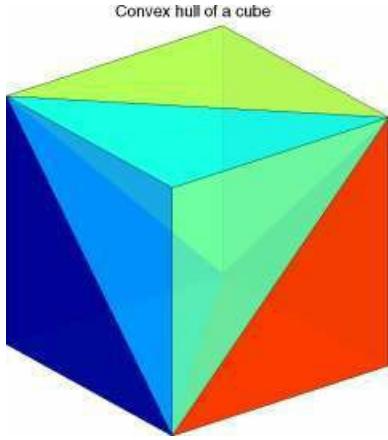
```
figure, hold on
d = [1 2 3 1 % Index into C column.
```

```

for i = 1:size(C,1) % Draw each triangle.

j= C(i,d); % Get the ith C to make a patch. h(i)=patch(X(j,1),X(j,2),X(j,3),i,'FaceAlpha',0.9); end
% 'FaceAlpha' is used to make it transparent.
hold off
view(3), axis equal, axis off
camorbit(90,-5); % To view it from another angle
title('Convex hull of a cube')

```



Delaunay Tessellations

A Delaunay tessellation is a set of simplices such that no data points are contained in any simplex's circumsphere. In two-dimensional space, a simplex is a triangle. In three-dimensional space, a simplex is a tetrahedron.

Computing a Delaunay Tessellation. Thedelaunay function returns the indices of the points in a data set that comprise the simplices of an n-dimensional Delaunay tessellation of the data set.

This example uses the same X as in the convex hull example, i.e. the 8 corner points of a cube, with the addition of a center point.

```

d = [-1 1];
[x,y,z] = meshgrid(d,d,d);
X = [x(:),y(:),z(:)]; % 8 corner points of a cube X(9,:) = [0 0 0]; % Add center to the vertex list. T
= delaunay(X) % Generate Delaunay tessellation.

```

```

T =
9 1 5 6
3 9 1 5
2 9 1 6
2 3 9 4
2 3 9 1
7 9 5 6
7 3 9 5
8 7 9 6
8 2 9 6

```

```
8 2 9 4  
8 3 9 4  
8 7 3 9
```

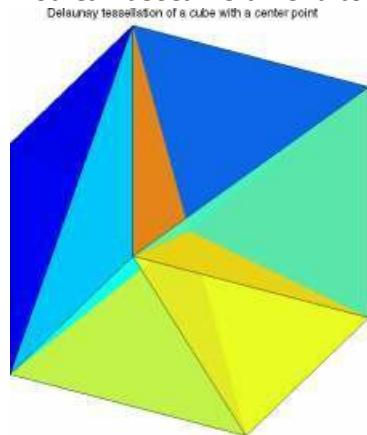
The 12 rows of T represent the 12 simplices, in this case irregular tetrahedrons, that partition the cube. Each row represents one tetrahedron, and the row elements are indices of points in X.

For three-dimensional tessellations, you can use tetramesh to plot the output. However, using patch to plot the output gives you more control over the color of the facets. Note that you cannot plot delaunay output for n > 3.

This code plots the tessellation T by drawing the tetrahedrons using three-dimensional patches.

```
figure, hold on  
d = [1 1 1 2; 2 2 3 3; 3 4 4 4]; % Index into T  
for i = 1:size(T,1) % Draw each tetrahedron.  
  
y = T(i,d); % Get the ith T to make a patch. x1 = reshape(X(y,1),3,4);  
x2 = reshape(X(y,2),3,4);  
x3 = reshape(X(y,3),3,4);  
h(i)=patch(x1,x2,x3,(1:4)*i,'FaceAlpha',0.9);  
  
end  
hold off  
view(3), axis equal  
axis off  
camorbit(65,120) % To view it from another angle title('Delaunay tessellation of a cube with a center point')
```

You can use camera menu to rotate the figure in any direction.



Voronoi Diagrams

Given m data points in n -dimensional space, a *Voronoi diagram* is the partition of n -dimensional space into m polyhedral regions, one region for each data point. Such a region is called a *Voronoi cell*. A Voronoi cell satisfies the condition that it contains all points that are closer to its data point than any other data point in the set.

Computing a Voronoi Diagram. The voronoin function returns two outputs:

- V is an m -by- n matrix of m points in n -space. Each row of V represents a Voronoi vertex.
 - C is a cell array of vectors. Each vector in the cell array C represents a Voronoi cell. The vector contains indices of the points in V that are the vertices of the Voronoi cell. Each Voronoi cell may have a different number of points.
- Because a Voronoi cell can be unbounded, the first row of V is a point at infinity. Then any unbounded Voronoi cell in C includes the point at infinity, i.e., the first point in V.
- This example uses the same X as in the Delaunay example, i.e., the 8 corner points of a cube and its center. Random noise is added to make the cube less regular. The resulting Voronoi diagram has 9 Voronoi cells.

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d);
X = [x(:),y(:),z(:)]; % 8 corner points of a cube X(9,:) = [0 0 0]; % Add center to the vertex list.
X = X+0.01*rand(size(X)); % Make the cube less regular. [V,C] = voronoin(X);

V = Inf Inf Inf 0.0055 1.5054 0.0004 0.0037 0.0101 -1.4990 0.0052 0.0087 -1.4990 0.0030
1.5054 0.0030 0.0072 0.0072 1.4971

-1.7912 0.0000 0.0044
-1.4886 0.0011 0.0036
-1.4886 0.0002 0.0045 0.0101 0.0044 1.4971 1.5115 0.0074 0.0033 1.5115 0.0081 0.0040
0.0104 -1.4846 -0.0007 0.0026 -1.4846 0.0071
```

```
C = [1x8 double] [1x6 double] [1x4 double] [1x6 double] [1x6 double] [1x6 double] [1x6
double] [1x6 double] [1x12 double]
```

In this example, V is a 13-by-3 matrix, the 13 rows are the coordinates of the 13 Voronoi vertices. The first row of V is a point at infinity. C is a 9-by-1 cell array, where each cell in the array contains an index vector into V corresponding to one of the 9 Voronoi cells. For example, the 9th cell of the Voronoi diagram is

```
C{9} = 2 3 4 5 6 7 8 9 10 11 12 13
```

If any index in a cell of the cell array is 1, then the corresponding Voronoi cell contains the first point in V, a point at infinity. This means the Voronoi cell is unbounded.

To view a *bounded* Voronoi cell, i.e., one that does not contain a point at infinity, use the convhulln function to compute the vertices of the facets that make up the Voronoi cell. Then use patch and other plot functions to generate the figure. For example, this code plots the Voronoi cell defined by the 9th cell in C.

```
X = V(C{9},:);
K = convhulln(X); figure
hold on
```

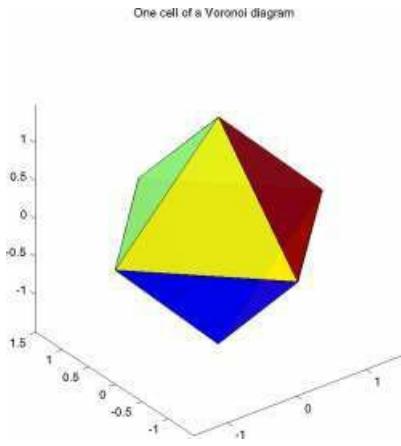
```

d = [1 2 3 1];
for i = 1:size(K,1)

j = K(i,d);
% View 9th Voronoi cell.
% Index into K

h(i)=patch(X(j,1),X(j,2),X(j,3),i,'FaceAlpha',0.9); end
hold off
view(3)
axis equal
title('One cell of a Voronoi diagram')

```



Interpolating N-Dimensional Data

Use the griddatan function to interpolate multidimensional data, particularly scattered data. griddatan uses the delaunayn function to tessellate the data, and then interpolates based on the tessellation.

Suppose you want to visualize a function that you have evaluated at a set of n scattered points. In this example, X is an n-by-3 matrix of points, each row containing the (x,y,z) coordinates for one of the points. The vector v contains the n function values at these points. The function for this example is the squared distance from the origin, $v = x.^2 + y.^2 + z.^2$.

Start by generating n = 5000 points at random in three-dimensional space, and computing the value of a function on those points.

```

n = 5000;
X = 2*rand(n,3)-1;
v = sum(X.^2,2);

```

The next step is to use interpolation to compute function values over a grid. Use meshgrid to create the grid, and griddatan to do the interpolation.

```

delta = 0.05;
d = -1:delta:1;

```

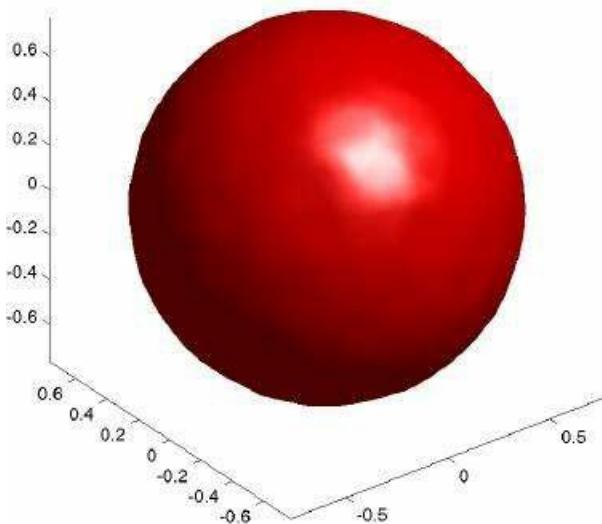
```
[x0,y0,z0] = meshgrid(d,d,d);
X0 = [x0(:), y0(:), z0(:)];
v0 = griddatan(X,v,X0);
v0 = reshape(v0, size(x0));
```

Then use isosurface and related functions to visualize the surface that consists of the (x,y,z) values for which the function takes a constant value. You could pick any value, but the example uses the value 0.6. Since the function is the squared distance from the origin, the surface at a constant value is a sphere.

```
p = patch(isosurface(x0,y0,z0,v0,0.6));
isonormals(x0,y0,z0,v0,p);
set(p,'FaceColor','red','EdgeColor','none');
view(3);
camlight;
lighting phong
axis equal
title('Interpolated sphere from scattered data')
```

Note A smaller delta produces a smoother sphere, but increases the compute time.

Interpolated sphere from scattered data



Selected Bibliography

- [1] National Science and Technology Research Center for Computation and Visualization of Geometric Structures (The Geometry Center), University of Minnesota. 1993. For information about qhull, see
<http://www.geom.umn.edu/software/qhull/>.

- [2] Parker, Robert. L., Loren Shure, & John A. Hildebrand, "The Application of Inverse Theory to Seamount Magnetism." *Reviews of Geophysics*. Vol. 25, No. 1, 1987.

Data Analysis and Statistics

Column-Oriented Data Sets.....	13-4
Basic Data Analysis Functions	13-8
Function Summary	13-8
Covariance and Correlation Coefficients.....	13-11
Finite Differences.....	13-12
Data Preprocessing	13-14
Missing Values.....	13-14
Removing Outliers.....	13-15
Regression and Curve Fitting.....	13-17
Polynomial Regression	13-18
Linear-in-the-Parameters Regression.....	13-19
Multiple Regression	13-21
Case Study: Curve Fitting	13-22
Polynomial Fit.....	13-22
Analyzing Residuals.....	13-24
Exponential Fit.....	13-27
Error Bounds.....	13-30
The Basic Fitting Interface	13-31
Difference Equations and Filtering	13-40

Fourier Analysis and the Fast Fourier

Transform (FFT)	13-43
Function Summary	13-43
Introduction.....	13-44
Magnitude and Phase of Transformed Data	13-49
FFT Length Versus Speed.....	13-50

This chapter introduces MATLAB's data analysis capabilities. It discusses how to organize arrays for data analysis, how to use simple descriptive statistics functions, and how to perform data preprocessing tasks in MATLAB. It also discusses other data analysis topics, including regression, curve fitting, data filtering, and fast Fourier transforms (FFTs). It includes:

Column-Oriented Data Sets Organizing arrays for data analysis.

Basic Data Analysis Functions

Basic data analysis functions and an example that uses some of the functions. This section also discusses functions for the computation of correlation coefficients and covariance, and for finite difference calculations.

Data Preprocessing

Working with missing values, and outliers or misplaced data points in a data set.

Regression and Curve Fitting

Investigates the use of different regression methods to find functions that describe the relationship among observed variables.

Case Study: Curve Fitting

Uses a case study to look at some of MATLAB's basic data analysis capabilities. This section also provides information about the Basic Fitting interface.

Difference Equations and Filtering

Discusses MATLAB functions for working with difference equations and filters.

Fourier Analysis and the Fast Fourier Transform (FFT)

Discusses Fourier analysis in MATLAB

Data Analysis and Statistics Functions

The data analysis and statistics functions are in the directory `datafun` in the MATLAB Toolbox. Use online help to get a complete list of functions.

Related Toolboxes

A number of related toolboxes provide advanced functionality for specialized data analysis applications.

Toolbox Data Analysis Application

Optimization Nonlinear curve fitting and regression. Signal Processing Signal processing, filtering, and frequency

analysis.

Spline Statistics

System Identification Wavelet Curve fitting and regression.

Advanced statistical analysis, nonlinear curve fitting, and regression.

Parametric / ARMA modeling.

Wavelet analysis.

Column-Oriented Data Sets

Univariate statistical data is typically stored in individual vectors. The vectors can be either 1-by- n or n -by-1. For multivariate data, a matrix is the natural representation but there are, in principle, two possibilities for orientation. By MATLAB convention, however, the different variables are put into columns, allowing observations to vary down through the rows. Therefore, a data set consisting of twenty four samples of three variables is stored in a matrix of size 24-by-3.

Vehicle Traffic Sample Data Set

Consider a sample data set comprising vehicle traffic count observations at three locations over a 24-hour period.

Vehicle Traffic Sample Data Set Time Location 1 Location 2 Location 3

01h00 11 11 9

02h00 7 13 11 03h00 14 17 20 04h00 11 13 9 05h00 43 51 69 06h00 38 46 76 07h00 61 132 186
08h00 75 135 180 09h00 38 88 115 10h00 28 36 55 11h00 12 12 14 12h00 18 27 30 13h00 18 19
29

Vehicle Traffic Sample Data Set (Continued) Time Location 1 Location 2 Location 3 14h00
17 15 18 15h00 19 36 48 16h00 32 47 10 17h00 42 65 92 18h00 57 66 151 19h00 44 55 90
20h00 114 145 257 21h00 35 58 68 22h00 11 12 15 23h00 13 9 15 24h00 10 9 7

Loading and Plotting the Data

The raw data is stored in the file, count.dat.

```
11 11 9
7 13 11
14 17 20
11 13 9
43 51 69
38 46 76
61 132 186
75 135 180
38 88 115
28 36 55
12 12 14
18 27 30
18 19 29
17 15 18
19 36 48
32 47 10
42 65 92
57 66 151
44 55 90
114 145 257
35 58 68
11 12 15
13 9 15
10 9 7
```

Use the load command to import the data.

```
load count.dat
```

This creates the matrix count in the workspace.

For this example, there are 24 observations of three variables. This is confirmed by

```
[n,p] = size(count)
```

```
n =
```

```
24
```

```
p =
```

```
3
```

Create a time vector, t, of integers from 1 ton.

t = 1:n;

Now plot the counts versus time and annotate the plot.

```
set(0,'defaultaxeslinestyleorder','--')
set(0,'defaultaxescolororder',[0 0 0])
plot(t,count), legend('Location 1','Location 2','Location 3',2) xlabel('Time'), ylabel('Vehicle Count'), grid on
```

The plot shows the vehicle counts at three locations over a 24-hour period.

300

Location 1
Location 2
Location 3 250

200
150
100
50
0 5 10 15 20 25 Time

Basic Data Analysis Functions

This section introduces functions for:

- Basic column-oriented data analysis
- Computation of correlation coefficients and covariance
- Calculating finite differences

Function Summary

A collection of functions provides basic column-oriented data analysis capabilities. These functions are located in the MATLAB datafun directory. This section also gives you some hints about using row and column data, and provides some basic examples. This table lists the functions.

Basic Data Analysis Function Summary

cumprod Cumulative product of elements. cumsum

cumtrapz

diff

max

mean

median

min

prod

sort

sortrows

std

Cumulative sum of elements.

Cumulative trapezoidal numerical integration. Difference function and approximate derivative.

Largest component.

Average or mean value.
Median value.
Smallest component.
Product of elements.
Sort in ascending order.
Sort rows in ascending order.
Standard deviation.

Basic Data Analysis Function Summary (Continued) Function Description sum Sum of elements. trapz Trapezoidal numerical integration.

For information about calculating the maximum, minimum, mean, median, range, and standard deviation on plotted data, and creating plots of these statistics, see “Adding Plots of Data Statistics to a Graph” in the MATLAB graphics documentation.

Working with Row and Column Data

For vector input arguments to these functions, it does not matter whether the vectors are oriented in row or column direction. For array arguments, however, the functions operate column by column on the data in the array. This means, for example, that if you apply max to an array, the result is a row vector containing the maximum values over each column.

Note You can add more functions to this list using M-files, but when doing so, you must exercise care to handle the row-vector case. If you are writing your own column-oriented M-files, check other M-files; for example, mean.m and diff.m.

Basic Examples

Continuing with the vehicle traffic count example, the statements

```
mx = max(count)
mu = mean(count)
sigma = std(count)
```

result in

```
mx =
114 145 257
mu =
32.0000 46.5417 65.5833
sigma =
25.3703 41.4057 68.0281
```

To locate the index at which the minimum or maximum occurs, a second output parameter can be specified. For example,

```
[mx,indx] = min(count)
```

```
mx =
7 9 7
indx = 2 23 24
```

shows that the lowest vehicle count is recorded at 02h00 for the first observation point (column one) and at 23h00 and 24h00 for the other observation points.

You can subtract the mean from each column of the data using an outer product involving a vector of n ones.

```
[n,p] = size(count)
e = ones(n,1)
x = count - e*mu
```

Rearranging the data may help you evaluate a vector function over an entire data set. For example, to find the smallest value in the entire data set, use

```
min(count(:))
```

which produces

```
ans =
```

7 The syntax `count(:)` rearranges the 24-by-3 matrix into a 72-by-1 column vector.

Covariance and Correlation Coefficients

MATLAB's statistical capabilities include two functions for the computation of correlation coefficients and covariance.

Covariance and Correlation Coefficient Function Summary Function Description

```
cov
```

```
corrcoef
```

Variance of vector – measure of spread or dispersion of sample variable.

Covariance of matrix – measure of strength of linear relationships between variables.

Correlation coefficient – normalized measure of linear relationship strength between variables.

Covariance `cov` returns the variance for a vector of data. The variance of the data in the first column of `count` is

```
cov(count(:,1))
```

```
ans = 643.6522
```

For an array of data, `cov` calculates the covariance matrix. The variance values for the array columns are arranged along the diagonal of the covariance matrix. The remaining entries reflect the covariance between the columns of the original array. For an m -by- n matrix, the covariance matrix has size n -by- n . For example, the covariance matrix for `count`, `cov(count)`, is arranged as

```
....2 ....2 ....2
11 12 13
....2 ....2 ....2
21 22 23
2 ....2 ....2 ....31 32 33
2 2
.... ij = .... ji
```

Correlation Coefficients

`corrcoef` produces a matrix of correlation coefficients for an array of data where each row is an

observation and each column is a variable. The *correlation coefficient* is a normalized measure of the strength of the linear relationship between two variables. Uncorrelated data results in a correlation coefficient of 0; equivalent data sets have a correlation coefficient of 1. For an m -by- n matrix, the correlation coefficient matrix has size n -by- n . The arrangement of the elements in the correlation coefficient matrix corresponds to the location of the elements in the covariance matrix described above. For our traffic count example

```
corrcoef(count)  
results in
```

```
ans =  
1.0000 0.9331 0.9599  
0.9331 1.0000 0.9553  
0.9599 0.9553 1.0000
```

Clearly there is a strong linear correlation between the three traffic counts observed at the three locations, as the results are close to 1.

Finite Differences

MATLAB provides three functions for finite difference calculations.

Function Description diff Difference between successive elements of a vector. Numerical partial derivatives of a vector. gradient Numerical partial derivatives a matrix. del2 Discrete Laplacian of a matrix.

The diff function computes the difference between successive elements in a numeric vector. That is, $\text{diff}(X)$ is $[X(2)-X(1) \ X(3)-X(2) \dots \ X(n)-X(n-1)]$. So, for a vector A,

```
A = [9 -2 3 0 1 5 4];  
diff(A)  
ans =  
-1 1 5 -3 1 4 -1
```

Besides computing the first difference, diff is useful for determining certain characteristics of vectors. For example, you can use diff to determine if a vector is monotonic (elements are always either increasing or decreasing), or if a vector has equally spaced elements. This table describes a few different ways to use diff with a vector x.

Test Description $\text{diff}(x) == 0$ Tests for repeated elements. $\text{all}(\text{diff}(x) > 0)$ Tests for monotonicity. $\text{all}(\text{diff}(\text{diff}(x)) == 0)$ Tests for equally spaced vector elements.

Data Preprocessing

This section tells you how to work with

- Missing values
- Outliers and misplaced data points

Missing Values

The special value, NaN, stands for Not-a-Number in MATLAB. IEEE floating-point arithmetic convention specifies NaN as the result of undefined expressions such as 0/0.

The correct handling of missing data is a difficult problem and often varies in different

situations. For data analysis purposes, it is often convenient to use NaNs to represent missing values or data that are *not available*.

MATLAB treats NaNs in a uniform and rigorous way. They propagate naturally through to the final result in any calculation. Any mathematical calculation involving NaNs produces NaNs in the results.

For example, consider a matrix containing the 3-by-3 magic square with its center element set to NaN.

```
a = magic(3); a(2,2) = NaN
```

```
a =  
8 1 6  
3 NaN 7  
4 9 2
```

Compute a sum for each column in the matrix. `sum(a)`

```
ans =  
15 NaN 15 Any mathematical calculation involving NaNs propagates NaNs through to the final result as appropriate.
```

Data Preprocessing

You should remove NaNs from the data before performing statistical computations. Here are some ways to use `isnan` to remove NaNs from data.

Code Description

```
i = find(~isnan(x)); Find indices of elements in vector that are x = x(i) not NaNs, then keep only the nonNaN elements.
```

```
x = x(find(~isnan(x))) Remove NaNs from vector.
```

```
x = x(~isnan(x)); Remove NaNs from vector (faster). x(isnan(x)) = []; Remove NaNs from vector.
```

```
X(any(isnan(X)'), :) = []; Remove any rows of matrix X containing NaNs.
```

Note You must use the special function `isnan` to find NaNs because, by IEEE arithmetic convention, the logical comparison `NaN == NaN` always produces 0. You *cannot* use `x == NaN` = [] to remove NaNs from your data.

If you frequently need to remove NaNs, write a short M-file function. `function X = excise(X)`
`X(any(isnan(X)'), :) = [];`

Now, typing

```
X = excise(X);
```

accomplishes the same thing.

Removing Outliers

You can remove outliers or misplaced data points from a data set in much the same manner as NaNs. For the vehicle traffic count data, the mean and standard deviations of each column of the data are

```

mu = mean(count)
sigma = std(count)
mu = 32.0000 46.5417 65.5833
sigma = 25.3703 41.4057 68.0281 The number of rows with outliers greater than three standard
deviations is obtained with

```

```

[n,p] = size(count)
outliers = abs(count - mu(ones(n, 1),:)) > 3*sigma(ones(n, 1),:); nout = sum(outliers)
nout =

```

1 0 0

There is one outlier in the first column. Remove this entire observation with
 $\text{count}(\text{any}(\text{outliers}'),:) = [];$

Regression and Curve Fitting

It is often useful to find functions that describe the relationship between some variables you have observed. Identification of the coefficients of the function often leads to the formulation of an overdetermined system of simultaneous linear equations. You can find these coefficients efficiently by using the MATLAB backslash operator.

Suppose you measure a quantityy at several values of timet.

```

t = [0 .3 .8 1.1 1.6 2.3]';
y = [0.5 0.82 1.14 1.25 1.35 1.40]';
plot(t,y,'o'), grid on

```

1.4
1.3
1.2
1.1
1
0.9
0.8
0.7
0.6
0.50 0.51 1.52 2.5

The following sections look at three ways of modeling the data:

- Polynomial regression
- Linear-in-the-parameters regression
- Multiple regression

Polynomial Regression

Based on the plot, it is possible that the data can be modeled by a polynomial function
 $y = a_0 + a_1 t + a_2 t^2$

The unknown coefficients a_0 , a_1 , and a_2 can be computed by doing a *least squares fit*, which minimizes the sum of the squares of the deviations of the data from the model. There are six equations in three unknowns,

```

1
t
1
t
2 1 y1 2
y
2
1 t2 2
y 1 t3 t2 a03 = 3 32 a1y4 1 t4 t2
y
5
4 a 1
t
5
t
22
5
y6 1 t6 t2
6

```

represented by the 6-by-3 matrix

X = [ones(size(t)) t t.^2]

```

X =
1.0000 0 0
1.0000 0.3000 0.0900
1.0000 0.8000 0.6400
1.0000 1.1000 1.2100
1.0000 1.6000 2.5600
1.0000 2.3000 5.2900

```

The solution is found with the backslash operator.

a = X\y

```

a =
0.5318
0.9191
- 0.2387

```

The second-order polynomial model of the data is therefore

$y = 0.5318 + 0.919t - 0.2387t^2$ Now evaluate the model at regularly spaced points and overlay the original data in a plot.

```

T = (0:0.1:2.5)';
Y = [ones(size(T)) T T.^2]*a;

```

```
plot(T,Y,'-',t,y,'o'), grid on
```

```
1.5  
1.4  
1.3  
1.2  
1.1  
1  
0.9  
0.8  
0.7  
0.6  
0.50 0.51 1.52 2.5
```

Clearly this fit does not perfectly approximate the data. We could either increase the order of the polynomial fit, or explore some other functional form to get a better approximation.

Linear-in-the-Parameters Regression

Instead of a polynomial function, we could try using a function that is linear-in-the-parameters. In this case, consider the exponential function

$$-t + a_2 te^{-t} y = a_0 + a_1 e$$

The unknown coefficients a_0 , a_1 , and a_2 , are computed by performing a *least squares fit*.

Construct and solve the set of simultaneous equations by forming the regression matrix, X , and solving for the coefficients using the backslash operator.

```
X = [ones(size(t)) exp(-t) t.*exp(-t)];  
a = X\y
```

```
a = 1.3974  
- 0.8988  
0.4097
```

The fitted model of the data is, therefore,

$y = 1.3974 - 0.8988 e^{-t} + 0.4097 te^{-t}$ Now evaluate the model at regularly spaced points and overlay the original data in a plot.

```
T = (0:0.1:2.5)';  
Y = [ones(size(T)) exp(-T) T.*exp(-T)]*a;  
plot(T,Y,'-',t,y,'o'), grid on
```

```
1.5  
1.4  
1.3  
1.2  
1.1  
1  
0.9  
0.8  
0.7  
0.6  
0.50 0.51 1.52 2.5
```

This is a much better fit than the second-order polynomial function.

Multiple Regression

If y is a function of more than one independent variable, the matrix equations that express the relationships among the variables can be expanded to accommodate the additional data.

Suppose we measure a quantity y for several values of parameters x_1 and x_2 . The observations are entered as

```
x1 = [.2 .5 .6 .8 1.0 1.1]';  
x2 = [.1 .3 .4 .9 1.1 1.4]';  
y = [.17 .26 .28 .23 .27 .24]';
```

A multivariate model of the data is

$$y = a_0 + a_1 x_1 + a_2 x_2$$

Multiple regression solves for unknown coefficients a_0 , a_1 , and a_2 , by performing a *least squares fit*. Construct and solve the set of simultaneous equations by forming the regression matrix, X , and solving for the coefficients using the backslash operator.

```
X = [ones(size(x1)) x1 x2];  
a = X\y
```

```
a =  
0.1018  
0.4844  
-0.2847
```

The least squares fit model of the data is

$$y = 0.1018 + 0.4844 x_1 - 0.2847 x_2$$

To validate the model, find the maximum of the absolute value of the deviation of the data from the model.

```
Y = X*a;  
MaxErr = max(abs(Y - y))  
MaxErr =
```

0.0038 This is sufficiently small to be confident the model reasonably fits the data.

Case Study: Curve Fitting

This section provides an overview of some of MATLAB's basic data analysis capabilities in the form of a case study. The examples that follow work with a collection of census data, using MATLAB functions to experiment with fitting curves to the data:

- Polynomial fit
- Analyzing residuals
- Exponential fit
- Error bounds

This section also tells you how to use the Basic Fitting interface to perform curve fitting tasks.

Loading the Data

The file census.mat contains U.S. population data for the years 1790 through 1990. Load it into MATLAB.

```
load census
```

Your workspace now contains two new variables, cdate and pop:

- cdate is a column vector containing the years from 1790 to 1990 in increments of 10.
- pop is a column vector with the U.S. population figures that correspond to the years in cdate.

Polynomial Fit

A first try in fitting the census data might be a simple polynomial fit. Two MATLAB functions help with this process.

Curve Fitting Function Summary Function Description polyfit Polynomial curve fit. polyval Evaluation of polynomial fit.

MATLAB's polyfit function generates a "best fit" polynomial (in the least squares sense) of a specified order for a given set of data. For a polynomial fit of the fourth-order

```
p = polyfit(cdate,pop,4)
```

Warning: Polynomial is badly conditioned. Remove repeated data points or try centering and scaling as described in HELP POLYFIT.

```
p = 1.0e+05
```

```
    °F
```

```
0.0000 -0.0000 0.0000 -0.0126 6.0020
```

The warning arises because the polyfit function uses the cdate values as the basis for a matrix with very large values (it creates a Vandermonde matrix in its calculations – see the polyfit M-file for details). The spread of the cdate values results in scaling problems. One way to deal with this is to normalize the cdate data.

Preprocessing: Normalizing the Data

Normalization is a process of scaling the numbers in a data set to improve the accuracy of the subsequent numeric computations. A way to normalize cdate is to center it at zero mean and scale it to unit standard deviation.

```
sdate = (cdate - mean(cdate))./std(cdate)
```

Now try the fourth-degree polynomial model using the normalized data. p = polyfit(sdate,pop,4)
p = 0.7047 0.9210 23.4706 73.8598 62.2285 Evaluate the fitted polynomial at the normalized year values, and plot the fit against the observed data points.

```
pop4 = polyval(p,sdate);
```

```
plot(cdate,pop4,'-',cdate,pop,'+') , grid on
```

```
300
```

```
250
```

```
200
```

```
150
```

```
100
```

50
0
1750 1800 1850 1900 1950 2000

Another way to normalize data is to use some knowledge of the solution and units. For example, with this data set, choosing 1790 to be year zero would also have produced satisfactory results.

Analyzing Residuals

A measure of the “goodness” of fit is the *residual*, the difference between the observed and predicted data. Compare the residuals for the various fits, using normalized cdate values. It’s evident from studying the fit plots and residuals that it should be possible to do better than a simple polynomial fit with this data set.

Comparison Plots of Fit and Residual Fit Residuals

```
p1 = polyfit(sdate,pop,1);  
pop1 = polyval(p1,sdate);  
plot(cdate,pop1,'b-',cdate,pop,'g+')
```

250

Linear fit appears unsatisfactory – note negative population values at lower end of scale.

150
40 Residuals of linear fit show strongly patterned behavior.
30
20
100 10
50 0
-10 0 -20
-
50
1750 1800 1850 1900 1950 2000 -30
1750 1800 1850 1900 1950 2000

```
p = polyfit(sdate,pop,2);  
pop2 = polyval(p,sdate);  
plot(cdate,pop2,'b-',cdate,pop,'g+') res2 = pop - pop2;  
figure, plot(cdate,res2,'g+')
```

250 6
Quadratic polynomial provides better fit to data points.

4
2
150 0
100 -2
50 -4
-6

Residuals still appear strongly patterned.

0
res1 = pop - pop1;

```
figure, plot(cdate,res1,'g+')
```

```
50
```

```
-8
```

Comparison Plots of Fit and Residual (Continued) Fit Residuals

```
p = polyfit(sdate,pop,4);  
pop4 = polyval(p,sdate);  
plot(cdate,pop4,'b-',cdate,pop,'g+')
```

```
300
```

Fourth-degree model provides 250 little improvement—notethat curve still begins to turn upward 200 at lower end of plot.

```
2
```

```
0  
150 -2
```

```
100 -4
```

Residuals still appear strongly patterned.

```
50 -6  
0
```

```
res4 = pop - pop4;  
figure, plot(cdate,res4,'g+')
```

```
6  
4  
-8
```

Exponential Fit

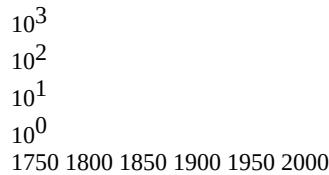
By looking at the population data plots on the previous pages, the population data curve is somewhat exponential in appearance. To take advantage of this, let's try to fit the logarithm of the population values, again working with normalized year values.

```
logp1 = polyfit(sdate,log10(pop),1);  
logpred1 = 10.^polyval(logp1,sdate);  
semilogy(cdate,logpred1,'-',cdate,pop,'+');  
grid on
```

```
103  
102  
101  
100  
1750 1800 1850 1900 1950 2000
```

Now try the logarithm analysis with a second-order model.

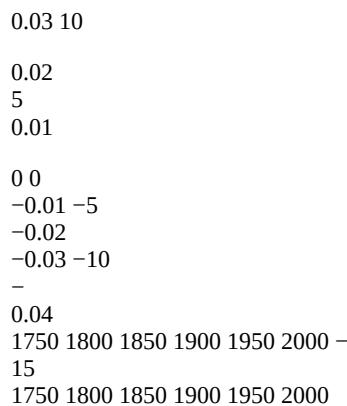
```
logp2 = polyfit(sdate,log10(pop),2);  
logpred2 = 10.^polyval(logp2,sdate);  
semilogy(cdate,logpred2,'-',cdate,pop,'+'); grid on
```



This is a more accurate model. The upper end of the plot appears to taper off, while the polynomial fits in the previous section continue, concave up, to infinity. Compare the residuals for the second-order logarithmic model.

Residuals in Log Population Scale Residuals in Population Scale

```
logres2 = log10(pop) - polyval(logp2,sdate); plot(cdate,logres2,'+') r = pop -  
10.^polyval(logp2,sdate)); plot(cdate,r,'+')
```



The residuals are more random than for the simple polynomial fit. As might be expected, the residuals tend to get larger in magnitude as the population increases. But overall, the logarithmic model provides a more accurate fit to the population data.

Error Bounds

Error bounds are useful for determining if your data is reasonably modeled by You can obtain the error bounds by passing an optional second output parameter frompolyfit as an input parameter to polyval.

This example uses the census demo data and normalizes the data by centering it at zero mean and scaling it to unit standard deviation. The example then usespolyfit and polyval to produce error bounds for a second-order polynomial model. Year values are normalized. This code uses an interval of $\frac{238}{93} \Delta t$, corresponding to a 95% confidence interval.

```
load census  
sdate = (cdate - mean(cdate))./std(cdate)  
  
[p2,S2] = polyfit(sdate,pop,2);  
[pop2,del2] = polyval(p2,sdate,S2);  
plot(cdate,pop,'+',cdate,pop2,'g-',cdate,pop2+2*del2,'r:',...
```

```
cdate,pop2-2
°Fdel2,'r:'), grid on
300
250
200
150
100
50
0
-50
1750 1800 1850 1900 1950 2000
```

The Basic Fitting Interface

MATLAB supports curve fitting through the Basic Fitting interface. Using this interface, you can quickly perform many curve fitting tasks within the same easy-to-use environment. The interface is designed so that you can:

- Fit data using a spline interpolant, a shape-preserving interpolant, or a polynomial up to degree 10.
- Plot multiple fits simultaneously for a given data set.
- Plot the fit residuals.
- Examine the numerical results of a fit.
- Evaluate (interpolate or extrapolate) a fit.
- Annotate the plot with the numerical fit results and the norm of residuals.
- Save the fit and evaluated results to the MATLAB workspace.

Depending on your specific curve fitting application, you can use the Basic Fitting interface, the command line functionality, or both.

You can use the Basic Fitting interface only



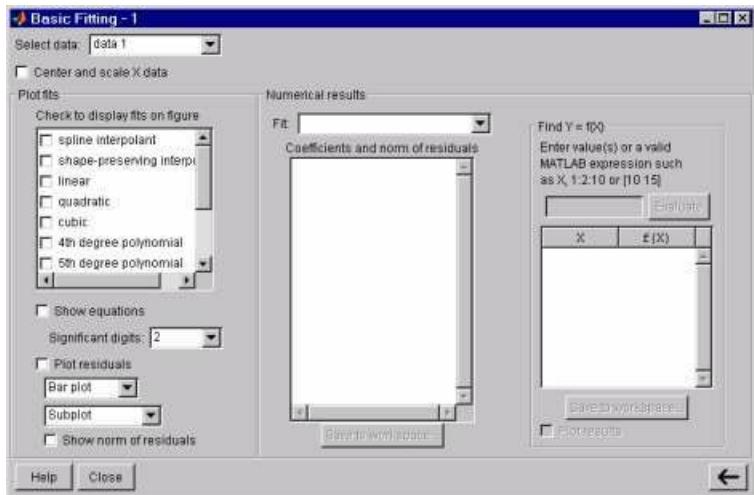
with 2-D data. However, if you plot multiple data sets as a subplot, and at least one data set is 2-D, then the interface is enabled.

Note For the HP, IBM, and SGI platforms, the Basic Fitting interface is not supported for Release 12.

Overview of the Basic Fitting Interface

The full Basic Fitting interface is shown below. To reproduce this state, follow these three steps:

- 1 Plot some data.
- 2 Select **Basic Fitting** from the **Tools** menu.
- 3 Click the button twice.



Select data – This parameter list is populated with the names of all the data sets you display in the figure window associated with the Basic Fitting interface.

Use this list to select the current data set. The current data set is defined as the data set that is to be fit. You can fit only one data set at a time. However, you can perform multiple fits for the current data set. Use the Plot Editor to change the name of a data set.

Center and scale X data – If checked, the data is centered at zero mean and scaled to unit standard deviation. You may need to center and scale your data to improve the accuracy of the subsequent numerical computations. A warning is displayed if a fit produces results that may be inaccurate.

Plot fits – This panel allows you to visually explore one or more fits to the current data set:

- **Check to display fits on figure** – Select the fits you want to display for the current data set. There are two types of fits to choose from: interpolants and polynomials. The spline interpolant uses the `spline` function, while the shape-preserving interpolant uses the `pchip` function. Refer to the `pchip` online help for a comparison of these two functions. The polynomial fits use the `polyfit` function. You can choose as many fits for a given data set as you want.

If your data set has N points, then you should use polynomials with, at most, N coefficients. If your fit uses polynomials with more than N coefficients, the interface automatically sets a sufficient number of coefficients to 0 during the calculation so that the system is not underdetermined.

- **Show equations** – If checked, the fit equation is displayed on the plot.
- **Significant digits** – Select the significant digits associated with the equation display.

- **Plot residuals** – If checked, the fit residuals are displayed. The fit residuals are defined as the difference between the ordinate data point and the resulting fit for each abscissa data point. You can display the residuals as a bar plot, as a scatter plot, or as a line plot in the same figure window as the data or in a separate figure window. If you use subplots to plot multiple data sets, then residuals can be plotted only in a separate figure window.

- **Show norm of residuals**– If checked, the norm of residuals are displayed. The norm of residuals is a measure of the goodness of fit, where a smaller value indicates a better fit than a larger value. It is calculated using the norm function, `norm(V,2)`, where `V` is the vector of residuals.

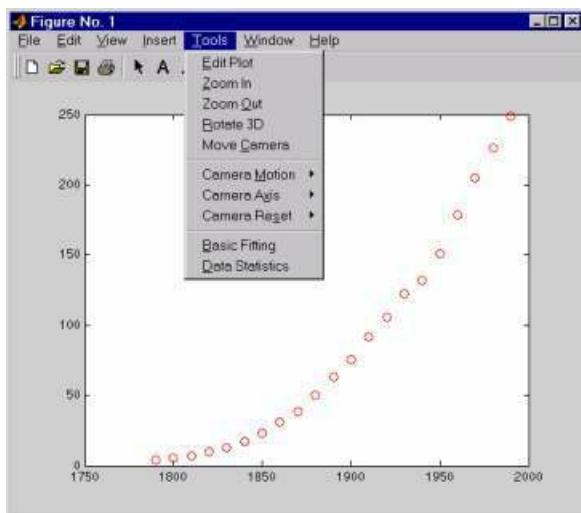
Numerical results – This panel allows you to explore the numerical results of a single fit to the current data set without plotting the fit:

- **Fit**– Select the equation to fit to the current data set. The fit results are displayed in the list box below the menu. Note that selecting an equation in this menu does not affect the state of the **Plot fits** panel. Therefore, if you want to display the fit in the data plot, you may need to select the associated check box in **Plot fits**.
- **Coefficients and norm of residuals** – Display the numerical results for the equation selected in **Fit**. Note that when you first open the **Numerical Results** panel, the results of the last fit you selected in **Plot fits** are displayed.
- **Save to workspace** – Launch a dialog box that allows you to save the fit results to workspace variables.
- **Find $Y = f(X)$** – Interpolate or extrapolate the current fit.
- **Enter value(s)**– Enter a MATLAB expression to evaluate for the current fit. The expression is evaluated after you press the **Evaluate** button, and the results are displayed in the associated table. The current fit is displayed in the **Fit** menu.
- **Save to workspace** – Launch a dialog box that allows you to save the evaluated results to workspace variables.
- **Plot results**– If checked, the evaluated results are displayed on the data plot.

Example: Using the Basic Fitting Interface

This example illustrates the features of the Basic Fitting interface by fitting a cubic polynomial to the census data. You may want to repeat this example using different equations and compare results. To launch the interface:

- 1 Plot some data.
`plot(cdate,pop,'ro')`
- 2 Select **Basic Fitting** from the **Tools** menu in the figure.



Configure the Basic Fitting interface to:

- Fit a cubic polynomial to the data.
- Display the equation in the data plot.
- Plot the fit residuals as a bar plot, and display the residuals as a subplot of the data figure window.
- Display the norm of the residuals.

This configuration is shown below.

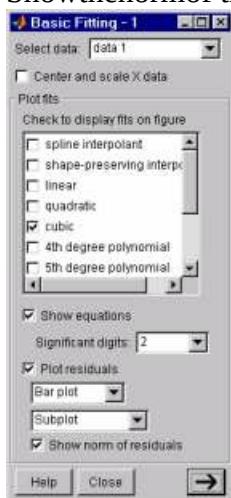
Currentdataset

Fitacubicpolynomial to the data

Showtheequation

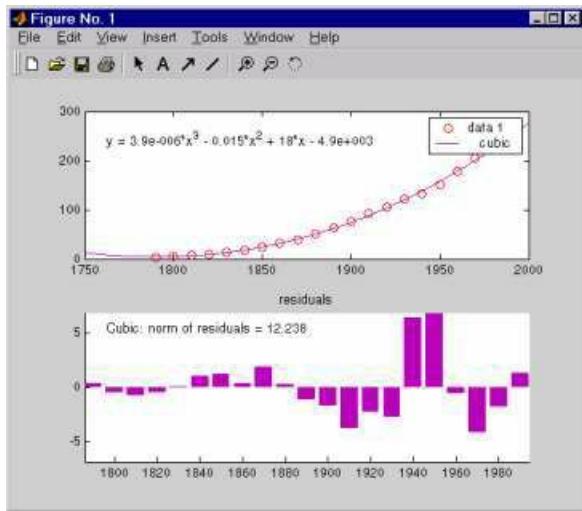
Plottheresidualsasabarplotin thedatafigurewindow

Showthenormof theresiduals



The **Plot fits** panel allows you to visually explore multiple fits to the current data set. For comparison, try fitting additional equations to the census data by selecting the appropriate check boxes. If an equation produces results that may be numerically inaccurate, a warning is displayed. In this case, you should select the **Center and scale X data** check box to improve the numerical accuracy.

The resulting fit and the residuals are shown below.

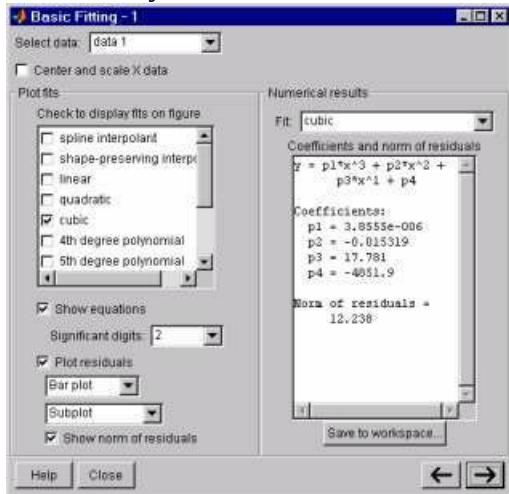


The plot legend indicates the name of the data set and the equation. The legend is automatically updated as you add or remove data sets or fits. Additionally, fits are displayed using a default set of line styles and colors. You can change any of the default plot settings using the Plot Editor. However, any changes you make are undone if you subsequently perform another fit. To retain changes, you should wait until after you have finished fitting your data.

Note If you change the name of a data set in the legend, then the name is automatically changed in the **Select data** menu.

By selecting the

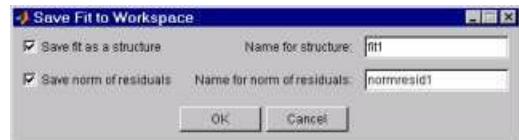
→ button, you can examine the fit coefficients and the norm of the residuals.



The **Fit** menu allows you to explore numerical fit results for the current data set without plotting the fit. For comparison, you can display the numerical results for other fits by selecting the desired equation. Note that if you want to display a fit in the data plot, you have to select the associated check box in **Plot fits**.

You can save the fit results to the MATLAB workspace by selecting the **Save to workspace**

button.



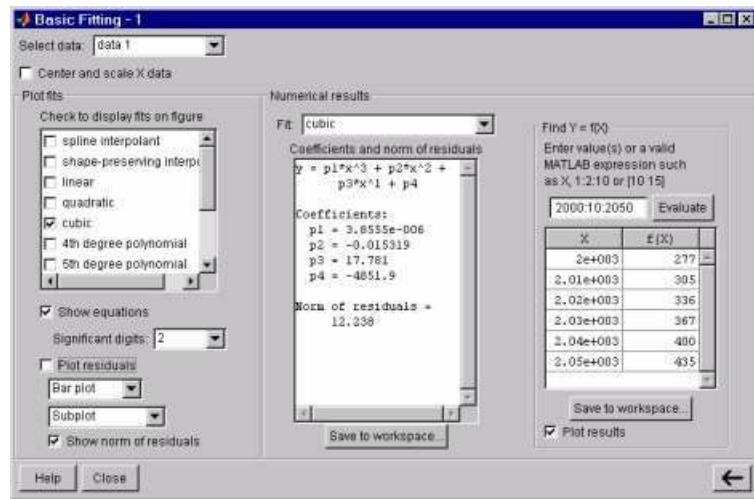
The fit structure is shown below.

```
fit1
fit1 =
type: 'polynomial degree 3'
coeff: [3.8555e-006 -0.0153 17.7815 -4.8519e+003]
```

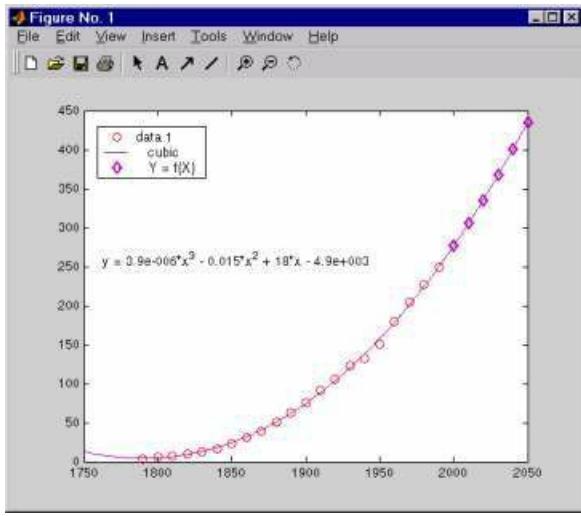
You may want to use this structure for subsequent display or analysis. For example, you can use the saved coefficients and the polyval function to evaluate the cubic polynomial at the command line.

By selecting the button again, you can evaluate the current fit at the

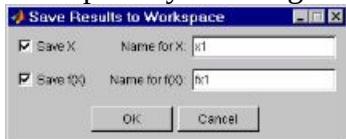
specified abscissa values. The current fit is displayed in the **Fit** menu. In this example, the population for the years 2000 to 2050 is evaluated in increments of 10, and then displayed in the data plot.



The evaluated data is shown below.



You can save the evaluated data to the MATLAB workspace by selecting the **Save to workspace** button.



Difference Equations and Filtering

MATLAB has functions for working with difference equations and filters. These functions operate primarily on vectors.

Vectors are used to hold sampled-data signals, or sequences, for signal processing and data analysis. For multi-input systems, each row of a matrix corresponds to a sample point with each input appearing as columns of the matrix.

The function

$y = \text{filter}(b, a, x)$

processes the data in vector x with the filter described by vectors a and b , creating filtered data y .

The filter command can be thought of as an efficient implementation of the difference equation. The filter structure is the general tapped delay-line filter described by the difference equation below, where n is the index of the current sample, na is the order of the polynomial described by vector a and nb is the order of the polynomial described by vector b . The output $y(n)$, is a linear combination of current and previous inputs, $x(n)$ $x(n-1)$..., and previous outputs, $y(n-1)$ $y(n-2)$...

$$\begin{aligned} a_1 y_{n-1} &= b_1 x_n + b_2 x_{n-1} + \dots + b_{nb} x_{n-nb} + a_1 y_{n-1} \\ &- a_2 y_{n-2} - \dots - a_{na} y_{n-na} \end{aligned}$$

Suppose, for example, we want to smooth our traffic count data with a moving average filter to see the average traffic flow over a 4-hour window. This process is represented by the difference equation

$$\begin{aligned} y_{n-1} &= -x_{n-1} + -x_{n-2} + -x_{n-3} \\ y_n &= -x_n + -x_{n-1} + -x_{n-2} + -x_{n-3} \end{aligned}$$

4 4 4 4

The corresponding vectors are

```
a = 1;  
b = [1/4 1/4 1/4 1/4];
```

Difference Equations and Filtering

Note Enter the `format` command, `format rat`, to display and enter data using the rational format.

Executing the command

```
load count.dat
```

creates the matrix `count` in the workspace.

For this example, extract the first column of traffic counts and assign it to the vector `x`.

```
x = count(:,1);
```

The 4-hour moving-average of the data is efficiently calculated with `y = filter(b,a,x);`

Compare the original data and the smoothed data with an overlaid plot of the two curves.

```
t = 1:length(x);  
plot(t,x,'.',t,y,'-'), grid on  
legend('Original Data','Smoothed Data',2)
```

```
120 Original Data Smoothed Data  
100  
80  
60  
40  
20  
0 0 5 10 15 20 25
```

The filtered data represented by the solid line is the 4-hour moving average of the observed traffic count data represented by the dashed line. For practical filtering applications, the Signal Processing Toolbox includes numerous functions for designing and analyzing filters.

Fourier Analysis and the Fast Fourier Transform (FFT)

Fourier analysis is extremely useful for data analysis, as it breaks down a signal into constituent sinusoids of different frequencies. For sampled vector data, Fourier analysis is performed using the discrete Fourier transform (DFT).

The fast Fourier transform (FFT) is an efficient algorithm for computing the DFT of a sequence; it is not a separate transform. It is particularly useful in areas such as signal and image processing, where its uses range from filtering, convolution, and frequency analysis to power spectrum estimation.

This section:

- Summarizes the Fourier transform functions
- Introduces Fourier transform analysis with an example about sunspot activity
- Calculates magnitude and phase of transformed data
- Discusses the dependence of execution time on length of the transform

Function Summary

MATLAB provides a collection of functions for computing and working with Fourier transforms.

FFT Function Summary Function Description fft Discrete Fourier transform. fft2 Two-dimensional discrete Fourier transform. ifft N-dimensional discrete Fourier transform. ifft Inverse discrete Fourier transform. ifft2 Two-dimensional inverse discrete Fourier transform. ifftn N-dimensional inverse discrete Fourier transform. abs Magnitude.

FFT Function Summary (Continued) Function Description angle Phase angle. unwrap Unwrap phase angle in radians. fftshift Move zeroth lag to center of spectrum. cplxpair Sort numbers into complex conjugate pairs. nextpow2 Next higher power of two.

Introduction

For length N input sequence x , the DFT is a length N vector, X .fft and ifft implement the relationships

$$\sum_{k=0}^{N-1} x(k) e^{-j2\pi k \frac{n}{N}} = X(n)$$

X
 \star
 k
 f

$=$
 \star
 x
 \star
 n
 f
 e

----- \star

N

$n = 1$

$$\sum_{k=0}^{N-1} x(k) e^{-j2\pi k \frac{n-1}{N}} = X(n-1)$$

\star

n
 f

$=$

$$X(n) = \sum_{k=0}^{N-1} x(k) e^{-j2\pi k \frac{n}{N}}$$

N

$k = 1$

Note Since the first element of a MATLAB vector has an index 1, the summations in the equations above are from 1 to N . These produce identical results as traditional Fourier equations with summations from 0 to $N-1$.

If $x(n)$ is real, we can rewrite the above equation in terms of a summation of sine and cosine functions with real coefficients

$$x[n] = \sum_{k=1}^N \left[a_k \cos(k\omega_n) + b_k \sin(k\omega_n) \right]$$

where

$$a_k = \text{real}(X[k]), b_k = -\text{imag}(X[k]), 1 \leq k \leq N$$

Finding an FFT

The FFT of a column vector x

$$x = [4 3 7 -9 1 0 0 0]'$$

is found with

$$y = \text{fft}(x)$$

which results in

$$\begin{aligned} y = & \\ & 6.0000 \\ & 11.4853 - 2.7574i \\ & -2.0000 - 12.0000i \\ & -5.4853 + 11.2426i \\ & 18.0000 \\ & -5.4853 - 11.2426i \\ & -2.0000 + 12.0000i \\ & 11.4853 + 2.7574i \end{aligned}$$

Notice that although the sequence x is real, y is complex. The first component of the transformed data is the constant contribution and the fifth element corresponds to the Nyquist frequency. The last three values of y correspond to negative frequencies and, for the real sequence x, they are complex conjugates of three components in the first half of y.

Example: Using FFT to Calculate Sunspot Periodicity

Suppose, we want to analyze the variations in sunspot activity over the last 300 years. You are probably aware that sunspot activity is cyclical, reaching a maximum about every 11 years. Let's confirm that.

Astronomers have tabulated a quantity called the Wolfer number for almost 300 years. This quantity measures both number and size of sunspots. Load and plot the sunspot data.

```
load sunspot.dat
year = sunspot(:,1);
wolfer = sunspot(:,2); plot(year,wolfer)
title('Sunspot Data')
```

Sunspot Data

200
180
160
140
120

```

100
80
60
40
20
0
1700 1750 1800 1850 1900 1950 2000

```

Now take the FFT of the sunspot data.

```
Y = fft(wolfer);
```

The result of this transform is the complex vector, Y. The magnitude of Y squared is called the power and a plot of power versus frequency is a “periodogram.” Remove the first component of Y, which is simply the sum of the data, and plot the results.

```

N = length(Y);
Y(1) = [];
power = abs(Y(1:N/2)).^2;
nyquist = 1/2;
freq = (1:N/2)/(N/2)*nyquist;
plot(freq,power), grid on
xlabel('cycles/year')
title('Periodogram')

```

```

x 107 Periodogram 2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
00 0.05 0.1 0.15 0.2 0.25 0.3 0.35 0.4 0.45 0.5 cycles/year

```

The scale in cycles/year is somewhat inconvenient. Let’s plot in years/cycle and estimate what one cycle is. For convenience, plot the power versus period (where period = 1./freq) from 0 to 40 years/cycle.

```

period = 1./freq;
plot(period,power), axis([0 40 0 2e7]), grid on
ylabel('Power')
xlabel('Period(Years/Cycle)')

```

```

2 x 107
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4

```

```
0.2  
00 5 10 15 20 25 30 35 40 Period(Years/Cycle)
```

In order to determine the cycle more precisely, [mp,index] = max(power);

period(index)

ans = 11.0769

Magnitude and Phase of Transformed Data

Important information about a transformed sequence includes its magnitude and phase. The MATLAB functions `abs` and `angle` calculate this information. To try this, create a time vector `t`, and use this vector to create a sequence `x` consisting of two sinusoids at different frequencies.

```
t = 0:1/100:10-1/100;
```

```
x = sin(2*pi*15*t) + sin(2*pi*40*t);
```

Now use the `fft` function to compute the DFT of the sequence. The code below calculates the magnitude and phase of the transformed sequence. It uses the `abs` function to obtain the magnitude of the data, the `angle` function to obtain the phase information, and `unwrap` to remove phase jumps greater than π to their 2π complement.

```
y = fft(x);  
m = abs(y);  
p = unwrap(angle(y));
```

Now create a frequency vector for the `x`-axis and plot the magnitude and phase.

```
f = (0:length(y)-1)*100/length(y);  
subplot(2,1,1), plot(f,m),  
ylabel('Abs. Magnitude'), grid on  
subplot(2,1,2), plot(f,p*180/pi)  
ylabel('Phase [Degrees]'), grid on  
xlabel('Frequency [Hertz]')
```

The magnitude plot is perfectly symmetrical about the Nyquist frequency of 50 hertz. The useful information in the signal is found in the range 0 to 50 hertz.

```
500  
400  
300  
200  
100  
00 10 20 30 40 50 60 70 80 90 100  
 $2.5 \times 10^4$ 
```

```
2  
1.5  
1  
0.5  
0
```



-0.5 0 10 20 30 40 50 60 70 80 90 100 Frequency [Hertz]

FFT Length Versus Speed

You can add a second argument to `fft` to specify a number of points `n` for the transform

`y = fft(x,n)`

With this syntax, `fft` pads `x` with zeros if it is shorter than `n`, or truncates it if it is longer than `n`. If you do not specify `n`, `fft` defaults to the length of the input sequence.

The execution time for `fft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

The inverse FFT function `ifft` also accepts a transform length argument. For practical application of the FFT, the Signal Processing Toolbox includes numerous functions for spectral analysis.

14

Function Functions

Function Summary	14-3
Representing Functions in MATLAB	14-4
Plotting Mathematical Functions	14-6
Minimizing Functions and Finding Zeros	14-9
Minimizing Functions of One Variable	14-9
Minimizing Functions of Several Variables	14-10
Setting Minimization Options.....	14-11
Finding Zeros of Functions.....	14-12
Tips	14-15
Troubleshooting.....	14-15
Converting Your Optimization Code to MATLAB Version	
⁵ Syntax	14-16
Numerical Integration (Quadrature)	14-19
Example: Computing the Length of a Curve.....	14-19
Example: Double Integration	14-20
All of the functions described in this chapter are called <i>function functions</i> because they accept a function as an input arguments. You can pass such a function either as a function handle or as an inline object that defines a mathematical function. The function that is passed in is referred to as the <i>objective function</i> .	

This chapter includes:

Function Summary

A summary of some function functions

Representing Functions in MATLAB

Some guidelines for representing functions in MATLAB

Plotting Mathematical Functions

A discussion about using `fplot` to plot mathematical functions

Minimizing Functions and Finding Zeros

A discussion of high-level function functions that perform optimization-related tasks

Numerical Integration (Quadrature)

A discussion of the MATLAB quadrature functions

Note See the “Differential Equations” and “Sparse Matrices” chapters for information about the use of other function functions.

Note For information about function handles, see `thefunction_handle (@)`, `func2str`, and `str2func` reference pages, and the “Function Handles” section of “Programming and Data Types” in the MATLAB documentation. Function Summary

Function Summary

The function functions are located in the MATLABfunfun directory. This table provides a brief description of the functions discussed in this chapter. Related functions are grouped by category.

Function Summary Category Function

Description Plotting

Optimization and zero finding

`fplot`

`fminbnd`

`fminsearch`

`fzero` Numerical integration

`quad`

`quadl`

`dblquad`

Plot function.

Minimize function of one variable with bound constraints.

Minimize function of several variables. Find zero of function of one variable.

Numerically evaluate integral, adaptive Simpson quadrature.

Numerically evaluate integral, adaptive Lobatto quadrature.

Numerically evaluate double integral.

Representing Functions in MATLAB

MATLAB can represent mathematical functions by expressing them as MATLAB functions in M-files or as inline objects. For example, consider the function

$$f(x) = -x^6 + 6x^4 - 16x^2 + 1$$

$$F(x) = 0.3x^2 + 0.01$$

This function can be used as input to any of the function functions.

As MATLAB Functions

You can find the function above in the M-file named `humps.m`. function `y = humps(x)`

`y = 1./((x - 0.3).^2 + 0.01) + 1./((x - 0.9).^2 + 0.04) - 6;` To evaluate the function `humps` at 2.0, use `@` to obtain a function handle for `humps`, and then pass the function handle to `feval`.

```
fh = @humps;
```

```
feval(fh,2.0)
```

```
ans =
```

-4.8552

As Inline Objects

A second way to represent a mathematical function at the command line is by creating an *inline* object from a string expression. For example, you can create an inline object of the humps function

```
f = inline('1./((x-0.3).^2 + 0.01) + 1./((x-0.9).^2 + 0.04)-6'); You can then evaluate f at 2.0.  
f(2.0)  
ans =  
-4.8552
```

You can also create functions of more than one argument with *inline* by specifying the names of the input arguments along with the string expression. For example, the following function has two input arguments *x* and *y*.

```
Representing Functions in MATLAB  
f= inline('y*sin(x)+x*cos(y)','x','y') f(pi,2*pi)  
ans =
```

3.1416

Plotting Mathematical Functions

The *fplot* function plots a mathematical function between a given set of axes limits. You can control the *x*-axis limits only, or both the *x*- and *y*-axis limits. For example, to plot the *humps* function over the *x*-axis range [-5 5], use

```
fplot(@humps,[-5 5])  
grid on  
100  
80  
60  
40  
20  
0  
-20 -5 -4 -3 -2 -1 0 1 2 3 4 5
```

You can zoom in on the function by selecting *y*-axis limits of -10 and 25, using *fplot(@humps,[-5 5 -10 25])*

```
grid on  
Plotting Mathematical Functions  
25  
20  
15  
10  
5  
0  
-5  
-10 -5 -4 -3 -2 -1 0 1 2 3 4 5
```

You can also pass an *inline* for *fplot* to graph, as in
*fplot(inline('2*sin(x+3)'),[-1 1])*

You can plot more than one function on the same graph with one call to fplot. If you use this with a function, then the function must take a column vector x and return a matrix where each column corresponds to each function, evaluated at each value of x .

If you pass an inline object of several functions to fplot, the inline object also must return a matrix where each column corresponds to each function evaluated at each value of x , as in

```
fplot(inline(['2*sin(x+3), humps(x)']),[-5 5])
```

which plots the first and second functions on the same graph.

```
100  
80  
60  
40  
20  
0  
-20 5 -4 -3 -2 -1 0 1 2 3 4 5
```

Note that the inline

```
f= inline(['2*sin(x+3), humps(x)'])
```

evaluates to a matrix of two columns, one for each function, when x is a column vector.

```
f([1;2;3])
```

returns

```
-1.5136 16.0000  
-1.9178 -4.8552  
-0.5588 -5.6383
```

Minimizing Functions and Finding Zeros

MATLAB provides a number of high-level function functions that perform optimization-related tasks. This section describes:

- Minimizing a function of one variable
- Minimizing a function of several variables
- Setting minimization options
- Finding a zero of a function of one variable
- Converting your code to MATLAB Version 5 syntax

The MATLAB optimization functions are:

```
fminbnd fminsearch fzero lsqnonneg optimget optimset
```

Minimize a function of one variable on a fixed interval Minimize a function of several variables
Find zero of a function of one variable

Linear least squares with nonnegativity constraints Get optimization options structure parameter values
Create or edit optimization options parameter structure

For more optimization capabilities, see the Optimization Toolbox.

Minimizing Functions of One Variable

Given a mathematical function of a single variable coded in an M-file, you can use the fminbnd

function to find a local minimizer of the function in a given interval. For example, to find a minimum of the humps function in the range (0.3, 1), use

```
x = fminbnd(@humps,0.3,1)
```

which returns

x =

0.6370 You can ask for a tabular display of output by passing a fourth argument created by the optimset command tofminbnd

```
x = fminbnd(@humps,0.3,1,optimset('Display','iter')) which gives the output
```

Func-count x f(x)

1 0.567376 12.9098

2 0.732624 13.7746

3 0.465248 25.1714

4 0.644416 11.2693

5 0.6413 11.2583

6 0.637618 11.2529

7 0.636985 11.2528

8 0.637019 11.2528

9 0.637052 11.2528

Procedure initial

golden

golden

parabolic parabolic parabolic parabolic parabolic parabolic

x = 0.6370

This shows the current value of x and the function value atf(x) each time a function evaluation occurs. Forfminbnd, one function evaluation corresponds to one iteration of the algorithm. The last column shows what procedure is being used at each iteration, either a golden section search or a parabolic interpolation.

Minimizing Functions of Several Variables

The fminsearch function is similar tofminbnd except that it handles functions of many variables, and you specify a starting vector x_0 rather than a starting interval.fminsearch attempts to return a vector x that is a local minimizer of the mathematical function near this starting vector.

To tryfminsearch, create a functionthree_var of three variables,x,y, andz.

```
function b = three_var(v)
x = v(1);
y = v(2);
z = v(3);
b = x.^2 + 2.5*sin(y) - z^2*x^2*y^2;
```

Now find a minimum for this function usingx = -0.6,y = -1.2, and z = 0.135 as the starting

values.

```
v = [-0.6 -1.2 0.135];
a = fminsearch(@three_var,v)
a = 0.0000 -1.5708 0.1803
```

Setting Minimization Options

You can specify control options that set some minimization parameters by calling fminbnd with the syntax

```
x = fminbnd(fun,x1,x2,options)
or fminsearch with the syntax
x = fminsearch(fun,x0,options)
```

options is a structure used by the optimization functions. Use optimset to set the values of the options structure.

```
options = optimset('Display','iter');
```

fminbnd and fminsearch use only the options parameters shown in the following table. See the optimset reference page for a complete list of the parameters that are used in the Optimization Toolbox.

options.Display
options.TolX
options.TolFun
options.MaxIter options.MaxFunEvals

A flag that determines if intermediate steps in the minimization appear on the screen. If set to 'iter', intermediate steps are displayed; if set to 'off', no intermediate solutions are displayed, if set to final, displays just the final output.

The termination tolerance for x . Its default value is 1.e-4.

The termination tolerance for the function value. The default value is 1.e-4. This parameter is used by fminsearch, but not fminbnd.

Maximum number of iterations allowed.

The maximum number of function evaluations allowed. The default value is 500 for fminbnd and 200*length(x_0) for fminsearch.

The number of function evaluations, the number of iterations, and the algorithm are returned in the structure output when you provide fminbnd or fminsearch with a fourth output argument, as in

```
[x,fval,exitflag,output] = fminbnd(@humps,0.3,1);
or
[x,fval,exitflag,output] = fminsearch(@three_var,v);
```

Finding Zeros of Functions

The fzero function attempts to find a zero of one equation with one variable. You can call this function with either a one-element starting point or a two-element vector that designates a

starting interval. If you givefzero a starting point x_0 ,fzero first searches for an interval around this point where the function changes sign. If the interval is found, thenfzero returns a value near where the function changes sign. If no such interval is found,fzero returnsNaN. Alternatively, if you know two points where the function value differs in sign, you can specify this starting interval using a two-element vector;fzero is guaranteed to narrow down the interval and return a value near a sign change.

Usefzero to find a zero of thehumps function near-0.2

```
a = fzero(@humps,-0.2)
a =
-0.1316
```

For this starting point, fzero searches in the neighborhood of 0.2 until it finds a change of sign between -0.10949 and -0.264. This interval is then narrowed down to -0.1316. You can verify that -0.1316 has a function value very close to zero using

```
humps(a)
ans =
8.8818e -16 Suppose you know two places where the function value ofhumps differs in sign such
as x = 1 and x = -1. You can use
humps(1)
ans = 16
humps(-1)
ans =
-5.1378
```

Then you can give fzero this interval to start with andfzero then returns a point near where the function changes sign. You can display information as fzero progresses with

```
options = optimset('Display','iter');
a = fzero(@humps,[-1 1],options)
```

Func-count x f(x) Procedure

```
1 -1 -5.13779
1 1 16
2 -0.513876 -4.02235
3 0.243062 71.6382
4 -0.473635 -3.83767
5 -0.115287 0.414441
6 -0.150214 -0.423446
7 -0.132562 -0.0226907
8 -0.131666 -0.0011492
9 -0.131618 1.88371e-07
10 -0.131618 -2.7935e-11
11 -0.131618 8.88178e-16
12 -0.131618 -9.76996e-15
initial
initial
```

```
interpolation bisection  
interpolation bisection  
interpolation interpolation interpolation interpolation interpolation interpolation interpolation
```

```
a =  
-0.1316
```

The steps of the algorithm include both bisection and interpolation under the Procedure column. If the example had started with a scalar starting point instead of an interval, the first steps after the initial function evaluations would have included some search steps while fzero searched for an interval containing a sign change.

You can specify a relative error tolerance using optimset. In the call above, passing in the empty matrix causes the default relative error tolerance ofeps to be used.

Tips

Optimization problems may take many iterations to converge. Most optimization problems benefit from good starting guesses. Providing good starting guesses improves the execution efficiency and may help locate the global minimum instead of a local minimum.

Sophisticated problems are best solved by an evolutionary approach, whereby a problem with a smaller number of independent variables is solved first. Solutions from lower order problems can generally be used as starting points for higher order problems by using an appropriate mapping.

The use of simpler cost functions and less stringent termination criteria in the early stages of an optimization problem can also reduce computation time. Such an approach often produces superior results by avoiding local minima.

Troubleshooting

Below is a list of typical problems and recommendations for dealing with them.

Problem Recommendation

The solution found by fminbnd or fminsearch does not appear to be a global minimum.

Sometimes an optimization problem has values of x for which it is impossible to evaluate f .

The minimization routine appears to enter an infinite loop or returns a solution that is not a minimum (or not a zero in the case of fzero).

There is no guarantee that you have a global minimum unless your problem is continuous and has only one minimum. Starting the optimization from a number of different starting points (or intervals in the case of fminbnd) may help to locate the global minimum or verify that there is only one minimum. Use different methods, where possible, to verify results.

Modify your function to include a penalty function to give a large positive value to f when infeasibility is encountered.

Your objective function (`fun`) may be returning Inf, NaN, or complex values. The optimization

routines expect only real numbers to be returned. Any other values may cause unexpected results. Insert code into your objective function M-file to verify that only real numbers are returned (use the functions `isreal` and `isfinite`).

Converting Your Optimization Code to MATLAB Version 5 Syntax

Most of the function names and calling sequences changed in Version 5 to accommodate new functionality and to clarify the roles of the input and output variables.

This table lists the optimization functions provided by MATLAB and indicates the functions whose names have changed in Version 5.

Old (Version 4) Name New (Version 5) Name

`fmin` `fminbnd` `fmins` `fminsearch` `foptions` `optimget`,`optimset`

`fzero` `nnls` `fzero` (name unchanged) `lsqnonneg`

This section:

- Tells you how to override default parameter settings with the `newoptimset` and `optimget` functions.
- Explains the reasons for the new calling sequences and explains how to convert your code. In addition to the information in this section, consult the individual function reference pages for information about the new functions and about the arguments they take.

Using `optimset` and `optimget`

The `optimset` function replaces `foptions` for overriding default parameter settings. `optimset` creates an `options` structure that contains parameters used

in the optimization routines. If, on the first call to an optimization routine, the options structure is not provided, or is empty, a set of default parameters is generated. See the `optimset` reference page for details.

New Calling Sequences

Version 5 of MATLAB makes these changes in the calling sequences:

- Each function takes an `options` structure to adjust parameters to the optimization functions (see `optimset`, `optimget`).
 - The new default output gives information if the function does not converge. (the Version 4 default was no output, Version 5 used 'final' as the default, the new default is `options.display = 'notify'`).
 - Each function returns an `exitflag` that describes the termination state.
- Each function now has an `output` structure that contains information about the problem solution relevant to that function.

The sections below describe how to convert from the old function names and calling sequences to the new ones. The calls shown are the most general cases, involving all possible input and

output arguments. Note that many of these arguments are optional. See the function reference pages for more information.

Converting from fmin to fminbnd. In Version 4, you used this call tofmin. [X,OPTIONS] = fmin('FUN',x1,x2,OPTIONS,P1,P2,...);

In Version 5, you callfminbnd like this.

```
[X,FVAL,EXITFLAG,OUTPUT] = fminbnd(@FUN,x1,x2,...  
OPTIONS,P1,P2,...);
```

Converting from fmins to fminsearch. In Version 4, you used this call tofmins. [X,OPTIONS] = fmmins('FUN',x0,OPTIONS,[],P1,P2,...);

In Version 5, you callfminsearch like this.

```
[X,FVAL,EXITFLAG,OUTPUT] = fminsearch(@FUN,x0,...  
OPTIONS,P1,P2,...);
```

Converting to the new form of fzero. In Version 4, you used this call tofzero. X = fzero('F',X,TOL,TRACE,P1,P2,...);

In Version 5, replace theTRACE andTOL arguments with

```
if TRACE == 0,  
val = 'none';  
elseif TRACE == 1  
val = 'iter';  
end  
OPTIONS = optimset('Display',val,'TolX',TOL);
```

Now callfzero like this.

```
[X,FVAL,EXITFLAG,OUTPUT] = fzero(@F,X,OPTIONS,P1,P2,...);
```

Converting from nnls to lsqnonneg. In Version 4, you used this call tonnls. [X,LAMBDA] = nnls(A,b,tol);

In Version 5, replace thetol argument with

```
OPTIONS = optimset('Display','none','TolX',tol);
```

Now callsqnonneg like this.

```
[X,RESNORM,RESIDUAL,EXITFLAG,OUTPUT,LAMBDA] =  
lsqnonneg(A,b,X0,OPTIONS);
```

Numerical Integration (Quadrature)

The area beneath a section of a function $F(x)$ can be determined by numerically integrating $F(x)$, a process referred to as *quadrature*. The MATLAB quadrature functions are:

quad quadl dblquad

Use adaptive Simpson quadrature Use adaptive Lobatto quadrature Numerically evaluate double integral

To integrate the function defined byhumps.m from0 to1, use q = quad(@humps,0,1)

$q = 29.8583$ Both quad and quadl operate recursively. If either method detects a possible singularity, it prints a warning.

You can include a fourth argument for quad or quadl that specifies a relative error tolerance for the integration. If a nonzero fifth argument is passed to quad or quadl, the function evaluations are traced.

Two examples illustrate use of these functions:

- Computing the length of a curve
- Double integration

Example: Computing the Length of a Curve

You can use quad or quadl to compute the length of a curve. Consider the curve parameterized by the equations

$$x = \sin(2t) \quad y = \cos(t) \quad z = t \\ \text{where } t \in [0, 3\sqrt{8}]$$

A three-dimensional plot of this curve is

```
t = 0:0.1:3*pi;
plot3(sin(2*t),cos(t),t)
```

The arc length formula says the length of the curve is the integral of the norm of the derivatives of the parameterized equations

$$\int_0^{3\sqrt{8}} \sqrt{4 \cos^2(2t) + \sin^2(t) + 1} dt$$

The function hcurve computes the integrand function $f = hcurve(t)$

```
f = sqrt(4*cos(2*t).^2 + sin(t).^2 + 1); Integrate this function with a call to quad len =
quad(@hcurve,0,3*pi)
```

len =

1.7222e+01 The length of this curve is about 17.2.

Example: Double Integration

Consider the numerical solution of

$y = ax$

$x = ay$

$$\int_0^a \int_0^{ay} f(x,y) dx dy$$

$y = in$ $x = in$

For this example $f(x,y) = y \sin(x) + x \cos(y)$. The first step is to build the

function to be evaluated. The function must be capable of returning a vector output when given a vector input. You must also consider which variable is in the inner integral, and which goes in the outer integral. In this example, the inner variable is x and the outer variable is y (the order in the integral is $dxdy$). In this case, the integrand function is

```
function out = integrnd(x,y)
out = y*sin(x) + x*cos(y);
```

To perform the integration, two functions are available in the funfun directory. The first, dblquad, is called directly from the command line. This M-file evaluates the outer loop using quad. At each iteration, quad calls the second helper function that evaluates the inner loop.

To evaluate the double integral, use

```
result = dblquad(@integrnd,xmin,xmax,ymin,ymax);
```

The first argument is a string with the name of the integrand function. The second to fifth arguments are

xmin Lower limit of inner integral xmax Upper limit of the inner integral
ymin Lower limit of outer integral ymax Upper limit of the outer integral

Here is a numeric example that illustrates the use of dblquad.

```
xmin = pi;
xmax = 2*pi;
ymin = 0;
ymax = pi;
result = dblquad(@integrnd,xmin,xmax,ymin,ymax)
```

The result is -9.8698.

By default, dblquad calls quad. To integrate the previous example using quadl (with the default values for the tolerance argument), use

```
result = dblquad(@integrnd,xmin,xmax,ymin,ymax,[],@quadl);
```

Alternatively, any user-defined quadrature function name can be passed to dblquad as long as the quadrature function has the same calling and return arguments as quad.

15

Differential Equations

Initial Value Problems for ODEs and DAEs	15-3
ODE Function Summary.....	15-3
Introduction to Initial Value ODE Problems.....	15-6
Initial Value Problem Solvers	15-6
Representing ODE Problems	15-10
Improving ODE Solver Performance.....	15-16
Examples: Applying the ODE Initial Value Problem Solvers	15-31
Questions and Answers, and Troubleshooting	15-50
Boundary Value Problems for ODEs	15-57
BVP Function Summary	15-57
Introduction to Boundary Value ODE Problems.....	15-59
Boundary Value Problem Solver	15-60

Representing BVP Problems.....	15-63
Using Continuation to Make a Good Initial Guess	15-68
Improving BVP Solver Performance	15-75
P artial Differential Equations	15-82
PDE Function Summary	15-82
Introduction to PDE Problems	15-83
MATLAB Partial Differential Equation Solver.....	15-84
Representing PDE Problems.....	15-88
Improving PDE Solver Performance	15-93
Example: Electrodynamics Problem.....	15-94

Selected Bibliography15-99

This chapter treats the numerical solution of differential equations using MATLAB. It includes:

Initial Value Problems for ODEs and DAEs

Describes the solution of ordinary differential equations (ODEs) and differential-algebraic equations (DAEs), where the solution of interest satisfies initial conditions at a given initial value of the independent variable.

Boundary Value Problems for ODEs

Describes the solution of ODEs, where the solution of interest satisfies certain boundary conditions. The boundary conditions specify a relationship between the values of the solution at the initial and final values of the independent variable.

Partial Differential Equations

Describes the solution of initial-boundary value problems for systems of parabolic and elliptic partial differential equations (PDEs) in one spatial variable and time.

Selected Bibliography

Lists published materials that support concepts described in this chapter.

Note In function tables, commonly used functions are listed first, followed by more advanced functions. The same is true of property tables.

Initial Value Problems for ODEs and DAEs

This section describes how to use MATLAB to solve initial value problems (IVPs) of ordinary differential equations (ODEs) and differential-algebraic equations (DAEs). It provides:

- A summary of the MATLAB ODE solvers, related functions, and examples
- An introduction to ODEs and initial value problems
- Descriptions of the ODE solvers and their basic syntax
- General instructions for representing an IVP in MATLAB
- A discussion about changing default integration properties to improve solver performance
- Examples of the kinds of IVP problems you can solve in MATLAB

- Answers to some frequently asked questions about the MATLAB ODE solvers, and some troubleshooting suggestions

ODE Function Summ ary

Initial Value ODE Problem Solvers

These are the MATLAB initial value problem solvers. The table lists the kind of problem you can solve with each solver, and the method each solver uses.

Solver Solves These Kinds of Problems Method
ode45 Nonstiff differential equations Runge-Kutta
ode23 Nonstiff differential equations Runge-Kutta
ode113 Nonstiff differential equations
Adams
ode15s Stiff differential equations and DAEs NDFs (BDFs)
ode23s Stiff differential equations
ode23t Moderately stiff differential equations and DAEs
Rosenbrock Trapezoidal rule
ode23tb Stiff differential equations TR-BDF2

ODE Solution Evaluation

If you call an ODE solver with one output argument, it returns a structure that you can use to evaluate the solution at any point on the interval of integration.

Function Description `deval` Evaluate the numerical solution using output of ODE solvers.

ODE Option Handling

An options structure contains named integration properties whose values are passed to the solver, and which affect problem solution. Use these functions to create, alter, or access an options structure.

Function Description `odeset` Create or alter options structure for input to ODE solvers. `odeget` Extract properties from options structure created with `odeset`.

ODE Solver Output Functions

If an output function is specified, the solver calls the specified function after every successful integration step. You can use `odeset` to specify one of these sample functions as the `OutputFcn` property, or you can modify them to create your own functions.

Function Description `odeplot` Time-series plot `odephas2` Two-dimensional phase plane plot `odephas3` Three-dimensional phase plane plot `odeprint` Print to command window

ODE Initial Value Problem Examples

These examples illustrate the kinds of problems you can solve in MATLAB. From the MATLAB Help browser, click the example name to see the code in an editor. Type the example name at the command line to run it.

Note The Differential Equations Examples browser enables you to view the code for the ODE examples and DAE examples. You can also run the examples from the browser. Click on these links to invoke the browser, or type `odeexamples('ode')` or `odeexamples('dae')` at the command line.

Example Description

`amp1dae` Stiff DAE – electrical circuit
`ballode` Simple event location – bouncing ball batonode
ODE with time- and state-dependent mass matrix – motion of a baton
`brussode`
`burgersode`

fem1ode
 fem2ode
 hb1dae
 hb1ode
 orbitode
 Stiff large problem – diffusion in a chemical reaction (the Brusselator)
 ODE with strongly state-dependent mass matrix – Burger's equation solved using a moving mesh technique
 Stiff problem with a time-dependent mass matrix – finite element method
 Stiff problem with a constant mass matrix – finite element method
 Stiff DAE from a conservation law
 Stiff problem solved on a very long interval – Robertson chemical reaction
 Advanced event location – restricted three body problem
Ex ample Description rigidode Nonstiff problem – Euler equations of a rigid body without external forces
 vdopode Parameterizable van der Pol equation (stiff for large ω)
Introduction to Initial Value ODE Problems

What Is an Ordinary Differential Equation

The MATLAB ODE solvers are designed to handle *ordinary differential equations*. An ordinary differential equation contains one or more derivatives of a dependent variable y with respect to a single independent variable t , usually referred to as *time*. The derivative of y with respect to t is denoted as y' , the second derivative as y'' , and so on. Often $y(t)$ is a vector, having elements $y_1 \hat{G} y_2 \hat{G} \dots \hat{G} y_n$.

Using Initial Conditions to Specify the Solution of Interest Generally there are many functions $y(t)$ that satisfy a given ODE, and additional information is necessary to specify the solution of interest. In an *initial value problem*, the solution of interest satisfies a specific *initial condition*, that is, y is equal to y_0 at a given initial time t_0 . An initial value problem for an ODE is then

$$\begin{aligned} y' &= f(t, y) \\ y|_{t=t_0} &= y_0 \quad (15 - 1) \end{aligned}$$

If the function $f(t, y)$ is sufficiently smooth, this problem has one and only one solution. Generally there is no analytic expression for the solution, so it is necessary to approximate $y(t)$ by numerical means, such as using one of the MATLAB ODE solvers.

Initial Value Problem Solvers

The MATLAB ODE solver functions implement numerical integration methods for solving IVPs for ODEs (Equation 15-1). Beginning at the initial time with initial conditions, they step through the time interval, computing a solution at each time step. If the solution for a time step satisfies the solver's error tolerance criteria, it is a successful step. Otherwise, it is a failed attempt; the solver shrinks the step size and tries again.

This section describes:

- MATLAB solvers for nonstiff ODE problems and solvers for stiff ODE problems
- ODE solver basic syntax
- Additional ODE solver arguments

“Mass Matrix and DAE Properties” on page 15-27 explains how to solve more general problems.

You can also use the MATLAB Help browser to get information on the syntax for any MATLAB function, as well as information on example files for these solvers.

Solvers for Nonstiff Problems

There are three solvers designed for nonstiff problems:

`ode45` Based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, `ode45` is the best function to apply as a “first try” for most problems.

`ode23` Based on an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude tolerances and in the presence of mild stiffness. Like `ode45`, `ode23` is a one-step solver.

`ode113` Variable order Adams-Basforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE function is particularly expensive to evaluate. `ode113` is a *multistep* solver – it normally needs the solutions at several preceding time points to compute the current solution.

Solvers for Stiff Problems

Not all difficult problems are stiff, but all stiff problems are difficult for solvers not specifically designed for them. Solvers for stiff problems can be used exactly like the other solvers. However, you can often significantly improve the efficiency of these solvers by providing them with additional information about the problem. (See “Improving ODE Solver Performance” on page 15-16.)

There are four solvers designed for stiff problems:

`ode15s` Variable-order solver based on the numerical differentiation formulas (NDFs). Optionally it uses the backward differentiation formulas, BDFs, (also known as Gear’s method). Like `ode113`, `ode15s` is a multistep solver. If you suspect that a problem is stiff or if `ode45` failed or was very inefficient, try `ode15s`.

`ode23s` Based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective.

`ode23t` An implementation of the trapezoidal rule using a “free” interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping.

`ode23tb` An implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that

is a trapezoidal rule step and a second stage that is a backward differentiation formula of order 2. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances.

ODE Solver Basic Syntax

All of the ODE solver functions share a syntax that makes it easy to try any of the different numerical methods if it is not apparent which is the most appropriate. To apply a different method to the same problem, simply change the ODE solver function name. The simplest syntax, common to all the solver functions, is

`[t,y] = solver(odefun,tspan,y0)`

where `solver` is one of the ODE solver functions listed previously. The basic input arguments are:

- `odefun` Function that evaluates the system of ODEs. It has the form

`dydt = odefun(t,y)` where `t` is a scalar, and `dydt` and `y` are column vectors. `tspan` Vector specifying the interval of integration. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`.

For `tspan` vectors with two elements [`t0 tf`], the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the MATLAB ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points.

Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.

`y0` Vector of initial conditions for the problem

See also “Introduction to Initial Value ODE Problems” on page 15-6.

The output arguments are:

- `t` Column vector of time points

`y` Solution array. Each row in `y` corresponds to the solution at a time returned in the corresponding row of `t`.

Additional ODE Solver Arguments

For more advanced applications, you can also specify as input arguments solver options and additional problem parameters.

`options` Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)`

“Improving ODE Solver Performance” on page 15-16 tells you how to create the structure and

describes the properties you can specify.

p1,p2... Parameters that the solver passes toodefun.
[t,y] = solver(odefun,tspan,y0,options,p1,p2...)

The solver passes any input parameters that follow the options argument toodefun and any functions you specify inoptions. Use options = [] as a placeholder if you set no options. The functionodefun must have the form

dydt = odefun(t,y,p1,p2,...)

See “Passing Additional Parameters to an ODE Function” on page 15-13 for an example.

Representing ODE Problems

This section describes the process for solving initial value ODE problems using one of the MATLAB ODE solvers. It uses the van der Pol equation:

- To describe the steps needed to solve an ODE problem using MATLAB
- As an example of a stiff ODE problem

It also tells you how to evaluate the solution at specific points.

Note See “ODE Solver Basic Syntax” on page 15-8 for more information.

Example: Solving an IVP ODE in MATLAB (van der Pol Equation, Nonstiff) This example explains and illustrates the steps you need to solve an initial value ODE problem using MATLAB.

1 Rewrite the Problem as a System of First-Order ODEs. ODEs often involve a number of dependent variables, as well as derivatives of order higher than one. To use the MATLAB ODE solvers, you must rewrite such equations as an equivalent system of first-order differential equations of the form

$$\dot{y} = f(t, y)$$

You can write any ordinary differential equation

$$y^{(n)} = f(t, y, \dot{y}, \dots, \ddot{y})$$

as a system of first-order equations by making the substitutions

$$\begin{aligned} y_1 &= y \\ y_2 &= \dot{y} \\ &\vdots \\ y_n &= y^{(n-1)} \end{aligned}$$

The result is an equivalent system of n first-order ODEs.

$$\dot{y}_1 = y_2$$

$$\dot{y}_2 = y_3$$

$$\dot{y}_n = f(t, y_1, y_2, \dots, y_{n-1})$$

For example, you can rewrite the van der Pol equation (second-order)

$$\ddot{y} - \mu^2 y^2 \dot{y} + y = 0$$

where μ is a scalar parameter, by making the substitution $y_1 = y$. The resulting system of first-order ODEs is

$$\dot{y}_1 = y_2$$

$$y$$

$$\begin{aligned}
 2 \\
 = \\
 \dot{x} \\
 1 \\
 - \\
 y \\
 2 \\
 _1 \dot{y}_2 - y_1
 \end{aligned}$$

2 Code the System of First-Order ODEs in MATLAB. Once you represent the equation as a system of first-order ODEs, you can code it as a function that a MATLAB ODE solver can use. The function must be of the form

`dydt = odefun(t,y)` Although `andy` must be the function's first two arguments, the function does not need to use them. The output `dydt`, a column vector, is the derivative of `y`.

The code below represents the van der Pol system in a MATLAB function, `vdp1`. The `vdp1` function assumes that $\mu = 1$. y_1 and y_2 become elements `y(1)` and `y(2)` of a two-element vector.

```
function dydt = vdp1(t,y)
dydt = [y(2); (1-y(1)^2)*y(2)-y(1)];
```

Note that, although `vdp1` must accept the arguments `t` and `y`, it does not use `t` in its computations.

3 Apply a Solver to the Problem. Decide which solver you want to use to solve the problem. Then call the solver and pass it the function you created to describe the first-order system of ODEs, the time interval on which you want to solve the problem, and an initial condition vector. See “Initial Value Problem Solvers” on page 15-6 and the ODE solver reference page for descriptions of the ODE solvers.

For the van der Pol system, you can use `ode45` on time interval `[0 20]` with initial values `y(1) = 2` and `y(2) = 0`.

```
[t,y] = ode45(@vdp1,[0 20],[2; 0]);
```

This example uses `@` to pass `vdp1` as a function handle to `ode45`. The resulting output is a column vector of time points `t` and a solution array `y`. Each row in `y` corresponds to a time returned in the corresponding row of `t`. The first column of `y` corresponds to y_1 , and the second column to y_2 .

Note See the `function_handle` (`@`), `func2str`, and `str2func` reference pages, and the Function Handles chapter of “Programming and Data Types” in the MATLAB documentation for information about function handles.

4 View the Solver Output. You can simply use the `plot` command to view the solver output.

```
plot(t,y(:,1),'-',t,y(:,2),'--')
title('Solution of van der Pol Equation, \mu = 1'); xlabel('time t'); ylabel('solution y');
legend('y_1','y_2')
```

Solution of van der Pol Equation, $\mu = 1$

```

y1 y2
2
1
0
-1
-2
-30 2 4 6 8 10 12 14 16 18 20 time t

```

As an alternative, you can use a solver output function to process the output. The solver calls the function specified in the integration property `OutputFcn` after each successful time step. Use `odeset` to set `OutputFcn` to the desired function. See “`OutputFcn`” on page 15-20 for more information.

Passing Additional Parameters to an ODEFunction. The solver passes any input parameters that follow the `options` argument to the ODE function and any function you specify in `options`. For example:

1 Generalize the van der Pol function by passing it a `mu` parameter, instead of specifying a value for `mu` explicitly in the code.

```

function dydt = vdp1(t,y,mu)
dydt = [y(2); mu*(1-y(1)^2)*y(2)-y(1)];

```

2 Pass the parameter `m` to the function `vdp1` by specifying it after the `options` argument in the call to the solver. This example uses `options = []` as a placeholder.

```

[t,y] = ode45(@vdp1,tspan,y0,[],mu)
calls
vdp1(t,y,mu)
See vdpode for a complete example based on these functions.

```

Example: The van der Pol Equation, $\approx = 1000$ (Stiff)

This example presents a stiff problem. For a stiff problem, solutions can change on a time scale that is very short compared to the interval of integration, but the solution of interest changes on a much longer time scale. Methods not designed for stiff problems are ineffective on intervals where the solution changes slowly because they use time steps small enough to resolve the fastest possible change.

When \approx is increased to 1000, the solution to the van der Pol equation changes dramatically and exhibits oscillation on a much longer time scale. Approximating the solution of the initial value problem becomes a more difficult task. Because this particular problem is stiff, a solver intended for nonstiff problems, such as `ode45`, is too inefficient to be practical. A solver such as `ode15s` is intended for such stiff problems.

The `vdp1000` function evaluates the van der Pol system with $\approx = 1000$.

```

function dydt =
vdp1000(t,y)
dydt = [y(2); 1000*(1-y(1)^2)*y(2)-y(1)];

```

Note This example hardcodes \approx in the ODE function. The `vdpode` example solves the same problem, but passes a user-specified \approx as an additional argument to the ODE function. See

“Additional ODE Solver Arguments” on page 15-10.

Now use the `ode15s` function to solve the problem with the initial condition vector of `[2; 0]`, but a time interval of `[0 3000]`. For scaling purposes, plot just the first component of $y(t)$.

```
[t,y] = ode15s(@vdp1000,[0 3000],[2; 0]);
plot(t,y(:,1),'-');
title('Solution of van der Pol Equation, \mu = 1000'); xlabel('time t');
ylabel('solution y_1');
```

```
Solution of van der Pol Equation, \mu = 1000
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
0 500 1000 1500 2000 2500 3000 time t
```

Note For detailed instructions for solving an initial value ODE problem in MATLAB, see “Example: Solving an IVP ODE in MATLAB (van der Pol Equation, Nonstiff)” on page 15-11.

Evaluating the Solution at Specific Points

The numerical methods implemented in the ODE solvers produce a continuous solution over the interval of integration $a \leq b$. You can evaluate the approximate solution, S , at any point in $a \leq b$ using the function `deval` and the structure `sol` returned by the solver.

```
Sxint = deval(sol,xint)
```

The ODE solvers return the structure `sol` when called with a single output argument.

The `deval` function is vectorized. For a vector `xint`, the i th column of `Sxint` approximates the solution $y(xint(i))$.

Improving ODE Solver Performance

The default integration properties in the ODE solvers are selected to handle common problems. In some cases, you can improve ODE solver performance by overriding these defaults. You do this by supplying the solvers with one or more property values in an options structure.

```
[t,y] = solver(odefun,tspan,y0,options)
```

This section:

- Explains how to create, modify, and query an options structure.
- Describes the properties that you can use in an options structure. In this and subsequent property tables, the most commonly used properties are listed first, followed by more advanced properties.

ODE Property Categories Properties Category Property Name Error control

RelTol,AbsTol,NormControl Solver output OutputFcn,OutputSel,Refine,Stats Jacobian matrix
Jacobian,JPattern,Vectorized Step-size InitialStep,MaxStep

ODE Property Categories Properties Category Property Name Mass matrix and DAEs

Event location ode15s-specific
Mass,MStateDependence,MvPattern, MassSingular, InitialSlope
Events
MaxOrder,BDF

Creating and Maintaining an ODE Options Structure

Creating an Options Structure. The odeset function creates an options structure that you can pass as an argument to any of the ODE solvers. To create an options structure, odeset accepts property name/property value pairs using the syntax

```
options = odeset('name1',value1,'name2',value2,...)
```

In the resulting structure, options, the named properties have the specified values. Any unspecified properties contain default values in the solvers. For all properties, it is sufficient to type only the leading characters that uniquely identify the property name. odeset ignores case for property names.

With no input arguments, odeset displays all property names and their possible values. It indicates defaults with {}.

Modifying an Existing Options Structure. To modify an existing options structure, oldopts, use

```
options = odeset(oldopts,'name1',value1,...)
```

This sets options equal to the existing structure oldopts, overwrites any values in oldopts that are respecified using name/value pairs, and adds any new pairs to the structure. The modified structure is returned as an output argument. In the same way, the command

```
options = odeset(oldopts,newopts)
```

combines the structures oldopts and newopts. In the output argument, any values in the second argument overwrite those in the first argument. **Querying Options.** The odeget function extracts property values from an options structure created with odeset.

```
o = odeget(options,'name')
```

This function returns the value of the specified property, or an empty matrix [], if the property value is unspecified in the options structure. As with odeset, it is sufficient to type only the leading characters that uniquely identify the property name. Case is ignored for property names.

Error Control Properties

At each step, the solver estimates the local error e in the i th component of the solution. This error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, RelTol, and the specified absolute tolerance, AbsTol.

$$|e(i)| \leq \max(\text{RelTol} * \text{abs}(y(i)), \text{AbsTol}(i))$$

For routine problems, the ODE solvers deliver accuracy roughly equivalent to the accuracy you request. They deliver less accuracy for problems integrated over “long” intervals and problems that are moderately unstable. Difficult problems may require tighter tolerances than the default values. For relative accuracy, adjust RelTol. For the absolute error tolerance, the scaling of the

solution components is important: if $|y|$ is somewhat smaller than AbsTol , the solver is not constrained to obtain any correct digits in y . You might have to solve a problem more than once to discover the scale of solution components.

Roughly speaking, this means that you want RelTol correct digits in all solution components except those smaller than $\text{thresholdsAbsTol}(i)$. Even if you are not interested in a component $y(i)$ when it is small, you may have to specify $\text{AbsTol}(i)$ small enough to get some correct digits in $y(i)$ so that you can accurately compute more interesting components. The following table describes the error control properties. Use `odeset` to set the properties.

ODE Error Control Properties **Property** **Value** **Description** **RelTol** Positive scalar {1e-3}

AbsTol Positive scalar or vector {1e-6}

NormControl on | {off}

A relative error tolerance that applies to all components of the solution vector. It is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components except those smaller than $\text{thresholdsAbsTol}(i)$.

The default, 1e-3, corresponds to 0.1% accuracy.

Absolute error tolerances that apply to the individual components of the solution vector. $\text{AbsTol}(i)$ is a threshold below which the value of the i th solution component is unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero. Even if you are not interested in a component $y(i)$ when it is small, you may have to specify $\text{AbsTol}(i)$ small enough to get some correct digits in $y(i)$ so that you can accurately compute more interesting components.

If AbsTol is a vector, the length of AbsTol must be the same as the length of the solution vector. If AbsTol is a scalar, the value applies to all components of y .

Control error relative to norm of solution. Set this property on to request that the solvers control the error in each integration step with $\text{norm}(e) \leq \max(\text{RelTol} * \text{norm}(y), \text{AbsTol})$. By default the solvers use a more stringent component-wise error control.

Solver Output Properties

The solver output properties let you control the output that the solvers generate. Use `odeset` to set these properties.

ODE Solver Output Properties **Property** **Value** **Description** **OutputFcn** Function {`odeplot`} Installable output function. The solver calls this function after every successful integration step.

For example, `options = odeset('OutputFcn',@myfun)`

sets the `OutputFcn` property to an output function, `myfun`, that can be passed to an ODE solver.

The output function must be of the form

```
status = myfun(t,y,flag)
```

The solver calls the specified output function with the following flags. Note that the syntax of the call differs with the flag. The function must respond appropriately:

init The solver calls `myfun(tspan,y0,'init')` before beginning the integration to allow the output function to initialize. `tspan` and `y0` are the input arguments to the ODE solver.

ODE Solver Output Properties (Continued) Property Value Description

{none} The solver calls `status = myfun(t,y)` after each

integration step on which output is requested. `t` contains points where output was generated during the step, and `y` is the numerical solution at the points `int`. If `t` is a vector, the `i`th column of `y` corresponds to the `i`th element of `t`.

When `length(tspan) > 2` the output is produced at every point `intspan`. When `length(tspan) = 2` the output is produced according to the Refine option.

`myfun` must return a `status` output value of 0 or 1. If `status = 1`, the solver halts integration. You can use this mechanism, for instance, to implement a **Stop** button.

done The solver calls `myfun([],[],'done')` when integration is complete to allow the output function to perform any cleanup chores.

You can use these general purpose output functions or you can edit them to create your own. Type `help` function at the command line for more information.

- `odeplot`— time series plotting (default when you call the solver with no output arguments and you have not specified an output function)
- `odephas2`— two-dimensional phase plane plotting
- `odephas3`— three-dimensional phase plane plotting
- `odeprint`— print solution as it is computed

Note If you call the solver with no output arguments, the solver does not allocate storage to hold the entire solution history.

ODE Solver Output Properties (Continued) Property Value Description `OutputSel` Vector of indices

`Refine` Positive integer Vector of indices specifying which components of the solution vector are to be passed to the output function. For example, if you want to use the `odeplot` output function, but you want to plot only the first and third components of the solution, you can do this using

```
options = odeset('OutputFcn',@odeplot,'OutputSel',[1 3]);
```

By default, the solver passes all components of the solution to the output function.

`Increases` the number of output points by a factor of `Refine`. If `Refine` is 1, the solver returns solutions only at the end of each time step. If `Refine` is $n > 1$, the solver subdivides each time step into n smaller intervals, and returns solutions at each time point. `Refine` does not apply

whenlength(tspan)>2.

Note In all the solvers, the default value of Refine is 1. Within ode45, however, the default is 4 to compensate for the solver's large step sizes. To override this and see only the time steps chosen by ode45, setRefine to 1.

The extra values produced for Refine are computed by means of continuous extension formulas. These are specialized formulas used by the ODE solvers to obtain accurate solutions between computed time steps without significant increase in computation time.

ODE Solver Output Properties (Continued) Property Value Description

Stats on | {off} Specifies whether the solver should display statistics about its computations. By default, Stats is off. If it is on, after solving the problem the solver displays:

- The number of successful steps
- The number of failed attempts
- The number of times the ODE function was called to evaluate $f(t, y)$
- The number of times that the partial derivatives matrix $\frac{\partial f}{\partial y}$ was formed
- The number of LU decompositions
- The number of solutions of linear systems

Jacobian Matrix Properties

The stiff ODE solvers often execute faster if you provide additional information about the Jacobian matrix $\frac{\partial f}{\partial y}$, a matrix of partial derivatives of the function that defines the differential equations.

$$\begin{aligned} & \frac{\partial f_1}{\partial y} \\ & \frac{\partial f_2}{\partial y} = \\ & \frac{\partial f_1}{\partial y} = \\ & \frac{\partial f_2}{\partial y} = \end{aligned}$$

The Jacobian matrix properties pertain only to those solvers for stiff problems (ode15s, ode23s, ode23t, and ode23tb) for which the Jacobian matrix $\frac{\partial f}{\partial y}$ can be critical to reliability and efficiency. If you do not provide a function to calculate the Jacobian, these solvers approximate the Jacobian numerically using finite differences. In this case, you may want to use the Vectorized, or JPatten properties.

The following table describes the Jacobian matrix properties. Use odeset to set these properties.

ODE Jacobian Matrix Properties Property Value Description

JPattern

Function |

constant matrix

Sparse matrix of {0,1}

A constant matrix or a function that evaluates the Jacobian. Supplying an analytical Jacobian often increases the speed and reliability of the solution for stiff problems. Set this property to a functionFJac, whereFJac(t,y) computes $\frac{\partial f}{\partial y}$, or to the constant value of $\frac{\partial f}{\partial y}$.

The Jacobian for the stiff van der Pol problem shown above can be coded as

```
function J = vdp1000jac(t,y)
```

```
J = [ 0 1
```

```
(-2000*y(1)*y(2)-1) (1000*(1-y(1)^2)) ];
```

Sparsity pattern with 1s where there might be nonzero entries in the Jacobian. It is used to generate a sparse Jacobian matrix numerically.

Set this property to a sparse matrix S with $S(i,j) = 1$ if component i of $f(t,y)$ depends on component j of y , and 0 otherwise. The solver uses this sparsity pattern to generate a sparse Jacobian matrix numerically. If the Jacobian matrix is large and sparse, this can greatly accelerate execution. For an example using theJPattern property, see “Example: Large, Stiff Sparse Problem” on page 15-38 (brussode).

ODE Jacobian Matrix Properties (Continued) Property Value Description

Vectorized on | {off} Seton to inform the solver that you have coded the ODE functionF so thatF(t,[y1 y2 ...]) returns [F(t,y1) F(t,y2) ...]. This allows the solver to reduce the number of function evaluations required to compute all the columns of the Jacobian matrix, and may significantly reduce solution time.

With MATLAB’s array notation, it is typically an easy matter to vectorize an ODE function. For example, the stiff van der Pol example shown previously can be vectorized by introducing colon notation into the subscripts and by using the array power (.^) and array multiplication (.*) operators.

```
function dydt = vdp1000(t,y)
dydt = [y(2,:); 1000*(1-y(1,:).^2).*y(2,:)-y(1,:)];
```

Note Vectorization of the ODE function used by the ODE solvers differs from the vectorization used by the BVP solver, bvp4c. For the ODE solvers, the ODE function is vectorized only with respect to the second argument, whilebvp4c requires vectorization with respect the first and second arguments.

Step-Size Properties

The step-size properties let you specify the size of the first step the solver tries, potentially helping it to better recognize the scale of the problem. In addition, you can specify bounds on the sizes of subsequent time steps.

The following table describes the step-size properties. Useodeset to set these properties.

ODE Step Size Properties	Property	Value	Description
	InitialStep	Positive scalar	
	MaxStep	Positive scalar	{0.1*fabs(t0-tf)}

Suggested initial step size. InitialStep sets an upper bound on the magnitude of the first step size the solver tries. If you do not setInitialStep, the initial step size is based on the slope of the solution at the initial time `tspan(1)`, and if the slope of all solution components is zero, the procedure might try a step size that is much too large. If you know this is happening or you want to be sure that the solver resolves important behavior at the start of the integration, help the code start by providing a suitable InitialStep.

Upper bound on solver step size. If the differential equation has periodic coefficients or solutions, it may be a good idea to setMaxStep to some fraction (such as 1/4) of the period. This guarantees that the solver does not enlarge the time step too much and step over a period of interest. Do *not* reduce MaxStep:

- To produce more output points. This can significantly slow down solution time. Instead, useRefineto compute additional outputs by continuous extension at very low cost.
- When the solution does not appear to be accurate enough. Instead, reduce the relative error tolerance RelTol, and use the solution you just computed to determine appropriate values for the absolute error tolerance vectorAbsTol. (See “Error Control Properties” on page 15-18 for a description of the error tolerance properties.)

ODE Step Size Properties (Continued) Property Value Description

- To make sure that the solver doesn’t step over some behavior that occurs only once during the simulation interval. If you know the time at which the change occurs, break the simulation interval into two pieces and call the solvers twice. If you do not know the time at which the change occurs, try reducing the error tolerancesRelTol and AbsTol. Use MaxStep as a last resort.

Mass Matrix and DAE Properties

The solvers of the ODE suite can solve ODEs of the form

$$M(t \hat{G} y) y' = f(t \hat{G} y) \quad (15 - 2)$$

with a mass matrix $M \hat{f} t \hat{G} y'$ that can be sparse.

When $M \hat{f} t \hat{G} y'$ is nonsingular, the equation above is equivalent to $y' = M^{-1} f \hat{f} t \hat{G} y'$ and the ODE has a solution for any initial values y_0 at t_0 . The more general form (Equation 15-2) is convenient when you express a model naturally in terms of a mass matrix. For large, sparse $M \hat{f} t \hat{G} y'$, solving Equation 15-2 directly reduces the storage and runtime needed to solve the problem.

When $M \hat{f} t \hat{G} y'$ is singular, then $M(t \hat{G} y) y' = f(t \hat{G} y)$ is a differential-algebraic equation (DAE). A DAE has a solution only when y_0 is consistent, that is, there

exists an initial slope $yp0$ such that $M(t0 \hat{G} y0) f(t0 \hat{G} y0) = f(t0 \hat{G} y0)$. If $y0$ and $yp0$ are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. For DAEs of index 1, solving an initial value problem with consistent initial conditions is much like solving an ODE.

The `ode15s` and `ode23t` solvers can solve DAEs of index 1. For examples of DAE problems, see `hb1dae`(“Example: Differential-Algebraic Problem” on page 15-47) and `amp1dae`.

The following table describes the mass matrix and DAE properties. Use `odeset` to set these

properties.

ODE Mass Matrix and DAE Properties

Property	Value Description
Mass	
MStateDependence	Constant matrix function
	none {weak} strong
MvPattern	Sparse matrix
	Constant mass matrix or a function that evaluates the mass matrix $M(t)\mathbf{y}$. For problems

$M(t)\mathbf{y} = f(t)\mathbf{y}$ set this property to the value of the constant mass matrix M . For problems $M(t)\mathbf{y} = f(t)\mathbf{y}$, set this property to a function Mfun, where Mfun(t, \mathbf{y}) evaluates the mass matrix $M(t)\mathbf{y}$. When solving DAEs, it is advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE). The ode23s solver can only solve problems with a constant mass matrix M .

For example problems, see fem1ode (“Example: Finite Element Discretization” on page 15-35), fem2ode, or batonode.

Dependence of the mass matrix on \mathbf{y} . Set this property to none for problems $M(t)\mathbf{y} = f(t)\mathbf{y}$. Both weak and strong indicate $M(t)\mathbf{y}$, but weak results in implicit solvers using approximations when solving algebraic equations.

Set this property to a sparse matrix S with $S[i,j] = 1$ if for any k , the i th component of $M(t)\mathbf{y}$ depends on component j of \mathbf{y} , and 0 otherwise. For use with the ode15s, ode23t, and ode23tb solvers when MStateDependence is strong. See burgersode as an example.

ODE Mass Matrix and DAE Properties (Continued)

Property	Value Description
MassSingular	

InitialSlope yes | no | {maybe}

Vector |

{zero vector} Indicates whether the mass matrix is singular. Set this property to no if the mass matrix is not singular and you are using either theode15s or ode23t solver. The default value of maybe causes the solver to test whether the problem is a DAE, i.e., whether $M(t_0)\mathbf{y}_0$ is singular.

For an example of a problem with a mass matrix, see “Example: Finite Element Discretization” on page 15-35 (fem1ode).

Vector representing the consistent initial slope \mathbf{y}_{p0} , where \mathbf{y}_{p0} satisfies $M(t_0)\mathbf{y}_{p0} = f(t_0)\mathbf{y}_{p0}$. The default is the zero vector.

This property is for use with theode15s and ode23t solvers when solving DAEs.

Event Location Property

In some ODE problems the times of specific events are important, such as the time at which a ball hits the ground, or the time at which a spaceship returns to the earth. While solving a problem, the MATLAB ODE solvers can detect such events by locating transitions to, from, or through zeros of user-defined functions.

The following table describes theEvents property. Useodeset to set this property.

ODE Events Property String Value Description

Events Function MATLAB function that includes one or more event functions. The MATLAB function is of the form

[value,isterminal,direction] = events(t,y) value, isterminal, and direction are vectors for which the i th element corresponds to the i th event function:

- value(i) is the value of the i th event function.
- isterminal(i) = 1 if the integration is to terminate at a zero of this event function and 0 otherwise.
- direction(i) = 0 if all zeros are to be located (the default), +1 if only zeros where the event function is increasing, and -1 if only zeros where the event function is decreasing.

When you specify an events function, the solver returns three additional outputs, TE, YE, and IE.

options = odeset('Events', @events)

[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)

- TE is a column vector of times at which events occur.
- Rows of YE are the solution values corresponding to times in TE.
- Indices in vector IE specify which event occurred at the time in TE.

For examples that use an event function, see “Example: Simple Event Location” on page 15-41 (ballode) and “Example: Advanced Event Location” on page 15-44 (orbitode).

ode15s Properties

ode15s is a variable-order solver for stiff problems. It is based on the numerical differentiation formulas (NDFs). The NDFs are generally more efficient than the closely related family of backward differentiation formulas (BDFs), also known as Gear’s methods. Theode15s properties let you choose between these formulas, as well as specifying the maximum order for the formula used.

The following table describes theode15s properties. Useodeset to set these properties.

ode 15s Properties Property Value Description MaxOrder 1 | 2 | 3 | 4 | {5} The maximum order formula used to compute the solution. BDF on | {off} Specifies whether you want to use the BDFs instead of the default NDFs. SetBDF on to have ode15s use the BDFs.

For both the NDFs and BDFs, the formulas of orders 1 and 2 are A-stable (the stability region includes the entire left half complex plane). The higher order formulas are not as stable, and the higher the order the worse the stability. There is a class of stiff problems (stiff oscillatory) that is solved more efficiently ifMaxOrder is reduced (for example to 2) so that only the most stable formulas are used.

Examples: Applying the ODE Initial Value Problem Solvers

This section contains several examples that illustrate the kinds of problems you can solve in MATLAB:

- Simple nonstiff problem (`rigidode`)
- Stiff problem (`vdpode`)
- Finite element discretization (`fem1ode`)
- Large, stiff sparse problem (`brussode`)
- Simple event location (`ballode`)
- Advanced event location (`orbitode`)
- Differential-algebraic problem (`hb1dae`)

Example: Simple Nonstiff Problem

`rigidode` illustrates the solution of a standard test problem proposed by Krogh for solvers intended for nonstiff problems [7].

The ODEs are the Euler equations of a rigid body without external forces.

$$\begin{aligned}y_1' &= y_2 y_3 \\y_2' &= -y_1 y_3 \\y_3' &= -0.51 y_1 y_2\end{aligned}$$

For your convenience, the entire problem is defined and solved in a single M-file. The differential equations are coded as a subfunction. Because the example calls the `ode45` solver without output arguments, the solver uses the default output function `odeplot` to plot the solution components.

To run this example from the MATLAB Help browser, click on the example name. Otherwise, type `rigidode` at the command line.

```
function rigidode
%RIGIDODE Euler equations of a rigid body without external forces
tspan = [0 12];
y0 = [0; 1; 1];
```

```
% Solve the problem using ode45
ode45(@f,tspan,y0);
```

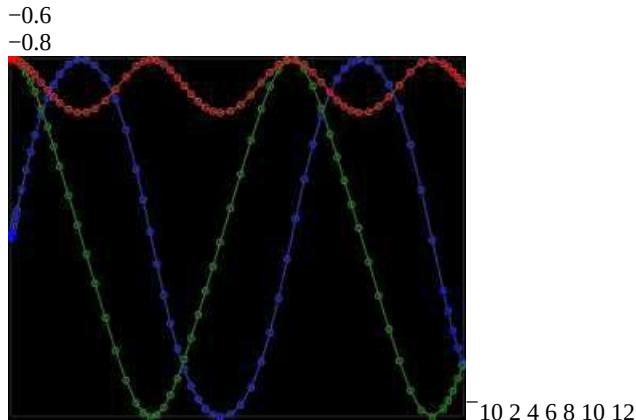
```
%
```

```
function dydt = f(t,y)
dydt = [ y(2)*y(3)
```

```
-y(1)*y(3)
```

```
-0.51*y(1)*y(2) ];
```

1
0.8
0.6
0.4
0.2
0
-0.2
-0.4



Example: Stiff Problem (van der Pol Equation)

vdpode illustrates the solution of the van der Pol problem described in “Example: The van der Pol Equation, $\mu = 1000$ (Stiff)” on page 15-14. The differential equations

$$\begin{aligned} y_1' &= y_2 \\ y \\ 2 \\ | \\ = \\ \mu \\ (1 \\ - \\ y \\ 2 \\ 1)y_2 - y_1 \end{aligned}$$

involve a constant parameter μ .

As μ increases, the problem becomes more stiff, and the period of oscillation becomes larger. When μ is 1000 the equation is in relaxation oscillation and the problem is very stiff. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff (quasi-discontinuities).

By default, the solvers in the ODE suite that are intended for stiff problems approximate Jacobian matrices numerically. However, this example provides a subfunction $J(t,y,\mu)$ to evaluate the Jacobian matrix $\frac{\partial f}{\partial y}$ analytically at (t,y) for $\mu = \mu$. The use of an analytic Jacobian can improve the reliability and efficiency of integration.

To run this example from the MATLAB Help browser, click on the example name. Otherwise, type `vdpode` at the command line. At the command line, you can specify a value of μ as an argument to `vdpode`. The default is $\mu = 1000$.

```
function vdpode(MU)
%VDPODE Parameterizable van der Pol equation (stiff for large MU) if nargin < 1
```

`MU = 1000;` % default

```

end

tspan = [0; max(20,3*MU)]; % Several periods y0 = [2; 0];
options = odeset('Jacobian',@J);

[t,y] = ode15s(@f,tspan,y0,options,MU);

plot(t,y(:,1));
title(['Solution of van der Pol Equation, \mu = ' num2str(MU)]); xlabel('time t');
ylabel('solution y_1');

axis([tspan(1) tspan(end) -2.5 2.5]);
function dydt = f(t,y,mu)
dydt = [ y(2)
mu*(1-y(1)^2)*y(2)-y(1) ]; function dfdy = J(t,y,mu)
dfdy = [ 0 1
-2*mu*y(1)*y(2)-1 mu*(1-y(1)^2) ];

Solution of van der Pol Equation, \mu = 1000
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.50 500 1000 1500 2000 2500 3000 time t

```

Example: Finite Element Discretization

fem1ode illustrates the solution of ODEs that result from a finite element discretization of a partial differential equation. The value of N in the call fem1ode(N) controls the discretization, and the resulting system consists of N equations. By default, N is 19.

This example involves a mass matrix. The system of ODEs comes from a method of lines solution of the partial differential equation

$$-\frac{t^3 u''}{8} = \frac{u''}{t^3 x^2}$$

with initial condition $u(0) = \sin x$ and boundary conditions $u(t=0) = u(t=\pi) = 0$. An integer N is chosen, h is defined as $\sqrt{\frac{d}{c}} N + 1$, and the solution of the partial differential equation is approximated at $x_k = kh$ for

$k = 0, 1, \dots, N+1$ by

N

$$u(t, x_k) \star c_k \delta_{k1}$$

Here δ_{kj} is a piecewise linear function that is 1 at x_k and 0 at all the other x_j . A Galerkin discretization leads to the system of ODEs

$c_1 M \hat{f} = J c$ where $c \hat{f} =$
 $c_N \hat{f}$ and the tridiagonal matrices $M \hat{f}$ and J are given by
 $\begin{cases} 2h_c^d 3 \exp(-t) & \text{if } i = j \\ M_{ij} = \frac{d}{c} 6 \exp(-t) & \text{if } i = j^{238} \\ 0 & \text{otherwise} \end{cases}$

and

$$J_{ij} =$$

$$\begin{cases} -2\frac{d}{c} h & \text{if } i = j \\ 1\frac{d}{c} h & \text{if } i = j^{238} \\ 0 & \text{otherwise} \end{cases}$$

The initial values $c \hat{f}_0$ are taken from the initial condition for the partial differential equation.
The problem is solved on the time interval $[0, \sqrt{8}]$.

In fem1ode the properties

```
options = odeset('Mass',@mass,'MStateDep','none','Jacobian',J)
```

indicate that the problem is of the form $M(t)y' = Jy$. The subfunction mass(t,N) evaluates the time-dependent mass matrix $M \hat{f}$ and J is the constant Jacobian.

To run this example from the MATLAB Help browser, click on the example name. Otherwise, type fem1ode at the command line. At the command line, you can specify a value of N as an argument to fem1ode. The default is $N = 19$.

```
function fem1ode(N)
%FEM1ODE Stiff problem with a time-dependent mass matrix if nargin < 1

N = 19;
end
h = pi/(N+1);
y0 = sin(h*(1:N)');
tspan = [0; pi];

% The Jacobian is constant.
e = repmat(1/h,N,1); % e=[(1/h) ... (1/h)];
d = repmat(-2/h,N,1); % d=[(-2/h) ... (-2/h)];
J = spdiags([e d e], -1:1, N, N);

options = odeset('Mass',@mass,'MStateDependence','none', ...
    'Jacobian',J);
[t,y] = ode15s(@f,tspan,y0,options,N);

surf((1:N)/(N+1),t,y);
set(gca,'ZLim',[0 1]);
view(142.5,30);
title(['Finite element problem with time-dependent mass ' ...
    'matrix, solved by ODE15S']);
xlabel('space ( x/pi )');
ylabel('time');
```

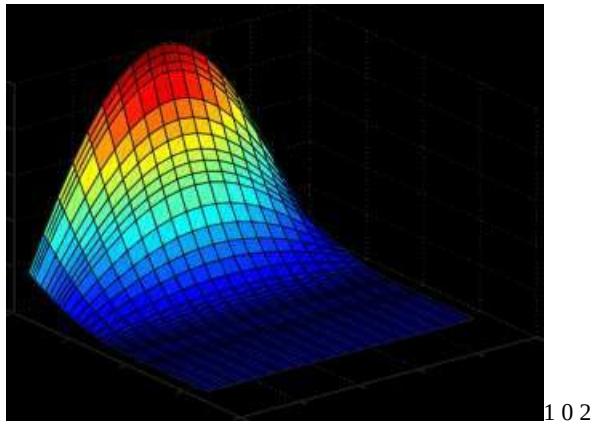
```

zlabel('solution');
%-----function out = f(t,y,N)
h = pi/(N+1);
e = repmat(1/h,N,1); % e=[(1/h) ... (1/h)];
d = repmat(-2/h,N,1); % d=[(-2/h) ... (-2/h)];
J = spdiags([e d e], -1:1, N, N);
out = J*y;
%-----function M = mass(t,N)
h = pi/(N+1);
e = repmat(exp(-t)*h/6,N,1); % e(i)=exp(-t)*h/6
e4 = repmat(4*exp(-t)*h/6,N,1);
M = spdiags([e e4 e], -1:1, N, N);

1
0.8
0.6
0.4
0.2

0
0
Finite element problem with time-dependent mass matrix, solved by ODE15S

```



0.2 0.4 0.6
0.8
4 1
time space (x/ ✓)

Example: Large, Stiff Sparse Problem

brussode illustrates the solution of a (potentially) large stiff sparse problem. The problem is the classic “Brusselator” system [3] that models diffusion in a chemical reaction

$$\begin{aligned} u_i'' &= 4u_i + \tilde{\alpha}N + 1f^2u_{i-1} - 2u_i + u_{i+1} \\ v_i'' &= 3u_i - u^2v_i + \tilde{\alpha}N + 1f^2v_{i-1} - 2v_i + v_{i+1} \end{aligned}$$

and is solved on the time interval [0,10] with $\tilde{\alpha} = 1/50$ and $u_i(0) = 1 + \sin(2\sqrt{x_i})$ with $x_i = i_c^d/N$
 $+ 1fG$ for $i = 1, \dots, N$

There are $2N$ equations in the system, but the Jacobian is banded with a constant width 5 if the equations are ordered as $u_1 \hat{\rightarrow} v_1 \hat{\rightarrow} u_2 \hat{\rightarrow} v_2 \hat{\rightarrow}$

In the call `brussode(N)`, where N corresponds to N , the parameter $N \geq 2$ specifies the number of grid points. The resulting system consists of $2N$ equations. By default, N is 20. The problem becomes increasingly stiff and the Jacobian increasingly sparse as N increases.

The subfunction `f(t,y,N)` returns the derivatives vector for the Brusselator problem. The subfunction `jpattern(N)` returns a sparse matrix of 1s and 0s showing the locations of nonzeros in the Jacobian $\frac{d}{dt}f \frac{d}{dy}$. The example assigns this matrix to the property `JPattern`, and the solver uses the sparsity pattern to generate the Jacobian numerically as a sparse matrix. Providing a sparsity pattern can significantly reduce the number of function evaluations required to generate the Jacobian and can accelerate integration. For the Brusselator problem, if the sparsity pattern is not supplied, $2N$ evaluations of the function are needed to compute the $2N$ -by- $2N$ Jacobian matrix. If the sparsity pattern is supplied, only four evaluations are needed, regardless of the value of N .

To run this example from the MATLAB Help browser, click on the example name. Otherwise, type `brussode` at the command line. At the command line, you can specify a value of N as an argument to `brussode`. The default is $N = 20$.

```

function brussode(N)
%BRUSSODE Stiff problem modeling a chemical reaction
if nargin<1 N = 20; end
tspan = [0; 10];
y0 = [1+sin((2*pi/(N+1))*(1:N)); repmat(3,1,N)];
options = odeset('Vectorized','on','JPattern',jpattern(N));
[t,y] = ode15s(@f,tspan,y0,options,N);

u = y(:,1:2:end); x = (1:N)/(N+1); surf(x,t,u);
view(-40,30);
xlabel('space'); ylabel('time'); zlabel('solution u');
title(['The Brusselator for N = ' num2str(N)]);
%
function dydt = f(t,y,N)
c = 0.02 * (N+1)^2;
dydt = zeros(2*N,size(y,2)); % preallocate dy/dt % Evaluate the two components of the function
at one edge of % the grid (with edge conditions).
i = 1;
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...
c*(1-2*y(i,:)+y(i+2,:));
dydt(i+1,:) = 3*y(i,:)-y(i+1,:).*y(i,:).^2 + ...
c*(3-2*y(i+1,:)+y(i+3,:));

% Evaluate the two components of the function at all interior % grid points.
i = 3:2:N-3;
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...

```

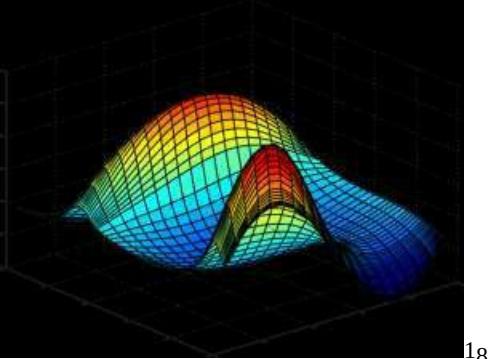
```

c*(y(i-2,:)-2*y(i,:)+y(i+2,:));
dydt(i+1,:) = 3*y(i,:)-y(i+1,:).*y(i,:).^2 + ...
c*(y(i-1,:)-2*y(i+1,:)+y(i+3,:));

% Evaluate the two components of the function at the other edge % of the grid (with edge
conditions).
i = 2*N-1;
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...

c*(y(i-2,:)-2*y(i,:)+1);
dydt(i+1,:) = 3*y(i,:)-y(i+1,:).*y(i,:).^2 + ...
c*(y(i-1,:)-2*y(i+1,:)+3);

%
function S = jpattern(N)
B = ones(2*N,5);
B(2:2:2*N,2) = zeros(N,1);
B(1:2:2*N-1,4) = zeros(N,1);
S = spdiags(B,-2:2,2*N,2*N);

3
2.5
2
1.5
1
0.5
0 10 The Brusselator for N = 20

time 0 0 space 0.8

```

Example: Simple Event Location

ballode models the motion of a bouncing ball. This example illustrates the event location capabilities of the ODE solvers.

The equations for the bouncing ball are

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -9.8 \end{aligned}$$

In this example, the event function is coded in a subfunction `events` [value,isterminal,direction] = `events(t,y)`

which returns:

- A value of the event function

- The information whether or not the integration should stop (isterminal = 1 or 0, respectively)
when value = 0

- The desired directionality of the zero crossings:

-1 Detect zero crossings in the negative direction only

0 Detect all zero crossings

1 Detect zero crossings in the positive direction only

The length of value, isterminal, and direction is the same as the number of event functions. The *i*th element of each vector, corresponds to the *i*th event function. For an example of more advanced event location, see `orbitode` (“Example: Advanced Event Location” on page 15-44).

In `ballode`, setting the `Events` property to @`events` causes the solver to stop the integration (isterminal = 1) when the ball hits the ground (the height(1) is 0) during its fall (direction = -1). The example then restarts the integration with initial conditions corresponding to a ball that bounced.

To run this example from the MATLAB Help browser, click on the example name. Otherwise, type `ballode` at the command line.

```
function ballode
```

```
%BALLODE Run a demo of a bouncing ball.
```

```
tstart = 0;
tfinal = 30;
y0 = [0; 20];
refine = 4;
options = odeset('Events',@events,'OutputFcn', @odeplot,...
    'OutputSel',1,'Refine',refine);

set(gca,'xlim',[0 30],'ylim',[0 25]); box on
hold on;

tout = tstart;
yout = y0.';
teout = [];
yeout = [];
ieout = [];
for i = 1:10

    % Solve until the first terminal event.
    [t,y,te,ye,ie] = ode23(@f,[tstart tfinal],y0,options); if ~ishold
        hold on
    end
```

```

end
% Accumulate output.
nt = length(t);
tout = [tout; t(2:nt)];
yout = [yout; y(2:nt,:)];
teout = [teout; te]; % Events at tstart are never reported. yeout = [yeout; ye];
ieout = [ieout; ie];

ud = get(gcf,'UserData'); if ud.stop
break;
end

% Set the new initial conditions, with .9 attenuation. y0(1) = 0;
y0(2) = -.9*y(nt,2);

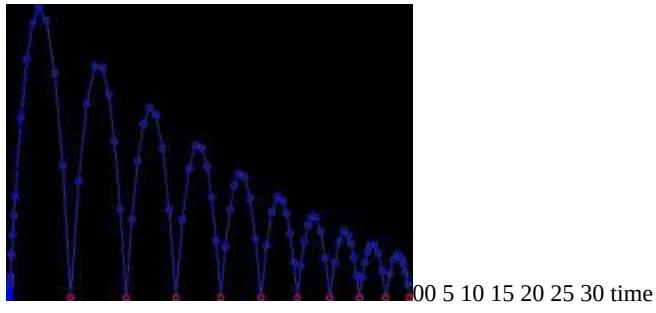
% A good guess of a valid first time step is the length of % the last valid time step, so use it for
faster computation. options = odeset(options,'InitialStep',t(nt)-t(nt-refine),...

'MaxStep',t(nt)-t(1));
tstart = t(nt);
end

plot(teout,yeout(:,1),'ro')
xlabel('time');
ylabel('height');
title('Ball trajectory and the events');
hold off
odeplot([],[],'done');
%
function dydt = f(t,y)
dydt = [y(2); -9.8];
%
function [value,isterminal,direction] = events(t,y) % Locate the time when height passes through
zero in a % decreasing direction and stop integration. value = y(1); % Detect height = 0
isterminal = 1; % Stop the integration direction = -1; % Negative direction only

Ball trajectory and the events 25
20
15
10
5

```



Example: Advanced Event Location

orbitode illustrates the solution of a standard test problem for those solvers that are intended for nonstiff problems. It traces the path of a spaceship traveling around the moon and returning to the earth. (Shampine and Gordon [7], p.246).

The orbitode problem is a system of the four equations shown below

$$y_1^{\dot{}} = y_3$$

$$y_2^{\dot{}} = y_4$$

$$\nabla^{\circ} \mathbf{F} \mathbf{f}$$

$$\begin{aligned} y \\ \nabla^{\circ} \mathbf{F} \mathbf{f}_1 + \nabla^{\circ} \mathbf{f} &= \nabla^{\circ} \mathbf{F} y_1 - y_3^{\dot{}} = 2y_4 + y_1 - 3 - \frac{r_3 r_1}{r_3^3 r_1^2} \\ \nabla^{\circ} \mathbf{F} y_2 &= y_2^{\dot{}} \\ y_4^{\dot{}} &= -2y_3 + y_2 - \frac{3}{r_3^3} \\ r_1 &= \sqrt{1 - \frac{y_1^2 + y_2^2}{r_3^2}} \end{aligned}$$

where $\nabla^{\circ} = 1 \text{e-82.45}$

$$\nabla^{\circ} \mathbf{F} = 1 - \nabla^{\circ}$$

$$r_1 =$$

$$\sqrt{1 - \frac{y_1^2 + y_2^2}{r_3^2}}$$

$$y_1$$

$$1$$

$$+ \nabla^{\circ} \mathbf{f}_1^2 + y_2^2$$

$$r_2 =$$

$$\sqrt{1 - \frac{y_1^2 + y_2^2}{r_3^2}}$$

$$y_1$$

$$1$$

$$- \nabla^{\circ} \mathbf{f}_1^2 + y_2^2$$

The first two solution components are coordinates of the body of infinitesimal mass, so plotting one against the other gives the orbit of the body. The initial conditions have been chosen to make the orbit periodic. The value of ∇° corresponds to a spaceship traveling around the moon and the earth. Moderately stringent tolerances are necessary to reproduce the qualitative behavior of the orbit. Suitable values are 1e-5 for RelTol and 1e-4 for AbsTol.

The events subfunction includes event functions which locate the point of maximum distance from the starting point and the time the spaceship returns to the starting point. Note that the events are located accurately, even though the step sizes used by the integrator are *not* determined by the location of the events. In this example, the ability to specify the direction of the zero crossing is critical. Both the point of return to the initial point and the point of maximum distance have the same event function value, and the direction of the crossing is used to distinguish them.

To run this example from the MATLAB Help browser, click on the example name. Otherwise, type orbitode at the command line. The example uses the output function nodephase2 to produce the two-dimensional phase plane plot and let you see the progress of the integration.

```

function orbitode
%ORBITODE Restricted three-body problem

tspan = [0 7];
y0 = [1.2; 0; 0; -1.04935750983031990726];
options = odeset('RelTol',1e-5,'AbsTol',1e-4, ...

'OutputFcn',@odephas2,'Events',@events);
[t,y,te,ye,ie] = ode45(@f,tspan,y0,options);

plot(y(:,1),y(:,2),ye(:,1),ye(:,2),'o');
title ('Restricted three body problem')
ylabel ('y(t)')
xlabel ('x(t)')
%
function dydt = f(t,y)
mu = 1 / 82.45;
mustar = 1 - mu;
r13 = ((y(1) + mu)^2 + y(2)^2) ^ 1.5;
r23 = ((y(1) - mustar)^2 + y(2)^2) ^ 1.5;
dydt = [ y(3)

y(4)
2*y(4) + y(1) - mustar*((y(1)+mu)/r13) - ... mu*((y(1)-mustar)/r23)

-2*y(3) + y(2) - mustar*(y(2)/r13) - mu*(y(2)/r23) ]; %

function [value,isterminal,direction] = events(t,y)

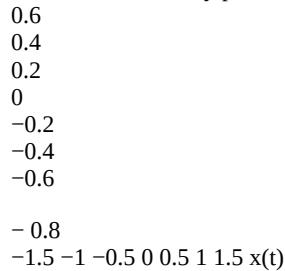
% Locate the time when the object returns closest to the % initial point y0 and starts to move
away, and stop integration. % Also locate the time when the object is farthest from the % initial
point y0 and starts to move closer.
%
% The current distance of the body is
%
% DSQ = (y(1)-y0(1))^2 + (y(2)-y0(2))^2
% = <y(1:2)-y0,y(1:2)-y0>
```

```

%
% A local minimum of DSQ occurs when d/dt DSQ crosses zero % heading in the positive
direction. We can compute d(DSQ)/dt as %
% d(DSQ)/dt = 2*(y(1:2)-y0)'*dy(1:2)/dt = 2*(y(1:2)-y0)'*y(3:4) %
y0 = [1.2; 0];
dDSQdt = 2 * ((y(1:2)-y0)' * y(3:4));
value = [dDSQdt; dDSQdt];
isterminal = [1; 0]; % Stop at local minimum direction = [1; -1]; % [local minimum, local
maximum]

```

Restricted three body problem 0.8



Example: Differential-Algebraic Problem

hb1dae reformulates the hb1ode example as a *differential-algebraic equation* (DAE) problem. The Robertson problem coded inhb1ode is a classic test problem for codes that solve stiff ODEs.

$$y_1' = -0.04y_1 + 10^4 y_2 y_3$$

$$\begin{aligned} y_2 &= 0.04 \\ y_1 &= -10 \\ &\quad 4y_2 y_3 - 3 \cdot 10^7 y_2^2 \\ y_3 &= 3 \\ &\quad 10 \\ &\quad 7y_2^2 \end{aligned}$$

Note The Robertson problem appears as an example in the prolog to LSODI [4].

In hb1ode, the problem is solved with initial conditions $y_1(0) = 1$, $y_2(0) = 0$, $y_3(0) = 0$ to steady state. These differential equations satisfy a linear conservation law that is used to reformulate the

problem as the DAE

$$\begin{aligned} y_1' &= -0.04y_1 + 10^4 y_2 y_3 \\ y_2' &= \\ | &= \\ = & \\ 0.04 & \\ y_1' &= \\ - & \\ 10 & \\ 4y_2 y_3 - 3 \cdot 10^7 y_2^2 &= y_1 + y_2 + y_3 - 1 \end{aligned}$$

Obviously these equations do not have a solution for $y(0)$ with components that do not sum to 1. The problem has the form of $M y' = f(t) G y$ with

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

M is obviously singular, but hb1dae does not inform the solver of this. The solver must recognize that the problem is a DAE, not an ODE. Similarly, although consistent initial conditions are obvious, the example uses an inconsistent value $y_3(0) = 10^{-3}$ to illustrate computation of consistent initial conditions.

To run this example from the MATLAB Help browser, click on the example name. Otherwise, type hb1dae at the command line. Note that hb1dae:

- Imposes a much smaller absolute error tolerance on y_2 than on the other components. This is because y_2 is much smaller than the other components and its major change takes place in a relatively short time.
- Specifies additional points at which the solution is computed to more clearly show the behavior of y_2 .
- Multiplies y_2 by 10^4 to make y_2 visible when plotting it with the rest of the solution.
- Uses a logarithmic scale to plot the solution on the long time interval.

```
function hb1dae
%HB1DAE Stiff differential-algebraic equation (DAE)
```

```
% A constant, singular mass matrix M = [1 0 0
0 1 0
0 0 0];
```

```
% Use an inconsistent initial condition to test initialization. y0 = [1; 0; 1e-3];
tspan = [0 4*logspace(-6,6)];
```

```
% Use the LSODI example tolerances. The 'MassSingular' property % is left at its default
'maybe' to test the automatic detection % of a DAE.
options = odeset('Mass',M,'RelTol',1e-4,...
```

```

'AbsTol',[1e-6 1e-10 1e-6],'Vectorized','on');
[t,y] = ode15s(@f,tspan,y0,options);
y(:,2) = 1e4*y(:,2);

semilogx(t,y);
ylabel('1e4 * y(:,2)');
title(['Robertson DAE problem with a Conservation Law, '...
'solved by ODE15S']);

xlabel('This is equivalent to the stiff ODEs coded in HB1ODE.');
function out = f(t,y)
out = [ -0.04*y(1,:)+1e4*y(2,:).*y(3,:)
        0.04*y(1,:)-1e4*y(2,:).*y(3,:)-3e7*y(2,:).^2
        y(1,:)+y(2,:)+y(3,:)-1];

```

Robertson DAE problem with a Conservation Law, solved by ODE15S

1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1

$10^{-6} \quad 10^{-4} \quad 10^{-2} \quad 10^0 \quad 10^2 \quad 10^4 \quad 10^6 \quad 10^8$

This is equivalent to the stiff ODEs coded in HB1ODE.

Questions and Answers, and Troubleshooting

This section contains a number of tables that answer questions about the use and operation of the MATLAB ODE solvers:

- General ODE solver questions
- Problem size, memory use, and computation speed
- Time steps for integration
- Error tolerance and other options
- Solving different kinds of problems
- Troubleshooting

General ODE Solver Questions Question Answer

How do the ODE solvers differ from quad or quadl? quad and quadl solve problems of the form $y' = f(t, y)$. The ODE solvers handle more general problems $y' = f(t, M(t) y)$, or problems that involve a mass matrix $M(t) y' = f(t, y)$.

Can I solve ODE systems in which there are more equations than unknowns, or vice versa?
No.

Problem Size, Memory Use, and Computation Speed Question Answer How large a problem can I solve with the ODE suite?

I'm solving a very large system, but only care about a couple of the components of y . Is there any

way to avoid storing all of the elements?

The primary constraints are memory and time. At each time step, the solvers for nonstiff problems allocate vectors of length n , where n is the number of equations in the system. The solvers for stiff problems allocate vectors of length n , but also an $n \times n$ Jacobian matrix. For these solvers it may be advantageous to use the sparse option.

If the problem is nonstiff, or if you are using the sparse option, it may be possible to solve a problem with thousands of unknowns. In this case, however, storage of the result can be problematic. Try asking the solver to evaluate the solution at specific points only, or call the solver with no output arguments and use an output function to monitor the solution.

Yes. The user-installable output function capability is designed specifically for this purpose. When you call the solver with no output arguments, the solver does not allocate storage to hold the entire solution history. Instead, the solver calls `OutputFcn(t,y)` at each time step. To keep the history of specific elements, write an output function that stores or plots only the elements you care about.

Problem Size, Memory Use, and Computation Speed (Continued) Question Answer

What is the startup cost of the integration and how can I reduce it?

The biggest startup cost occurs as the solver attempts to find a step size appropriate to the scale of the problem. If you happen to know an appropriate step size, use the `InitialStep` property. For example, if you repeatedly call the integrator in an event location loop, the last step that was taken before the event is probably on scale for the next integration. See `ballode` for an example.

Time Steps for Integration Question Answer

The first step size that the integrator takes is too large, and it misses important behavior.

Can I integrate with fixed step sizes?

You can specify the first step size with the `InitialStep` property. The integrator tries this value, then reduces it if necessary.

No.

Error Tolerance and Other Options Question Answer How do I choose `RelTol` and `AbsTol`?

I want answers that are correct to the precision of the computer. Why can't I simply set `RelTol` to eps ?

How do I tell the solver that I don't care about getting an accurate answer for one of the solution components? `RelTol`, the relative accuracy tolerance, controls the number of correct digits in the answer. `AbsTol`, the absolute error tolerance, controls the difference between the answer and the solution. At each step, the error in component i of the solution satisfies

$$|e(i)| \leq \max(\text{RelTol} * \text{abs}(y(i)), \text{AbsTol}(i))$$

Roughly speaking, this means that you want `RelTol` correct digits in all solution components

except those smaller than thresholdsAbsTol(i). Even if you are not interested in a componenty(i) when it is small, you may have to specify AbsTol(i) small enough to get some correct digits iny(i) so that you can accurately compute more interesting components.

You can get close to machine precision, but not that close. The solvers do not allowRelTol neareps because they try to approximate a continuous function. At tolerances comparable to eps, the machine arithmetic causes all functions to look discontinuous.

You can increase the absolute error tolerance corresponding to this solution component. If the tolerance is bigger than the component, this specifies no correct digits for the component. The solver may have to get some correct digits in this component to compute other components accurately, but it generally handles this automatically.

Solving Different Kinds of Problems Question Answer

Can the solvers handle partial differential equations (PDEs) that have been discretized by the method of lines?

Can I solve differential-algebraic equation (DAE) systems?

Can I integrate a set of sampled data?

Yes, because the discretization produces a system of ODEs. Depending on the discretization, you might have a form involving mass matrices – MATLAB ODE solvers provide for this. Often the system is stiff. This is to be expected when the PDE is parabolic and when there are phenomena that happen on very different time scales such as a chemical reaction in a fluid flow. In such cases, use one of the four solvers:ode15s, ode23s, ode23t, ode23tb. If there are many equations, set theJPattern property. This might make the difference between success and failure due to the computation being too expensive. When the system is not stiff, or not very stiff,ode23 orode45 is more efficient than ode15s, ode23s, ode23t, orode23tb. For an example that uses JPatter, see “Example: Large, Stiff Sparse Problem” on page 15-38.

Parabolic-elliptic partial differential equations in 1-D can be solved directly with the MATLAB PDE solver,pdepe. See “Partial Differential Equations” on page 15-82 for more information.

Yes. The solvers ode15s andode23t can solve some DAEs of the form $M\dot{f}(t) = f(t)$ where $M\dot{f}(t)$ is singular. The DAEs must be of index 1. For examples, see amp1dae orhb1dae.

Not directly. You have to represent the data as a function by interpolation or some other scheme for fitting data. The smoothness of this function is critical. A piecewise polynomial fit like a spline can look smooth to the eye, but rough to a solver; the solver takes small steps where the derivatives of the fit have jumps. Either use a smooth function to represent the data or use one of the lower order solvers (ode23, ode23s, ode23t, ode23tb) that is less sensitive to this.

Solving Different Kinds of Problems (Continued) Question Answer

Can I solve delay-differential equations?

What do I do when I have the final and not the initial value?

Not directly. In some cases it is possible to use the initial value problem solvers to solve delay-differential equations by breaking the simulation interval into smaller intervals the length of a single delay. For more information about this approach, see [6].

All the solvers of the MATLAB ODE suite allow you to solve backwards or forwards in time. The syntax for the solvers is $[t,y] = \text{ode45}(\text{odefun}, [t0 \text{ } tf], y0)$; and the syntax acceptst0 > tf.

Troubleshooting Question Answer The solution doesn't look like what I expected.

My plots aren't smooth enough.

If you're right about its appearance, you need to reduce the error tolerances from their default values. A smaller relative error tolerance is needed to compute accurately the solution of problems integrated over "long" intervals, as well as solutions of problems that are moderately unstable. You should check whether there are solution components that stay smaller than their absolute error tolerance for some time. If so, you are not asking for any correct digits in these components. This may be acceptable for these components, but failing to compute them accurately may degrade the accuracy of other components that depend on them.

Increase the value of Refine from its default of 4 inode45 and 1 in the other solvers. The bigger the value of Refine, the more output points. Execution speed is not affected much by the value of Refine.

Troubleshooting (Continued) Question Answer

I'm plotting the solution as it is computed and it looks fine, but the code gets stuck at some point. First verify that the ODE function is smooth near the point where the code gets stuck. If it isn't, the solver must take small steps to deal with this. It may help to break tspan into pieces on which the ODE function is smooth.

If the function is smooth and the code is taking extremely small steps, you are probably trying to solve a stiff problem with a solver not intended for this purpose. Switch toode15s, ode23s, ode23t, or ode23tb.

My integration proceeds very slowly, using too many time steps.

I know that the solution undergoes a radical change at time t where

$t_0 \mid t \mid t_f$

but the integrator steps past without "seeing" it. First, check that your tspan is not too long. Remember that the solver uses as many time points as necessary to produce a smooth solution. If the ODE function changes on a time scale that is very short compared to the tspan, the solver uses a lot of time steps. Long-time integration is a hard problem. Break tspan into smaller pieces.

If the ODE function does not change noticeably on the tspan interval, it could be that your problem is stiff. Try usingode15s, ode23s, ode23t, orode23tb.

Finally, make sure that the ODE function is written in an efficient way. The solvers evaluate the derivatives in the ODE function many times. The cost of numerical integration depends critically on the expense of evaluating the ODE function. Rather than recompute complicated constant parameters at each evaluation, store them in globals or calculate them once outside the function and pass them in as additional parameters.

If you know there is a sharp change at time t , it might help to break the tspan interval into two pieces, $[t_0 t]$ and $[t \text{tf}]$, and call the integrator twice.

If the differential equation has periodic coefficients or solution, you might restrict the maximum step size to the length of the period so the integrator won't step over periods.

Boundary Value Problems for ODEs

This section describes how to use MATLAB to solve boundary value problems (BVPs) of ordinary differential equations (ODEs). It provides:

- A summary of the MATLAB BVP functions and examples
- An introduction to BVPs
- A description of the BVP solver and its syntax
- General instructions for representing a BVP in MATLAB
- A discussion and examples about using continuation to solve a difficult problem
- A discussion about changing default integration properties to improve solver performance

BVP Function Summary ODE Boundary Value Problem Solver

Solver Description `bvp4c` Solve two-point boundary value problems for ordinary differential equations.

BVP Helper Functions

Function Description `bvpinit` Form the initial guess for `bvp4c`. `deval` Evaluate the numerical solution using the output of `bvp4c`.

BVP Option Handling

An options structure contains named properties whose values are passed to the solver, and which affect problem solution. Use these functions to create, alter, or access an options structure.

Function Description `bvpset` Create/alter the BVP options structure. `bvpget` Extract properties from options structure created with `bvpset`.

ODE Boundary Value Problem Examples

These examples illustrate the kind of problems you can solve using the MATLAB BVP solver. From the MATLAB Help browser, click the example name to see the code in an editor. Type the example name at the command line to run it.

Note The Differential Equations Examples browser enables you to view the code for the BVP examples, and also run them. Click on the link to invoke the browser, or type `odeexamples('bvp')` at the command line.

Example Description `fsbvp` Falkner-Skan BVP on an infinite interval `mat4bvp` Fourth

eigenfunction of Mathieu's equation shockbvp Solution with a shock layer near $x = 0$ twobvp
BVP with exactly two solutions

Additional examples are provided with the tutorial by Shampine, Reichelt, and Kierzenka, "Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c." The tutorial and the examples are available at [attp://ftp.mathworks.com/pub/doc/papers/bvp/](http://ftp.mathworks.com/pub/doc/papers/bvp/). This tutorial illustrates techniques for solving nontrivial real-life problems.

Introduction to Boundary Value ODE Problems

The BVP solver is designed to handle systems of ordinary differential equations

$$\dot{y} = f(x, y)$$

where x is the independent variable, y is the dependent variable, and \dot{y} represents $\frac{dy}{dx}$.

See "What Is an Ordinary Differential Equation" on page 15-6 for general information about ODEs.

Using Boundary Conditions to Specify the Solution of Interest In a *boundary value problem*, the solution of interest satisfies certain boundary conditions. These conditions specify a relationship between the values of the solution at more than one x . bvp4c is designed to solve two-point BVPs, i.e., problems where the solution sought on an interval $[a, b]$ must satisfy the boundary conditions

$$g(y(a), y(b)) = 0$$

Unlike initial value problems, a boundary value problem may not have a solution, may have a finite number of solutions, or may have infinitely many solutions. As an integral part of the process of solving a BVP, you need to provide a guess for the required solution. The quality of this guess can be critical for the solver performance and even for a successful computation.

There may be other difficulties when solving BVPs, such as problems imposed on infinite intervals or problems that involve singular coefficients. Often BVPs involve unknown parameters p that have to be determined as part of solving the problem

$$\dot{y} = f(x, y, p)$$

$$g(y(a), y(b), p) = 0$$

In this case, the boundary conditions must suffice to determine the value of p .

Boundary Value Problem Solver

This section describes:

- The BVP solver, bvp4c
- BVP solver basic syntax
- Additional BVP solver arguments

You can also use the MATLAB Help browser to get information about the syntax for MATLAB functions.

The BVP Solver

The function bvp4c solves two-point boundary value problems for ordinary differential equations

(ODEs). It integrates a system of first-order ordinary differential equations

$$y' = f(x, y)$$

on the interval $a \leq x \leq b$, subject to general two-point boundary conditions

$$bc_f(y_a) = 0$$

It can also accommodate unknown parameters for problems of the form

$$y' = f(x, y, p)$$

$$bc_f(y_a) = 0$$

In this case, the number of boundary conditions must be sufficient to determine the solution and the unknown parameters. For more information, see “[Finding Unknown Parameters](#)” on page 15-67.

`bvp4c` produces a solution that is continuous on $a \leq x \leq b$ and has a continuous first derivative there. You can use the function `deval` and the output of `bvp4c` to evaluate the solution at specific points on the interval of integration.

`bvp4c` is a finite difference code that implements the 3-stage Lobatto IIIa formula. This is a collocation formula and the collocation polynomial provides a C^1 -continuous solution that is fourth-order accurate uniformly in the interval of integration. Mesh selection and error control are based on the residual of the continuous solution.

The collocation technique uses a mesh of points to divide the interval of integration into subintervals. The solver determines a numerical solution by solving a global system of algebraic equations resulting from the boundary conditions, and the collocation conditions imposed on all the subintervals. The solver then estimates the error of the numerical solution on each subinterval. If the solution does not satisfy the tolerance criteria, the solver adapts the mesh and repeats the process. The user *must* provide the points of the initial mesh as well as an initial approximation of the solution at the mesh points.

BVP Solver Basic Syntax

The basic syntax of the BVP solver is

```
sol = bvp4c(odefun,bcfun,solinit)
```

The input arguments are:

`odefun` Function that evaluates the differential equations. It has the basic form
`dydx = odefun(x,y)`

where `x` is a scalar, and `dydx` and `y` are column vectors. `odefun` can also accept a vector of unknown parameters and a variable number of known parameters.

`bcfun` Function that evaluates the residual in the boundary conditions. It has the basic form
`res = bcfun(ya,yb)`

where `ya` and `yb` are column vectors representing `y(a)` and `y(b)`, and `res` is a column vector of the residual in satisfying the boundary conditions. `bcfun` can also accept a vector of unknown

parameters and a variable number of known parameters.

`solinit` Structure with `fieldsx` and `y`:

`x` Ordered nodes of the initial mesh. Boundary conditions are imposed at $a = \text{solinit.x}(1)$ and $b = \text{solinit.x}(\text{end})$.

`y` Initial guess for the solution with `solinit.y(:,i)` a guess for the solution at the node `solinit.x(i)`. The structure can have any name, but the fields must be named `x` and `y`. It can also contain a vector that provides an initial guess for unknown parameters. You can form `solinit` with the helper function `bvpinit`. See the `bvpinit` reference page for details.

The output argument `sol` is a structure created by the solver. In the basic case the structure has `fieldsx`, `y`, and `yp`.

`sol.x` Nodes of the mesh selected by `bvp4c`

`sol.y` Approximation to y at the mesh points of `sol.x`

`sol.parameters`

Approximation to \dot{y} at the mesh points of `sol.x` Value of unknown parameters, if present, found by the solver.

`sol.solver` 'bvp4c'

The function `deval` uses the output structure `sol` to evaluate the numerical solution at any point from $[a,b]$.

Additional BVP Solver Arguments

For more advanced applications, you can also specify as input arguments solver options and additional known parameters.

`options` Structure of optional parameters that change the default integration properties. This is the fourth input argument. `sol = bvp4c(odefun,bcfun,solinit,options)`

“Creating and Maintaining a BVP Options Structure” on page 15-76 tells you how to create the structure and describes the properties you can specify.

`p1,p2...` Known parameters that the solver passes to `odefun` and `bcfun`.

`sol = bvp4c(odefun,bcfun,solinit,options,p1,p2...)`

The solver passes any input parameters that follow the `options` argument to `odefun` and `bcfun` every time it calls them. Use `options = []` as a placeholder if you set no options. In the `odefun` argument list, known parameters follow `x`, `y`, and a vector of unknown parameters (`parameters`), if present.

`dydx = odefun(x,y,p1,p2,...)`

`dydx = odefun(x,y,parameters,p1,p2,...)`

In the `bcfun` argument list, known parameters follow `ya`, `yb`, and a vector of unknown parameters, if present.

`res = bcfun(ya,yb,p1,p2,...)`

`res = bcfun(ya,yb,parameters,p1,p2,...)`

See “Example: Using Continuation to Solve a Difficult BVP” on page 15-68 for an example.

Representing BVP Problems

This section describes:

- The process for solving boundary value problems (BVPs) using the MATLAB solver,bvp4c
- Finding unknown parameters
- Evaluating the solution at specific points

Example:Mathieu'sEquation

This example determines the fourth eigenvalue of Mathieu's Equation. It illustrates how to write second-order differential equations as a system of two first-order ODEs and how to usebvp4c to determine an unknown parameter λ .

The task is to compute the fourth ($q = 5$) eigenvalue λ of Mathieu's equation $y'' + \lambda - 2q \cos 2x y = 0$

Because the unknown parameter λ is present, this second-order differential equation is subject to three boundary conditions

$$y(0) = 1 \quad y'(0) = 0 \quad y'(\pi) = 0$$

Note The demomat4bvp contains the complete code for this example. The demo uses subfunctions to place all functions required bybvp4c in a single M-file. To run this example type mat4bvp at the command line. See “BVP Solver Basic Syntax” on page 15-61 for more information.

1 Rewrite the Problem as a First-Order System. To usebvp4c, you must rewrite the equations as an equivalent system of first-order differential equations. Using a substitution $y_1 = y$ and $y_2 = y'$, the differential equation is written as a system of two first-order equations

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -\lambda - 2q \cos 2x y_1 \end{aligned}$$

Note that the differential equations depend on the unknown parameter λ . The boundary conditions become

$$\begin{aligned} y_1(0) &= 1 \\ y_2(0) &= 0 \\ y_2(\pi) &= 0 \end{aligned}$$

2 Code the System of First-Order ODEs in MATLAB. Once you represent the equation as a first-order system, you can code it as a function thatbvp4c can use. Because there is an unknown parameter, the function must be of the form

`dydx = odefun(x,y,parameters)`

The code below represents the system in the MATLAB function,mat4ode.

```
function dydx = mat4ode(x,y,lambda)
q = 5;
dydx = [ y(2)
          -lambda - 2*q*cos(2*x)*y(1) ]
```

```
-(lambda - 2*q*cos(2*x))*y(1)];
```

See “Finding Unknown Parameters” on page 15-67 for more information about using unknown parameters withbvp4c.

3 Code the Boundary Conditions Function. You must also code the boundary conditions in a MATLAB function. Because there is an unknown parameter, the function must be of the form

```
res = bcfun(ya,yb,parameters)
```

The code below represents the boundary conditions in the MATLAB function, mat4bc.

```
function res = mat4bc(ya,yb,lambda)
res = [ ya(2)
yb(2)
ya(1)-1];
```

4 Create an Initial Guess. To form the guess structuresolinit withbvpinit, you need to provide initial guesses for both the solution and the unknown parameter.

The function mat4init provides an initial guess for the solution. mat4init uses $y = \cos 4x$ because this function satisfies the boundary conditions and has the correct qualitative behavior (the correct number of sign changes).

```
function yinit = mat4init(x)
yinit = [ cos(4*x)
-4*sin(4*x)];
```

In the call tobvpinit, the third argument, lambda, provides an initial guess for the unknown parameter λ .

```
lambda = 15;
```

solinit = bvpinit(linspace(0,pi,10),@mat4init,lambda); This example uses @ to pass mat4init as a function handle to tobvpinit.

Note See the `function_handle (@)`, `func2str`, and `str2func` reference pages, and the “Function Handles” chapter of “Programming and Data Types” in the MATLAB documentation for information about function handles. **5 Apply the BVP Solver.** The mat4bvp example calls `bvp4c` with the functions `mat4ode` and `mat4bc` and the structure `solinit` created with `bvpinit`.

```
sol = bvp4c(@mat4ode,@mat4bc,solinit);
```

6 View the Results. Complete the example by displaying the results:

- Print the value of the unknown parameter λ found by `bvp4c`. `fprintf('The fourth eigenvalue is approximately %7.3f.\n',... sol.parameters)`

- Use `deval` to evaluate the numerical solution at 100 equally spaced points in the interval $[0, \sqrt{8}]$, and plot its first component. This component approximates $y = \cos 4x$.

```
xint = linspace(0,pi);
Sxint = deval(sol,xint);
plot(xint,Sxint(1,:))
axis([0 pi -1 1.1])
```

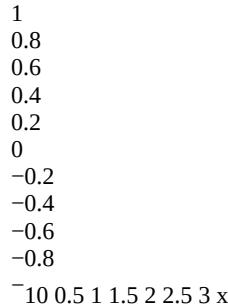
```

title('Eigenfunction of Mathieu''s equation.')
xlabel('x')
ylabel('solution y')

```

See “Evaluating the Solution at Specific Points” on page 15-68 for information about using `deval`. The following plot shows the eigenfunction associated with the final eigenvalue $\lambda = 17.097$.

Eigenfunction of Mathieu’s equation.



Finding Unknown Parameters

The `bvp4c` solver can find unknown parameters p for problems of the form $y' = f(x, y, p)$
 $bc(y(a), y(b)) = 0$

You must provide `bvp4c` an initial guess for any unknown parameters in the `vectorsolinit.parameters`. When you call `bvpinit` to create the structure `solinit`, specify the initial guess as a vector in the additional argument `parameters`.

```
solinit = bvpinit(x,v,parameters)
```

The `bvp4c` function arguments `odefun` and `bcfun` must each have a third argument.

```
dydx = odefun(x,y,parameters)
```

```
res = bcfun(ya,yb,parameters)
```

The `bvp4c` solver calculates intermediate values of unknown parameters at each iteration, and passes the latest values to `odefun` and `bcfun` in the `parameters` arguments. The solver returns the final values of these unknown parameters in `sol.parameters`. See “Example: Mathieu’s Equation” on page 15-63.

Evaluating the Solution at Specific Points

The collocation method implemented in `bvp4c` produces a C^1 -continuous solution over the whole interval of integration $a \leq x \leq b$. You can evaluate the approximate solution, $S(x)$, at any point in $a \leq x \leq b$ using the helper function `deval` and the structure `sol` returned by `bvp4c`.

```
Sxint = deval(sol,xint)
```

The `deval` function is vectorized. For a vector `xint`, the i th column of `Sxint` approximates the solution $y(xint(i))$.

Using Continuation to Make a Good Initial Guess

To solve a boundary value problem, you need to provide an initial guess for the solution. The quality of your initial guess can be critical to the solver performance, and to being able to solve the problem at all. However, coming up with a sufficiently good guess can be the most challenging part of solving a boundary value problem. Certainly, you should apply the

knowledge of the problem's physical origin. Often a problem can be solved as a sequence of relatively simpler problems, i.e., a continuation. This section provides examples that illustrate how to use continuation to:

- Solve a difficult BVP.
- Verify a solution's consistent behavior.

Example: Using Continuation to Solve a Difficult BVP This example solves the differential equation

$$\frac{dy}{dx} + xy^2 = \sqrt{e^x - x \sin(\sqrt{e^x})}$$

for $e = 10^{-4}$, on the interval $[-1, 1]$, with boundary conditions $y(-1) = -2$ and $y(1) = 0$. For $e < 1$, the solution has a transition layer at $x = 0$. Because of this rapid change in the solution for small values of e , the problem becomes difficult to solve numerically.

The example solves the problem as a sequence of relatively simpler problems, i.e., a continuation. The solution of one problem is used as the initial guess for solving the next problem.

Note The demoshockbvp contains the complete code for this example. The demo uses subfunctions to place all required functions in a single M-file. To run this example typeshockbvp at the command line. See “BVP Solver Basic Syntax” on page 15-61 and “Representing BVP Problems” on page 15-63 for more information.

Note This problem appears in [1] to illustrate the mesh selection capability of a well established BVP code COLSYS.

1 Code the ODE and Boundary Condition Functions. Code the differential equation and the boundary conditions as functions that bvp4c can use. Because there is an additional known parameter e , the functions must be of the form

```
dydx = odefun(x,y,p1)
res = bcfun(ya,yb,p1)
```

The code below represents the differential equation and the boundary conditions in the MATLAB function shockODE and shockBC. Note that shockODE is vectorized to improve solver performance. The additional parameter e is represented by π .

```
function dydx = shockODE(x,y,e)
pix = pi*x;
dydx = [ y(2,:)
-x./e.*y(2,:)-pi^2*cos(pix)-pix./e.*sin(pix) ];
function res = shockBC(ya,yb,e)
res = [ ya(1)+2
yb(1) ];
```

The example passes e as an additional input argument to bvp4c. $sol = bvp4c(@shockODE,@shockBC,sol,options,e);$ bvp4c then passes this argument to the functions shockODE and shockBC when it evaluates them. See “Additional BVP Solver

Arguments” on page 15-62 for more information.

2 Provide Analytical Partial Derivatives. For this problem, the solver benefits from using analytical partial derivatives. The code below represents the derivatives in functions `shockJac` and `shockBCJac`.

```
function jac = shockJac(x,y,e)
jac = [ 0 1
0 -x/e ];
function [dBCdyA,dBCdyB] = shockBCJac(yA,yB,e) dBCdyA = [ 1 0
0 0 ];
dBCdyB = [ 0 0
1 0 ];
```

`shockJac`
`and``shockBCJac` must accept the additional argument `e`, because `bvp4c` passes the additional argument to all the functions the user supplies.

Tell `bvp4c` to use these functions to evaluate the partial derivatives by setting the options `FJacobian` and `BCJacobian`. Also set 'Vectorized' to 'on' to indicate that the differential equation function `shockODE` is vectorized.

```
options = bvpset('FJacobian',@shockJac,...
'BCJacobian',@shockBCJac,...
'Vectorized','on');
```

3 Create an Initial Guess. You must provide `bvp4c` with a guess structure that contains an initial mesh and a guess for values of the solution at the mesh points. A constant guess of $y = 1$ and $\dot{y} = 0$, and a mesh of five equally spaced points on $[-1, 1]$ suffice to solve the problem for $\epsilon = 10^{-2}$. Use `bvpinit` to form the guess structure.

```
sol = bvpinit([-1 -0.5 0 0.5 1],[1 0]);
```

4 Use Continuation to Solve the Problem. To obtain the solution for the parameter $\epsilon = 10^{-4}$, the example uses continuation by solving a sequence of problems for $\epsilon = 10^{-2}, 10^{-3}, 10^{-4}$. The solver `bvp4c` does not perform continuation automatically, but the code's user interface has been designed to make continuation easy. The code uses the output `sol` that `bvp4c` produces for one value of `e` as the guess in the next iteration.

```
e = 0.1;
for i=2:4
e = e/10;
sol = bvp4c(@shockODE,@shockBC,sol,options,e);
end
```

5 View the Results. Complete the example by displaying the final solution

```

plot(sol.x,sol.y(1,:))
axis([-1 1 -2.2 2.2])
title(['There is a shock at x = 0 when \epsilon = '...
sprintf('%e',epsilon)])
xlabel('x')
ylabel('solution y')

There is a shock at x = 0 when \epsilon = 1e-004.
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-1 -0.8 -0.6 -0.4 -0.2 0 0.2 0.4 0.6 0.8 1 x

```

Example: Using Continuation to Verify a Solution's Consistent Behavior Falkner-Skan BVPs arise from similarity solutions of viscous, incompressible, laminar flow over a flat plate. An example is

$$\frac{d^2f}{dx^2} + f \frac{df}{dx} + \frac{1}{2} f^2 = 0$$

for $\epsilon = 0.5$ on the interval $[0, \frac{n}{k}]$ with boundary conditions $f(0) = 0$, $f'(0) = 0$, and $f'(\frac{n}{k}) = 1$.

The BVP cannot be solved on an infinite interval, and it would be impractical to solve it for even a very large finite interval. So, the example tries to solve a sequence of problems posed on increasingly larger intervals to verify the solution's consistent behavior as the boundary approaches $\frac{n}{k}$.

The example imposes the infinite boundary condition at a finite point called infinity. The example then uses continuation in this end point to get convergence for increasingly larger values of infinity. It uses `bvpinit` to extrapolate the solution `sol` for one value of infinity as an initial guess for the new value of infinity. The plot of each successive solution is superimposed over those of previous solutions so they can easily be compared for consistency.

Note The demo `fsbvp` contains the complete code for this example. The demo uses subfunctions to place all required functions in a single M-file. To run this example type `fsbvp` at the command line. See “BVP Solver Basic Syntax” on page 15-61 and “Representing BVP Problems” on page 15-63 for more information.

1 Code the ODE and Boundary Condition Functions. Code the differential equation and the boundary conditions as functions that `bvp4c` can use.

```

function dfdeta = fsode(eta,f)
beta = 0.5;
dfdeta = [ f(2)

```

```

f(3)
```

```

-f(1)*f(3) - beta*(1 - f(2)^2) ];
function res = fsbc(f0,finf) res = [f0(1)
f0(2)
finf(2) - 1];

```

2 Create an Initial Guess. You must provide bvp4c with a guess structure that contains an initial mesh and a guess for values of the solution at the mesh points. A crude mesh of five points and a constant guess that satisfies the boundary conditions are good enough to get convergence wheninfinity = 3.

```

infinity = 3;
maxinfinity = 6;
solinit = bvpinit(linspace(0,infinity,5),[0 0 1]);

```

4 Solve on the Initial Interval. The example obtains the solution forinfinity = 3. It then prints the computed value of $f'''(0)$ for comparison with the value reported by Cebeci and Keller [2].

```

sol = bvp4c(@fsode,@fsbc,solinit);
eta = sol.x;
f = sol.y;

fprintf('\n');
fprintf('Cebeci & Keller report that f'''(0) = 0.92768.\n') fprintf('Value computed using infinity =
%g is '...
">%7.5f.\n',Bnew,f(3,1))

```

The example prints

Cebeci & Keller report that $f'''(0) = 0.92768.$

Value computed using infinity = 3 is 0.92915.

5 Setup the Figure and Plot the Initial Solution.

```

figure
plot(eta,f(2,:),eta(end),f(2,end),'o');
axis([0 maxinfinity 0 1.4]);
title('Falkner-Skan equation, positive wall shear, \beta = 0.5.') xlabel('\eta')
ylabel('df/d\eta')
hold on
drawnow
shg

```

Falkner-Skan equation, positive wall shear, $\beta = 0.5.$

1.2
1
0.8
0.6
0.4
0.2
0 1 2 3 4 5 6

6 Use Continuation to Solve the Problem and Plot Subsequent Solutions. The example then solves the problem for $\infty = 4, 5, 6$. It uses `bvpinit` to extrapolate the solution `sol` for one value of ∞ as an initial guess for the next value of ∞ . For each iteration, the example prints the computed value of $f'(0)$ and superimposes a plot of the solution in the existing figure.

```
for Bnew = infinity+1:maxinfinity
```

```
    solinit = bvpinit(sol,[0 Bnew]); % Extend solution to Bnew. sol = bvp4c(@fsode,@fsbc,solinit);
    eta = sol.x;
    f = sol.y;
```

```
    fprintf('Value computed using infinity = %g is '... '%7.5f.\n',Bnew,f(3,1))
    plot(eta,f(2,:),eta(end),f(2,end),'o');
    drawnow
```

```
end
hold off
```

The example prints

```
Value computed using infinity = 4 is 0.92774.
Value computed using infinity = 5 is 0.92770.
Value computed using infinity = 6 is 0.92770.
```

Note that the values approach 0.92768 as reported by Cebeci and Keller. The superimposed plots confirm the consistency of the solution's behavior.

Falkner–Skan equation, positive wall shear, $\epsilon = 0.5$.

```
1.2
1
0.8
0.6
0.4
0.2
00 1 2 3 4 5 6 ↘
```

Improving BVP Solver Performance

The default integration properties in the BVP solver `bvp4c` are selected to handle common problems. In some cases, you can improve solver performance by changing these defaults. To do this, supply `bvp4c` with one or more property values in an options structure.

```
sol = bvp4c(odefun,bcfun,solinit,options)
```

This section:

- Explains how to create, modify, and query an options structure
- Describes the properties that you can use in an options structure

In this and subsequent property tables, the most commonly used property categories are listed first, followed by more advanced categories.

BVP Property Categories	Properties	Category	Property Names	Error control	RelTol,AbsTol
Vectorization	Vectorized	Analytical	partial derivatives	FJacobian,BCJacobian	Mesh size
Output	displayed	Stats		NMax	

Note For other ways to improve solver efficiency, check “Using Continuation to Make a Good Initial Guess” on page 8-68 and the tutorial, “Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c,” available at [ftp://ftp.mathworks.com/pub/doc/papers/bvp/](http://ftp.mathworks.com/pub/doc/papers/bvp/).

Creating and Maintaining a BVP Options Structure

The `bvpset` function creates an options structure that you can supply to `bvp4c`. You can use `bvpget` to query the options structure for the value of a specific property.

Creating an Options Structure. The `bvpset` function accepts property name/property value pairs using the syntax
`options = bvpset('name1',value1,'name2',value2,...)`

This creates a structure `options` in which the named properties have the specified values. Unspecified properties retain their default values. For all properties, it is sufficient to type only the leading characters that uniquely identify the property name. `bvpset` ignores case for property names.

With no arguments, `bvpset` displays all property names and their possible values, indicating defaults with braces {}.

Modifying an Existing Options Structure. To modify an existing `options` argument, use
`options = bvpset(olddopts,'name1',value1,...)`

This overwrites any values in `olddopts` that are specified using name/value pairs. The modified structure is returned as the output argument. In the same way, the command

`options = bvpset(olddopts,newopts)`
combines the structures `olddopts` and `newopts`. In `options`, any values set in `newopts` overwrite those in `olddopts`.

Querying an Options Structure. The `bvpget` function extracts a property value from an options structure created with `bvpset`.

`o = bvpget(options,'name')`

This returns the value of the specified property, or an empty matrix [] if the property value is unspecified in the options structure.

As with `bvpset`, it is sufficient to type only the leading characters that uniquely identify the property name; case is ignored for property names.

Error Tolerance Properties

Because `bvp4c` uses a collocation formula, the numerical solution is based on a mesh of points at which the collocation equations are satisfied. Mesh selection and error control are based on the residual of this solution, such that the computed solution $S\tilde{F}x\tilde{f}$ is the exact solution of a perturbed

problem $S\|f(x)\| = \|f(x)\|_2 \leq \text{AbsTol} + \text{RelTol} \cdot \text{max}(\|f(x)\|, \text{AbsTol})$. On each subinterval of the mesh, a norm of the residual in the i th component of the solution, $\text{res}(i)$, is estimated and is required to be less than or equal to a tolerance. This tolerance is a function of the relative and absolute tolerances, RelTol and AbsTol , defined by the user.

$$(\text{res}(i)/\max(\|f(x)\|, \text{AbsTol})) \leq \text{RelTol}$$

The following table describes the error tolerance properties. Use `bvpset` to set these properties.

BVP Error Tolerance Properties	Property	Value	Description
<code>RelTol</code>	Positive scalar	{1e-3}	
<code>AbsTol</code>	Positive scalar or vector	{1e-6}	A relative error tolerance that applies to all components of the residual vector. It is a measure of the residual relative to the size of $\ f(x)\ _2$. The default, 1e-3, corresponds to 0.1% accuracy.

Absolute error tolerances that apply to the corresponding components of the residual vector. `AbsTol(i)` is a threshold below which the values of the corresponding components are unimportant. If a scalar value is specified, it applies to all components.

Vectorization

The following table describes the BVP vectorization property. Vectorization of the ODE function used by `bvp4c` differs from the vectorization used by the ODE solvers:

- For `bvp4c`, the ODE function must be vectorized with respect to the first argument as well as the second one, so that $F([x_1 \ x_2 \ ...], [y_1 \ y_2 \ ...])$ returns $[F(x_1, y_1) \ F(x_2, y_2) \ ...]$.
- `bvp4c` benefits from vectorization even when analytical Jacobians are provided. For stiff ODE solvers, vectorization is ignored when analytical Jacobians are used.

Use `bvpset` to set this property.

Property	Value	Description
<code>Vectorized</code>	on {off}	Set on to inform <code>bvp4c</code> that you have coded the ODE function F so that $F([x_1 \ x_2 \ ...], [y_1 \ y_2 \ ...])$ returns $[F(x_1, y_1) \ F(x_2, y_2) \ ...]$. This allows the solver to reduce the number of function evaluations, and may significantly reduce solution time.

With MATLAB's array notation, it is typically an easy matter to vectorize an ODE function. In the `shockbvp` example shown previously, the `shockODE` function has been vectorized using colon notation into the subscripts and by using the array multiplication (`*`) operator.

```
function dydx = shockODE(x,y,e)
pix = pi*x;
dydx = [ y(2,:)
          -x/e.*y(2,:)-pi^2*cos(pix)-pix/e.*sin(pix) ];
```

Analytical Partial Derivatives

By default, the `bvp4c` solver approximates all partial derivatives with finite differences. `bvp4c` can be more efficient if you provide analytical partial derivatives $\frac{\partial f}{\partial c} \frac{\partial f}{\partial y}$ of the differential equations, and analytical partial derivatives, $\frac{\partial b}{\partial c} \frac{\partial a}{\partial y}$ and $\frac{\partial b}{\partial c} \frac{\partial b}{\partial y}$, of the boundary conditions. If

the problem involves unknown parameters, you must also provide partial derivatives, $\frac{\partial f}{\partial c} \frac{\partial f}{\partial p}$ and $\frac{\partial^2 f}{\partial c^2} \frac{\partial^2 f}{\partial p^2}$, with respect to the parameters.

The following table describes the analytical partial derivatives properties. Use bvpset to set these properties.

BVP Analytical Partial Derivative Properties Property Value Description FJacobian

Function The function computes the analytical partial derivatives of

$f'_{x,y}$. When solving $y' = f'_{x,y}$, set this property to @fjac if $dfdy = fjac(x,y)$ evaluates the Jacobian $\frac{\partial f}{\partial c} \frac{\partial f}{\partial y}$. If the problem involves unknown parameters p , $[dfdy, dfdp] = fjac(x,y,p)$ must also return the partial derivative $\frac{\partial^2 f}{\partial c^2} \frac{\partial^2 f}{\partial p^2}$.

BCJacobian Function The function computes the analytical partial derivatives of

$bc'_{ya,yb}$. For boundary conditions $bc'_{ya,yb}$, set this property to @bcjac if $[dbcdfa, ddbcdb] = bcjac(ya,yb)$ evaluates the partial derivatives $\frac{\partial^2 bc}{\partial c^2} \frac{\partial^2 ya}{\partial a^2}$, and $\frac{\partial^2 bc}{\partial c^2} \frac{\partial^2 yb}{\partial b^2}$. If the problem involves unknown parameters p , $[dbcdfa, ddbcdb, dbcdp] = bcjac(ya,yb,p)$ must also return the partial derivative $\frac{\partial^2 bc}{\partial c^2} \frac{\partial^2 p}{\partial p^2}$.

Mesh Size Property

bvp4c solves a system of algebraic equations to determine the numerical solution to a BVP at each of the mesh points. The size of the algebraic system depends on the number of differential equations (n) and the number of mesh points in the current mesh (N). When the allowed number of mesh points is exhausted, the computation stops, bvp4c displays a warning message and returns the solution it found so far. This solution does not satisfy the error

tolerance, but it may provide an excellent initial guess for computations restarted with relaxed error tolerances or an increased value of NMax. The following table describes the mesh size property. Use bvpset to set this property.

BVP Mesh Size Property Property Value Description NMax positive integer {floor(1000/n)}

Maximum number of mesh points allowed when solving the BVP, where n is the number of differential equations in the problem. The default value of NMax limits the size of the algebraic system to about 1000 equations. For systems of a few differential equations, the default value of NMax should be sufficient to obtain an accurate solution.

Solution Statistic Property

The Stats property lets you view solution statistics.

The following table describes the solution statistics property. Use bvpset to set this property.

BVP Solution Statistic Property Property Value Description

Stats on | {off} Specifies whether statistics about the computations are displayed. If the stats property is on, after solving the problem, bvp4c displays:

- The number of points in the mesh
- The maximum residual of the solution

- The number of times it called the differential equation functionodefun to evaluate $f(x)$
- The number of times it called the boundary condition functionbcfun to evaluate $bc(y)$

Partial Differential Equations

This section describes how to use MATLAB to solve initial-boundary value problems for partial differential equations (PDEs). It provides:

- A summary of the MATLAB PDE functions and examples
- An introduction to PDEs
- A description of the PDE solver and its syntax
- General instructions for representing a PDE in MATLAB, including an example
- A discussion about changing default integration properties to improve solver performance
- An example of solving a real-life problem

PDE Function Summary MATLABPDE Solver

This is the MATLAB PDE solver.

PDE Initial-Boundary Value Problem Solver pdepe Solve initial-boundary value problems for systems of parabolic and elliptic PDEs in one space variable and time.

PDE Helper Function

PDE Helper Function pdeval Evaluate the numerical solution of a PDE using the output of pdepe.

PDE Examples

These examples illustrate some problems you can solve using the MATLAB PDE solver. From the MATLAB Help browser, click the example name to see the code in an editor. Type the example name at the command line to run it. **Note** The Differential Equations Examples browser enables you to view the code for the PDE examples, and also run them. Click on the link to invoke the browser, or typeodeexamples('pde')at the command line.

Example Description pdex1 Simple PDE that illustrates the straightforward formulation, computation, and plotting of the solution

pdex2 Problem that involves discontinuities

pdex3 Problem that requires computing values of the partial derivative

pdex4 System of two PDEs whose solution has boundary layers at both ends of the interval and changes rapidly for small t pdex5 System of PDEs with step functions as initial conditions

Introduction to PDE Problems

pdepe solves systems of PDEs in one spatial variable x and time t , of the form

$\frac{\partial u}{\partial t}$

$- \frac{\partial^2 u}{\partial x^2} + s$

$$\frac{\partial^m u}{\partial x^m} - m \frac{\partial^{m-1} u}{\partial x^{m-1}} \frac{\partial}{\partial x} \left(\frac{\partial u}{\partial x} \right) - \frac{\partial^2 u}{\partial x^2} = x f(x) \frac{\partial u}{\partial x}$$

The PDEs hold for $t_0 \leq t \leq t_f$ and $a \leq x \leq b$. The interval $[a, b]$ must be finite. m can be 0, 1, or 2, corresponding to slab, cylindrical, or spherical symmetry, respectively. If $m=0$, then $a=0$ must also hold.

In Equation 15-3, $\int f(x,t) u \frac{\partial}{\partial x} u dx$ is a flux term and $\int s(x,t) u \frac{\partial}{\partial x} u dx$ is a source term. The coupling of the partial derivatives with respect to time is restricted to multiplication by a diagonal matrix $c(x,t) u \frac{\partial}{\partial x} u$. The diagonal elements of this matrix are either identically zero or positive. An element that is identically zero corresponds to an elliptic equation and otherwise to a parabolic equation. There must be at least one parabolic equation. An element of c that corresponds to a parabolic equation can vanish at isolated values of x if they are mesh points. Discontinuities in c and/or s due to material interfaces are permitted provided that a mesh point is placed at each interface.

At the initial time $t = t_0$, for all x the solution components satisfy initial conditions of the form

$$u(x,t_0) = u_0(x) \quad (15 - 4)$$

At the boundary $x = a$ or $x = b$, for all t the solution components satisfy a boundary condition of the form

$$\star \frac{\partial u}{\partial x}$$

$$p(x,t) u + q(x,t) \frac{\partial u}{\partial x} = 0 \quad (15 - 5)$$

$q(x,t)$ is a diagonal matrix with elements that are either identically zero or never zero. Note that the boundary conditions are expressed in terms of the flux f rather than $\frac{\partial u}{\partial x}$. Also, of the two coefficients, only p can depend on u .

MATLAB Partial Differential Equation Solver

This section describes:

- The PDE solver, `pdepe`
- PDE solver basic syntax
- Additional PDE solver arguments

The PDE Solver

The MATLAB PDE solver, `pdepe`, solves initial-boundary value problems for systems of parabolic and elliptic PDEs in the one space variable x and time t . There must be at least one parabolic equation in the system.

The `pdepe` solver converts the PDEs to ODEs using a second-order accurate spatial discretization based on a set of nodes specified by the user. The discretization method is described in [8]. The time integration is done with `ode15s`. The `pdepe` solver exploits the capabilities of `ode15s` for solving the differential-algebraic equations that arise when Equation 15-3 contains elliptic equations, and for handling Jacobians with a specified sparsity pattern. `ode15s` changes both the time step and the formula dynamically.

After discretization, elliptic equations give rise to algebraic equations. If the elements of the initial conditions vector that correspond to elliptic equations are not “consistent” with the discretization, `pdepe` tries to adjust them before beginning the time integration. For this reason, the solution returned for the initial time may have a discretization error comparable to that at any other time. If the mesh is sufficiently fine, `pdepe` can find consistent initial conditions close to the given ones. If `pdepe` displays a message that it has difficulty finding consistent initial conditions, try refining the mesh. No adjustment is necessary for elements of the initial conditions vector

that correspond to parabolic equations.

PDE Solver Basic Syntax

The basic syntax of the solver is

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)
```

Note Correspondences given are to terms used in “Introduction to PDE Problems” on page 15-83.

The input arguments are: m Specifies the symmetry of the problem.m can be0 = slab, 1 = cylindrical, or2 = spherical. It corresponds to m in Equation 15-3. pdefun Function that defines the components of the PDE. It computes the terms c , f , and s in Equation 15-3, and has the form $[c,f,s] = \text{pdefun}(x,t,u,\text{dudx})$

where x and t are scalars, and u and dudx are vectors that approximate the solution u and its partial derivative with respect to x . c , f , and s are column vectors. c stores the diagonal elements of the matrix c .

icfun Function that evaluates the initial conditions. It has the form $u = \text{icfun}(x)$

When called with an argument x , icfun evaluates and returns the initial values of the solution components at x in the column vector u . bcfun Function that evaluates the terms p and q of the boundary

conditions. It has the form

```
[pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
```

where ul is the approximate solution at the left boundary $x_l = a$ and ur is the approximate solution at the right boundary $x_r = b$. pl and ql are column vectors corresponding to p and the diagonal of q evaluated at x_l . Similarly, pr and qr correspond to x_r . When $m \neq 0$ and $a = 0$, boundedness of the solution near $x = 0$ requires that the flux f vanish at $a = 0$. pdepe imposes this boundary condition automatically and it ignores values returned in pl and ql .

xmesh Vector [x_0, x_1, \dots, x_n] specifying the points at which a numerical solution is requested for every value in tspan . x_0 and x_n correspond to a and b , respectively.

Second-order approximation to the solution is made on the mesh specified in $xmesh$. Generally, it is best to use closely spaced mesh points where the solution changes rapidly. pdepe does not select the mesh in x automatically. You must provide an appropriate fixed mesh in $xmesh$. The cost depends strongly on the length of $xmesh$. When $m \neq 0$, it is not necessary to use a fine mesh near $x = 0$ to account for the coordinate singularity.

The elements of $xmesh$ must satisfy $x_0 < x_1 < \dots < x_n$. The length of $xmesh$ must be $12^{\frac{24}{2}}$.

tspan Vector [t_0, t_1, \dots, t_f] specifying the points at which a solution is requested for every value in $xmesh$. t_0 and t_f correspond to t_0 and t_f , respectively.

pdepe performs the time integration with an ODE solver that selects both the time step and formula dynamically. The solutions at the points specified in $tspan$ are obtained using the natural continuous extension of the integration formulas. The elements of $tspan$ merely specify where you want answers and the cost depends weakly on the length of $tspan$.

The elements of tspan must satisfy $t_0 < t_1 < \dots < t_f$. The length of tspan must be 24.

The output argument sol is a three-dimensional array, such that:

- $\text{sol}(:,:,k)$ approximates component k of the solution u .
- $\text{sol}(i,:,:k)$ approximates component k of the solution at time tspan(i) and mesh points xmesh(:).
- $\text{sol}(i,j,k)$ approximates component k of the solution at time tspan(i) and the mesh point xmesh(j).

Additional PDE Solver Arguments

For more advanced applications, you can also specify as input arguments solver options and additional parameters that are passed to the PDE functions. options Structure of optional parameters that change the default integration properties. This is the seventh input argument. sol = pdepe(m,pdefun,icfun,bcfun,...,xmesh,tspan,options)

See “Improving PDE Solver Performance” on page 15-93 for more information.

p1,p2... Parameters that the solver passes topdefun,icfun, and bcfun. sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,... options,p1,p2...)

The solver passes any input parameters that follow the options argument topdefun,icfun, and bcfun every time it calls them. Use options = [] as a placeholder if you set no options. In the pdefun argument list, parameters follow x, t, u, and dudx.

f = pdefun(x,t,u,dudx,p1,p2,...)

In the icfun argument list, parameters follow x. res = icfun(x,p1,p2,...)

In the bcfun argument list, parameters follow xl, ul, xr, ur, and t.

res = bcfun(xl,ul,xr,ur,t,p1,p2,...) See the pdex3 demo for an example.

Representing PDE Problems

This section describes:

- The process for solving PDE problems using the MATLAB solver, pdepe
- Evaluating the solution at specific points

Example: A Single PDE

This example illustrates the straightforward formulation, solution, and plotting of the solution of a single PDE

$$\frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} = \frac{\partial^3 u}{\partial t^3} + \frac{\partial^3 u}{\partial x^3}$$

This equation holds on an interval $0 \leq x \leq 1$ for times $0 \leq t \leq 1$. At $t = 0$, the solution satisfies the initial condition

$$u(x,0) = \sin(\sqrt{x})$$

At $x = 0$ and $x = 1$, the solution satisfies the boundary conditions $u(0,t) = 0$

$$u(1,t) = 0$$

x

Note The demopdex1 contains the complete code for this example. The demo uses subfunctions to place all functions it requires in a single M-file. To run the demo type pdex1 at the command line. See “PDE Solver Basic Syntax” on page 15-85 for more information.

1 Rewrite the PDE. Write the PDE in the form

$$\frac{\partial^3 u}{\partial t^3} = x \frac{\partial^3 u}{\partial x^3} - \frac{\partial^3 u}{\partial x^2} + s \frac{\partial^3 u}{\partial x \partial t^2}$$

This is the form shown in Equation 15-3 and expected by pdepe. See “Introduction to PDE Problems” on page 15-83 for more information. For this example, the resulting equation is

$$\begin{aligned} & \frac{\partial^3 u}{\partial t^3} = x^0 \frac{\partial^3 u}{\partial x^3} + \frac{\partial^3 u}{\partial x^2} + s \frac{\partial^3 u}{\partial x \partial t^2} \text{ with parameter } m = 0 \\ & c \star x \frac{\partial^3 u}{\partial x^3} + f \star x \frac{\partial^3 u}{\partial x^2} + s \star x \frac{\partial^3 u}{\partial x \partial t^2} = 0 \end{aligned}$$

2 Code the PDE in MATLAB. Once you rewrite the PDE in the form shown above (Equation 15-3) and identify the terms, you can code the PDE in a function that pdepe can use. The function must be of the form

[c,f,s] = pdefun(x,t,u,dudx)

where c, f, and s correspond to the c , f , and s terms. The code below computes c, f, and s for the example problem.

```
function [c,f,s] = pdex1pde(x,t,u,DuDx)
c = pi^2;
f = DuDx;
s = 0;
```

3 Code the Initial Conditions Function. You must code the initial conditions in a MATLAB function of the form

$u = \text{icfun}(x)$

The code below represents the initial conditions in the MATLAB function pdex1ic.

```
function u0 = pdex1ic(x)
```

$u0 = \sin(\pi * x);$

4 Code the Boundary Conditions Function. You must also code the boundary conditions in a MATLAB function of the form

```
[pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
```

The boundary conditions, written in the same form as Equation 15-5, are $u(0) = 0$ and $\frac{\partial^3 u}{\partial x^3}(0) = 0$.

$\frac{\partial^3 u}{\partial x^3}(0) = 0$

and

$$\sqrt{e^{-t} + 1} \frac{\partial^3 u}{\partial x^3}(t) = 0 \text{ at } x = 1 \frac{3}{8}x$$

The code below evaluates the components $p(x,t)$ and $q(x,t)$ of the boundary conditions in the MATLAB function `pdex1bc`.

```
function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
pr = pi * exp(-t);
qr = 1;
```

In the function `pdex1bc`, `pl` and `ql` correspond to the left boundary conditions ($x = 0$), and `pr` and `qr` correspond to the right boundary condition ($x = 1$).

5 Select Mesh Points for the Solution. Before you use the MATLAB PDE solver, you need to specify the mesh points t at which you want `pdepe` to evaluate the solution. Specify the points as vectors `t` and `x`.

The vectors `t` and `x` play different roles in the solver (see “MATLAB Partial Differential Equation Solver” on page 15-84). In particular, the cost and the accuracy of the solution depend strongly on the length of the vector `x`. However, the computation is much less sensitive to the values in the vector `t`.

This example requests the solution on the mesh produced by 20 equally spaced points from the spatial interval $[0,1]$ and five values of `t` from the time interval $[0,2]$.

```
x = linspace(0,1,20);
t = linspace(0,2,5);
```

6 Apply the PDE Solver. The example calls `pdepe` with $m = 0$, the functions `pdex1pde`, `pdex1ic`, and `pdex1bc`, and the mesh defined by `x` and `t` at which `pdepe` is to evaluate the solution. The `pdepe` function returns the numerical solution in a three-dimensional array `sol`, where `sol(i,j,k)` approximates the k th component of the solution, u_k , evaluated at $(t(i), x(j))$.

```
m = 0;
sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t); This example uses @ to
pass pdex1pde, pdex1ic, and pdex1bc as function handles to pdepe.
```

Note See the `function_handle` (@), `func2str`, and `str2func` reference pages, and the “Function Handles” chapter of “Programming and Data Types” in the MATLAB documentation for information about function handles.

7 View the Results. Complete the example by displaying the results:

- Extract and display the first solution component. In this example, the solution u has only one component, but for illustrative purposes, the example “extracts” it from the three-dimensional array. The surface plot shows the behavior of the solution.

```

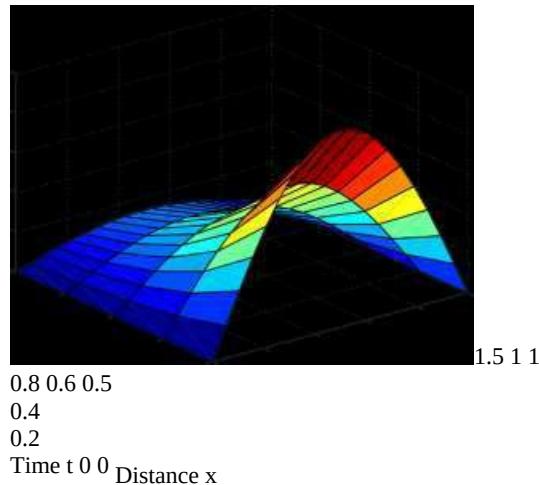
u = sol(:,:,1);
surf(x,t,u)
title('Numerical solution computed with 20 mesh points') xlabel('Distance x')
ylabel('Time t')

```

1
0.8
0.6
0.4
0.2

0
2

Numerical solution computed with 20 mesh points.



- Display a solution profile at t_f , the final value of t . In this example, $t_f = t = 2$. See “Evaluating the Solution at Specific Points” on page 15-93 for more information.

```

figure
plot(x,u(end,:))
title('Solution at t = 2')
xlabel('Distance x')
ylabel('u(x,2)')

```

Solution at t = 2 0.14

0.12
0.1
0.08
0.06
0.04
0.02
0

-0.020 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 Distance x

Evaluating the Solution at Specific Points

After obtaining and plotting the solution above, you might be interested in a solution profile for a particular value of t , or the time changes of the solution at a particular point x . The k th

`columnu(:,k)` (of the solution extracted in step 7) contains the time history of the solution at $x(k)$. The j th row $u(j,:)$ contains the solution profile $u(j)$.

Using the vectors x and $u(j,:)$, and the helper function `pdeval`, you can evaluate the solution u and its derivative $\frac{\partial u}{\partial x}$ at any set of points $xout$

`[uout,DuoutDx] = pdeval(m,x,u(j,:),xout)`

The example `pdedex3` uses `pdeval` to evaluate the derivative of the solution at $xout = 0$. See `pdeval` for details.

Improving PDE Solver Performance

The default integration properties in the MATLAB PDE solver are selected to handle common problems. In some cases, you can improve solver performance by overriding these defaults. You do this by supplying `pdepe` with one or more property values in an options structure.

`sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)`

Use `odeset` to create the options structure. Only those options of the underlying ODE solver shown in the following table are available for `pdepe`. The defaults obtained by leaving off the input argument `options` are generally satisfactory. “Improving ODE Solver Performance” on page 15-16 tells you how to create the structure and describes the properties.

PDE Property Categories Properties Category Property Name Error control

`RelTol`, `AbsTol`, `NormControl` Step-size `InitialStep`, `MaxStep`

Example: Electrodynamics Problem

This example illustrates the solution of a system of partial differential equations. The problem is taken from electrodynamics. It has boundary layers at both ends of the interval, and the solution changes rapidly for small t .

The PDEs are

$$\frac{\partial^2 u_1}{\partial x^2} = 0.024 - F(u_1 - u_2) \quad t \in [0, 2]$$

$$\frac{\partial^2 u_2}{\partial x^2} = 0.170 + F(u_1 - u_2) \quad t \in [0, 2]$$

where $F(y) = \exp(5.73y) - \exp(-11.46y)$. The equations hold on an interval $0 \leq x \leq 1$ for times $t \in [0, 2]$.

The solution u satisfies the initial conditions

$$u_1(x, 0) = 1$$

$$u_2(x, 0) = 0$$

and boundary conditions

$$u_1(0, t) = 0$$

$$u_2(1, t) = 1$$

$$u_1'(0, t) = 0$$

Note The demopdex4 contains the complete code for this example. The demo uses subfunctions to place all required functions in a single M-file. To run this example type pdex4 at the command line. See “PDE Solver Basic Syntax” on page 15-85 and “Representing PDE Problems” on page 15-88 for more information.

1 Rewrite the PDE. In the form expected bypdepe, the equations are

$$\begin{aligned} 1 \\ . \\ \overset{\circ}{F} \\ \frac{\frac{3}{8} u_1^3 - 0.024 F^3 u_1^{\frac{d}{c}} \frac{3}{8} x f - F f u_1 - u_2 f}{\frac{3}{8} t_u^3} = 1 \\ 2 \\ x 0.170 F^3 u_2^{\frac{d}{c}} \frac{3}{8} x f - F f u_1 - u_2 f \end{aligned}$$

The boundary conditions on the partial derivatives of u have to be written in terms of the flux. In the form expected bypdepe, the left boundary condition is

$$0 + 1 \cdot F 0.024 F^3 u_1^{\frac{d}{c}} \frac{3}{8} x f = 0$$

and the right boundary condition is

$$u_1 - 1 + 0 \cdot F 0.024 F^3 u_1^{\frac{d}{c}} \frac{3}{8} x f = 0$$

2 Code the PDE in MATLAB. After you rewrite the PDE in the form shown above, you can code it as a function thatpdepe can use. The function must be of the form

```
[c,f,s] = pdefun(x,t,u,dudx)
where c, f, and s correspond to the c, f, and s terms in Equation 15-3.
```

```
function [c,f,s] = pdex4pde(x,t,u,DuDx)
c = [1; 1];
f = [0.024; 0.17] .* DuDx;
y = u(1) - u(2);
F = exp(5.73*y)-exp(-11.47*y);
s = [-F; F];
```

3 Code the Initial Conditions Function. The initial conditions function must be of the form
 $u = icfun(x)$

The code below represents the initial conditions in the MATLAB function pdex4ic.

```
function u0 = pdex4ic(x);
u0 = [1; 0];
```

4 Code the Boundary Conditions Function. The boundary conditions functions must be of the form

```
[pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
```

The code below evaluates the components $p \frac{\partial u}{\partial x}$ and $q \frac{\partial u}{\partial t}$ (Equation 15-5) of the boundary conditions in the MATLAB functionpdex4bc.

```
function [pl,ql,pr,qr] = pdex4bc(xl,ul,xr,ur,t)
```

```

pl = [0; ul(2)];
ql = [1; 0];
pr = [ur(1)-1; 0];
qr = [0; 1];

```

5 Select Mesh Points for the Solution. The solution changes rapidly for small t . The program selects the step size in time to resolve this sharp change, but to see this behavior in the plots, output times must be selected accordingly. There are boundary layers in the solution at both ends of $[0,1]$, so mesh points must be placed there to resolve these sharp changes. Often some experimentation is needed to select the mesh that reveals the behavior of the solution.

```

x = [0 0.005 0.01 0.05 0.1 0.2 0.5 0.7 0.9 0.95 0.99 0.995 1]; t = [0 0.005 0.01 0.05 0.1 0.5 1 1.5 2];

```

6 Apply the PDE Solver. The example calls pdepe with $m = 0$, the functions pdex4pde, pdex4ic, and pdex4bc, and the mesh defined by x and t at which pdepe is to evaluate the solution.

The pdepe function returns the numerical solution in a three-dimensional array sol , where $\text{sol}(i,j,k)$ approximates the k th component of the solution, u_k , evaluated at (i) and $x(j)$.

```

m = 0;
sol = pdepe(m,@pdex4pde,@pdex4ic,@pdex4bc,x,t);

```

7 View the Results. The surface plots show the behavior of the solution components.

```

u1 = sol(:,:,1);
u2 = sol(:,:,2);

```

```

figure
surf(x,t,u1)
title('u1(x,t)')
xlabel('Distance x') ylabel('Time t')

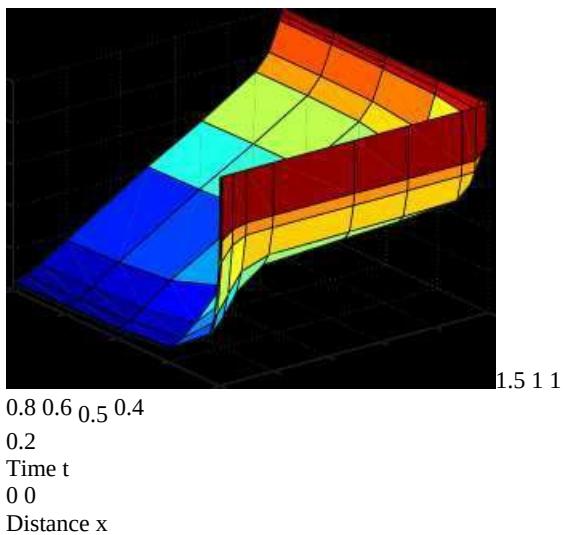
```

```

1
0.8
0.6
0.4
0.2

0
2
u1(x,t)

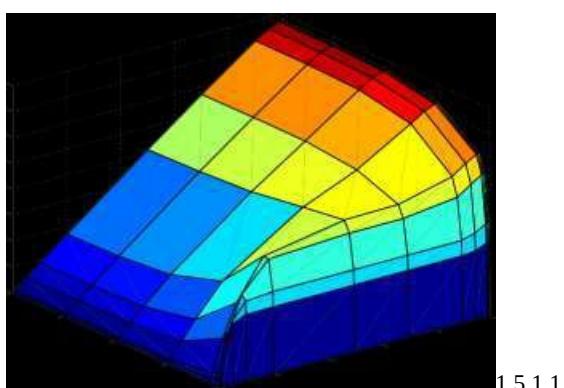
```



```
figure
surf(x,t,u2)
title('u2(x,t)')
xlabel('Distance x') ylabel('Time t')
```

0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1

0
2
u2(x,t)



0.8 0.6 0.5^{0.4}
0.2
Time t 0 0 Distance x

Selected Bibliography
Selected Bibliography

[1] Ascher, U., R. Mattheij, and R. Russell, *Numerical Solution of Boundary Value Problems for*

Ordinary Differential Equations , SIAM, Philadelphia, PA, 1995, p. 372.

[2] Cebeci, T. and H. B. Keller, "Shooting and Parallel Shooting Methods for Solving the Falkner-Skan Boundary-layer Equation," *J. Comp. Phys.*, Vol. 7, 1971, pp. 289-300.

[3] Hairer, E., and G. Wanner, *Solving Ordinary Differential Equations II, Stiff and Differential-Algebraic Problems* , Springer-Verlag, Berlin, 1991, pp. 5-8. [4] Hindmarsh, A. C., "LSODE and LSODI, Two New Initial Value Ordinary Differential Equation Solvers," *SIGNUM Newsletter*, Vol. 15, 1980, pp. 10-11.

[5] Hindmarsh, A. C., and G. D. Byrne, "Applications of EPISODE: An Experimental Package for the Integration of Ordinary Differential Equations," *Numerical Methods for Differential Systems* , L. Lapidus and W. E. Schiesser eds., Academic Press, Orlando, FL, 1976, pp 147-166.

[6] Shampine, L. F., *Numerical Solution of Ordinary Differential Equations* , Chapman & Hall Mathematics, 1994.

[7] Shampine, L. F., and M. K. Gordon, *Computer Solution of Ordinary Differential Equations* , W.H. Freeman & Co., 1975.

[8] Skeel, R. D. and M. Berzins, "A Method for the Spatial Discretization of Parabolic Equations in One Space Variable," *SIAM Journal on Scientific and Statistical Computing* , Vol. 11, 1990, pp.1-32.

16

Sparse Matrices

Function Summary	16-3
Introduction	9-5
Sparse Matrix Storage	9-5 Introduction
.....	16-6
Creating Sparse Matrices.....	9-6
Sparse Matrix Storage	16-6
Importing Sparse Matrices from Outside MATLAB	9-10
General Storage Information	16-7 Creating Sparse
Matrices.....	16-7
Viewing Sparse Matrices	9-11
Importing Sparse Matrices from Outside MATLAB	16-12
Information About Nonzero Elements	9-11
Viewing Sparse Matrices	16-13
Viewing Sparse Matrices Graphically.....	9-13
Information About Nonzero Elements	16-13
The find Function and Sparse Matrices.....	9-14
Viewing Sparse Matrices Graphically.....	16-15 The find Function and
Sparse Matrices.....	16-16
Example: Adjacency Matrices and Graphs.....	9-15

Graphing Using Adjacency Matrices.....	9-15
Example: Adjacency Matrices and Graphs.....	16-17
Introduction to Adjacency Matrices	16-17
An Airflow Model.....	9-21
Graphing Using Adjacency Matrices.....	16-18 The Bucky Ball.....
Sparse Matrix Operations	9-23
An Airflow Model.....	16-23
Computational Considerations.....	9-23
Standard Mathematical Operations.....	9-23
Sparse Matrix Operations	16-25
Computational Considerations.....	16-25
Factorization	9-27
Standard Mathematical Operations.....	16-25
Simultaneous Linear Equations.....	9-33
Permutation and Reordering.....	16-26
Eigenvalues and Singular Values	9-36
Factorization	16-30
Simultaneous Linear Equations.....	16-36
Eigenvalues and Singular Values	16-39
Selected Bibliography	16-42

MATLAB supports *sparse matrices* , matrices that contain a small proportion of nonzero elements. This characteristic provides advantages in both matrix storage space and computation time.

This chapter explains how to create sparse matrices in MATLAB, and how to use them in both specialized and general mathematical operations. It includes:

Function Summary

A summary of the sparse matrix functions

Introduction

An introduction to sparse matrices in MATLAB

Viewing Sparse Matrices

How to obtain quantitative and graphical information about sparse matrices

Example: Adjacency Matrices and Graphs

Examples that use adjacency matrices to demonstrate sparse matrices

Sparse Matrix Operations

A discussion of functions that perform operations specific to sparse matrices

Selected Bibliography

Published materials that support concepts described in this chapter

Function Summary

The sparse matrix functions are located in thespfun directory in the MATLABtoolbox directory.

Function Summary Category Function Description Elementary sparse matrices
speye Sparse identity matrix.
sprand Sparse uniformly distributed random matrix. sprandn Sparse normally distributed random matrix. sprandsym Sparse random symmetric matrix. spdiags Sparse matrix formed from diagonals. Full to sparse conversion
sparse Create sparse matrix.
full Convert sparse matrix to full matrix. find Find indices of nonzero elements. spconvert Import from sparse matrix external format. Working with sparse matrices
nnz Number of nonzero matrix elements.

nonzeros Nonzero matrix elements. nzmax Amount of storage allocated for nonzero matrix elements. spones Replace nonzero sparse matrix elements with ones. spalloc Allocate space for sparse matrix. issparse True for sparse matrix. spfun Apply function to nonzero matrix elements. spy Visualize sparsity pattern.

Function Summary (Continued) Category Function Description Graph theory gplot Plot graph, as in “graph theory.” etree Elimination tree. etreeplot Plot elimination tree. treelayout Layout tree or forest. treeplot Plot picture of tree. Reordering algorithms
colamd Column approximate minimum degree permutation.

colmmd Column minimum degree permutation. symamd Symmetric approximate minimum degree permutation. symmmd Symmetric minimum degree permutation. symrcm Symmetric reverse Cuthill-McKee permutation. colperm Column permutation. randperm Random permutation. dmperm Dulmage-Mendelsohn permutation. Linear algebra eigs A few eigenvalues. svds A few singular values. luinc Incomplete LU factorization. cholinc Incomplete Cholesky factorization. normest Estimate the matrix 2-norm. condest 1-norm condition number estimate. sprank Structural rank.

Function Summary (Continued) Category Function Description Linear equations (iterative methods)

bicg BiConjugate Gradients Method.
bicgstab BiConjugate Gradients Stabilized Method. cgs Conjugate Gradients Squared Method.
gmres Generalized Minimum Residual Method. lsqr LSQR implementation of Conjugate Gradients on the Normal Equations.
minres Minimum Residual Method. pcg Preconditioned Conjugate Gradients Method. qmr Quasi-Minimal Residual Method. symmlq Symmetric LQ method Miscellaneous spaugment Form least squares augmented system. spparms Set parameters for sparse matrix routines. symbfact Symbolic factorization analysis.

Introduction

Sparse matrices are a special class of matrices that contain a significant number of zero-valued elements. This property allows MATLAB to:

- Store only the nonzero elements of the matrix, together with their indices.
- Reduce computation time by eliminating operations on zero elements. This section provides information about:

- Sparse matrix storage
- General storage information
- Creating sparse matrices
- Importing sparse matrices

Sparse Matrix Storage

For full matrices, MATLAB stores internally every matrix element. Zero-valued elements require the same amount of storage space as any other matrix element. For sparse matrices, however, MATLAB stores only the nonzero elements and their indices. For large matrices with a high percentage of zero-valued elements, this scheme significantly reduces the amount of memory required for data storage.

MATLAB uses three arrays internally to store sparse matrices with real elements. Consider an $n \times n$ sparse matrix with nnz nonzero entries stored in arrays of length $nzmax$:

- The first array contains all the nonzero elements of the array in floating-point format. The length of this array is equal to $nzmax$.
- The second array contains the corresponding integer row indices for the nonzero elements stored in the first nnz entries. This array also has length equal to $nzmax$.
- The third array contains n integer pointers to the start of each column in the other arrays and an additional pointer that marks the end of those arrays. The length of the third array is $n+1$.

This matrix requires storage for $nzmax$ floating-point numbers and $nzmax+n+1$ integers. At 8 bytes per floating-point number and 4 bytes per integer, the total number of bytes required to store a sparse matrix is

$$8*nzmax + 4*(nzmax+n+1)$$

Sparse matrices with complex elements are also possible. In this case, MATLAB uses a fourth array with nnz elements to store the imaginary parts of the nonzero elements. An element is considered nonzero if either its real or imaginary part is nonzero.

General Storage Information

The `whos` command provides high-level information about matrix storage, including size and storage class. For example, this `whos` listing shows information about sparse and full versions of the same matrix.

```
whos
Name      Size    Bytes Class
M_full   1100x1100 9680000 double array
M_sparse 1100x1100 4404 sparse array
Grand total is 1210000 elements using 9684404 bytes
```

Notice that the number of bytes used is much less in the sparse case, because zero-valued elements are not stored. In this case, the density of the sparse matrix is $4404/9680000$, or approximately .00045%.

Creating Sparse Matrices

MATLAB never creates sparse matrices automatically. Instead, you must determine if a matrix contains a large enough percentage of zeros to benefit from sparse techniques.

The *density* of a matrix is the number of non-zero elements divided by the total number of matrix elements. Matrices with very low density are often good candidates for use of the sparse format.

Converting Full to Sparse

You can convert a full matrix to sparse storage using `thesparse` function with a single argument.

```
S = sparse(A)
```

For example

```
A = [ 0 0 0 5  
      0 2 0 0  
      1 3 0 0  
      0 0 4 0];
```

```
S = sparse(A)
```

produces

```
S =
```

```
(3,1) 1  
(2,2) 2  
(3,2) 3  
(4,3) 4  
(1,4) 5
```

The printed output lists the nonzero elements of S, together with their row and column indices. The elements are sorted by columns, reflecting the internal data structure.

You can convert a sparse matrix to full storage using the `full` function, provided the matrix order is not too large. For example `A = full(S)` reverses the example conversion.

Converting a full matrix to sparse storage is not the most frequent way of generating sparse matrices. If the order of a matrix is small enough that full storage is possible, then conversion to sparse storage rarely offers significant savings.

Creating Sparse Matrices Directly

You can create a sparse matrix from a list of nonzero elements using `thesparse` function with five arguments.

```
S = sparse(i,j,s,m,n)
```

i and j are vectors of row and column indices, respectively, for the nonzero elements of the matrix. s is a vector of nonzero values whose indices are specified by the corresponding (i,j) pairs. m is the row dimension for the resulting matrix, and n is the column dimension.

The matrix S of the previous example can be generated directly with

```
S = sparse([3 2 3 4 1],[1 2 2 3 4],[1 2 3 4 5],4,4)
```

```
S =
```

```
(3,1) 1  
(2,2) 2  
(3,2) 3
```

(4,3) 4
(1,4) 5

The sparse command has a number of alternate forms. The example above uses a form that sets the maximum number of nonzero elements in the matrix to length(s). If desired, you can append a sixth argument that specifies a larger maximum, allowing you to add nonzero elements later without reallocating the sparse matrix.

Example: Generating a Second Difference Operator

The matrix representation of the second difference operator is a good example of a sparse matrix. It is a tridiagonal matrix with -2s on the diagonal and 1s on the super- and subdiagonal. There are many ways to generate it – here's one possibility.

```
D = sparse(1:n,1:n,-2);  
E = sparse(2:n,1:n-1,ones(1,n-1),n,n);  
S = E+D+E'
```

For n = 5, MATLAB responds with
S =

```
(1,1) -2  
(2,1) 1  
(1,2) 1  
(2,2) -2  
(3,2) 1  
(2,3) 1  
(3,3) -2  
(4,3) 1  
(3,4) 1  
(4,4) -2  
(5,4) 1  
(4,5) 1  
(5,5) -2
```

Now F = full(S) displays the corresponding full matrix.

```
F = full(S)  
F =
```

```
-2 1 0 0 0 1 -2 1 0 0 0 1 -2 1 0 0 0 1 -2
```

Creating Sparse Matrices from Their Diagonal Elements

Creating sparse matrices based on their diagonal elements is a common operation, so the function spdiags handles this task. Its syntax is

```
S = spdiags(B,d,m,n)
```

To create an output matrix S of size m-by-n with elements on the diagonals:

- B is a matrix of size min(m,n)-by-p. The columns of B are the values to populate the diagonals

ofS.

- d is a vector of lengthp whose integer elements specify which diagonals ofS to populate. That is, the elements in columnj ofB fill the diagonal specified by elementj ofd.

Note If a column of B is longer than the diagonal it's replacing, super-diagonals are taken from the lower part of the column ofB, and sub-diagonals are taken from the upper part of the column of B. As an example, consider the matrixB and the vectord.

```
B = [ 41 11 0  
      52 22 0  
      63 33 13  
      74 44 24 ];
```

```
d = [-3  
      0  
      2];
```

Use these matrices to create a 7-by-4 sparse matrixA.

```
A = spdiags(B,d,7,4)  
A =
```

```
(1,1) 11  
(4,1) 41  
(2,2) 22  
(5,2) 52  
(1,3) 13  
(3,3) 33  
(6,3) 63  
(2,4) 24  
(4,4) 44  
(7,4) 74
```

In its full form,A looks like this. full(A)

```
ans =
```

```
11 0 13 0  
0 22 0 24  
0 0 33 0  
41 0 0 44  
0 52 0 0  
0 0 63 0  
0 0 0 74
```

spdiags can also extract diagonal elements from a sparse matrix, or replace matrix diagonal elements with new values. Type help spdiags for details.

Importing Sparse M atrices from Outside M ATLAB

You can import sparse matrices from computations outside MATLAB. Use the spconvert

function in conjunction with the `load` command to import text files containing lists of indices and nonzero elements. For example, consider a three-column text file `T.dat` whose first column is a list of row indices, second column is a list of column indices, and third column is a list of nonzero values. These statements load `T.dat` into MATLAB and convert it into a sparse matrix `S`:

```
load T.dat  
S = spconvert(T)
```

The `save` and `load` commands can also process sparse matrices stored as binary data in MAT-files.

Vie wing Sparse M atrices

MATLAB provides a number of functions that let you get quantitative or graphical information about sparse matrices.

This section provides information about:

- Obtaining information about nonzero elements
- Viewing graphs of sparse matrices
- Finding indices and values of nonzero elements

Inform ation About N onzero Elements

There are several commands that provide high-level information about the nonzero elements of a sparse matrix:

- `nnz` returns the number of nonzero elements in a sparse matrix.
- `nonzeros` returns a column vector containing all the nonzero elements of a sparse matrix.
- `nzmax` returns the amount of storage space allocated for the nonzero entries of a sparse matrix.

To try some of these, load the supplied sparse matrix `west0479`, one of the Harwell-Boeing collection.

```
load west0479
```

```
whos
```

Name	Size	Bytes	Class
west0479	479x479	24576	sparse array

This matrix models an eight-stage chemical distillation column. Try these commands.

```
nnz(west0479)  
ans =  
1887  
format short e  
west0479 =  
  
(25,1) 1.0000e+00  
(31,1) -3.7648e-02  
(87,1) -3.4424e-01  
(26,2) 1.0000e+00  
(31,2) -2.4523e-02  
(88,2) -3.7371e-01  
(27,3) 1.0000e+00  
(31,3) -3.6613e-02
```

```
(89,3) -8.3694e-01
(28,4) 1.3000e+02
.
.
.

nonzeros(west0479); ans =
1.0000e+00
-3.7648e-02
-3.4424e-01

1.0000e+00
-2.4523e-02
-3.7371e-01

1.0000e+00
-3.6613e-02
-8.3694e-01

1.3000e+02 .
.
.
```

Note Use **Ctrl+C** to stop the nonzeros listing at any time. Note that initially `nnz` has the same value as `nzmax` by default. That is, the number of nonzero elements is equivalent to the number of storage locations allocated for nonzeros. However, MATLAB does not dynamically release memory if you zero out additional array elements. Changing the value of some matrix elements to zero changes the value of `nnz`, but not that of `nzmax`.

However, you can add as many nonzero elements to the matrix as desired. You are not constrained by the original value of `nzmax`.

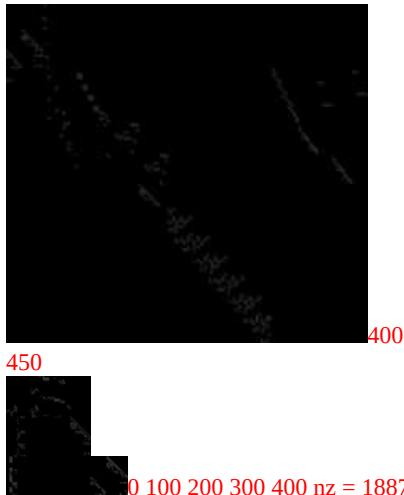
Vie w ing Sparse M atrices Graphically

It is often useful to use a graphical format to view the distribution of the nonzero elements within a sparse matrix. MATLAB's `spy` function produces a template view of the sparsity structure, where each point on the graph represents the location of a nonzero array element.

For example,

```
spy(west0479)
```

```
0
50
100
150
200
250
300
350
```



The find Function and Sparse M atrices

For any matrix, full or sparse, the `find` function returns the indices and values of nonzero elements. Its syntax is

`[i,j,s] = find(S)`

`find` returns the row indices of nonzero values in vector `i`, the column indices in vector `j`, and the nonzero values themselves in the vectors. The example below uses `find` to locate the indices and values of the nonzeros in a sparse matrix. The `sparse` function uses the `find` output, together with the size of the matrix, to recreate the matrix.

`[i,j,s] = find(S)`

`[m,n] = size(S)`

`S = sparse(i,j,s,m,n)`

Ex ample: Adjacency M atrices and Graphs

This section includes:

- An introduction to adjacency matrices
- Instructions for graphing adjacency matrices with `gplot`
- A Bucky ball example, including information about using `spy` plots to illustrate fill-in and distance
- An airflow model example

Introduction to Adjacency M atrices

The formal mathematical definition of a *graph* is a set of points, or nodes, with specified connections between them. An economic model, for example, is a graph with different industries as the nodes and direct economic ties as the connections. The computer software industry is connected to the computer hardware industry, which, in turn, is connected to the semiconductor industry, and so on.

This definition of a graph lends itself to matrix representation. The *adjacency matrix* of an *undirected* graph is a matrix whose (i,j) th and (j,i) th entries are 1 if node i is connected to node j , and 0 otherwise. For example, the adjacency matrix for a diamond-shaped graph looks like

1
A =

0 1 0 1
1 0 1 0



$^4 0 1 0 1^2$
1 0 1 0

3

Since most graphs have relatively few connections per node, most adjacency matrices are sparse. The actual locations of the nonzero elements depend on how the nodes are numbered. A change in the numbering leads to permutation of the rows and columns of the adjacency matrix, which can have a significant effect on both the time and storage requirements for sparse matrix computations.

Graphing Using Adjacency Matrices

MATLAB's gplot function creates a graph based on an adjacency matrix and a related array of coordinates. To trygplot, create the adjacency matrix shown above by entering

A = [0 1 0 1; 1 0 1 0; 0 1 0 1; 1 0 1 0];

The columns ofgplot's coordinate array contain the Cartesian coordinates for the corresponding node. For the diamond example, create the array by entering xy = [1 3; 2 1; 3 3; 2 5];

This places the first node at location (1,3), the second at location(2,1), the third at location(3,3), and the fourth at location(2,5). To view the resulting graph, enter

gplot(A,xy)

The Bucky Ball

One interesting construction for graph analysis is the *Bucky ball*. This is composed of 60 points distributed on the surface of a sphere in such a way that the distance from any point to its nearest neighbors is the same for all the points. Each point has exactly three neighbors. The Bucky ball models four different physical objects:

- The geodesic dome popularized by Buckminster Fuller
- The C₆₀ molecule, a form of pure carbon with 60 atoms in a nearly spherical configuration
- In geometry, the truncated icosahedron
- In sports, the seams in a soccer ball

The Bucky ball adjacency matrix is a 60-by-60 symmetric matrix B.B has three nonzero elements in each row and column, for a total of 180 nonzero values. This matrix has important

applications related to the physical objects listed earlier. For example, the eigenvalues of B are involved in studying the chemical properties of C_{60} .

To obtain the Bucky ball adjacency matrix, enter

```
B = bucky;
```

At order 60, and with a density of 5%, this matrix does not require sparse techniques, but it does provide an interesting example.

You can also obtain the coordinates of the Bucky ball graph using $[B,v] = \text{bucky};$

This statement generates v , a list of xyz-coordinates of the 60 points in 3-space equidistributed on the unit sphere. The function gplot uses these points to plot the Bucky ball graph.

```
gplot(B,v)
axis equal
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1 -0.5 0 0.5 1
```

It is not obvious how to number the nodes in the Bucky ball so that the resulting adjacency matrix reflects the spherical and combinatorial symmetries of the graph. The numbering used by bucky.m is based on the pentagons inherent in the ball's structure.

The vertices of one pentagon are numbered 1 through 5, the vertices of an adjacent pentagon are numbered 6 through 10, and so on. The picture on the following page shows the numbering of half of the nodes (one hemisphere); the numbering of the other hemisphere is obtained by a reflection about the equator. Use gplot to produce a graph showing half the nodes. You can add the node numbers using a for loop.

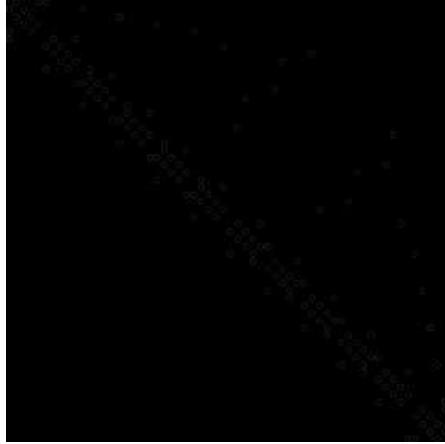
```
k = 1:30;
gplot(B(k,k),v);
axis square
for j = 1:30, text(v(j,1),v(j,2), int2str(j)); end
```

```
1 14 15 13 0.8
18 17 0.6 11 12
0.4 19 16 2 10 0.2 20 3 9
0 1 6
-0.2 22 4 8 5 7 -0.4 23 21
-0.6 26 30 24 25
-0.8
27 29
-1 28
-1 -0.8 -0.6 -0.4 -0.2 0 0.2 0.4 0.6 0.8 1
```

To view a template of the nonzero locations in the Bucky ball's adjacency matrix, use the spy function:

```
spy(B)
```

```
0  
10  
20  
30  
40  
50
```



```
60  
0 10 20 30 40 50 60  
nz = 180
```

The node numbering that this model uses generates a spy plot with 12 groups of five elements, corresponding to the 12 pentagons in the structure. Each node is connected to two other nodes within its pentagon and one node in some other pentagon. Since the nodes within each pentagon have consecutive numbers, most of the elements in the first super- and sub-diagonals of B are nonzero. In addition, the symmetry of the numbering about the equator is apparent in the symmetry of the spy plot about the antidiagonal.

Graphs and Characteristics of Sparse Matrices

Spy plots of the matrix powers of B illustrate two important concepts related to sparse matrix operations, fill-in and distance. Spy plots help illustrate these concepts.

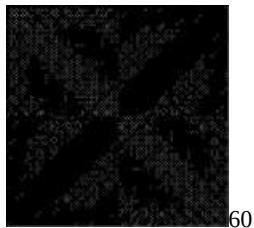
```
spy(B^2)  
spy(B^3)  
spy(B^4)  
spy(B^8)
```

```
0  
10  
20  
30  
40  
50
```



60
0 20 40 60 nz = 420

0
10
20
30
40
50

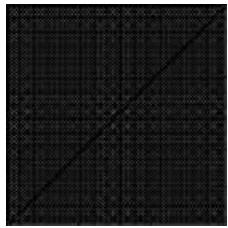


60
0
10
20
30
40
50



60
0 20 40 60 nz = 780

0
10
20
30
40
50



60

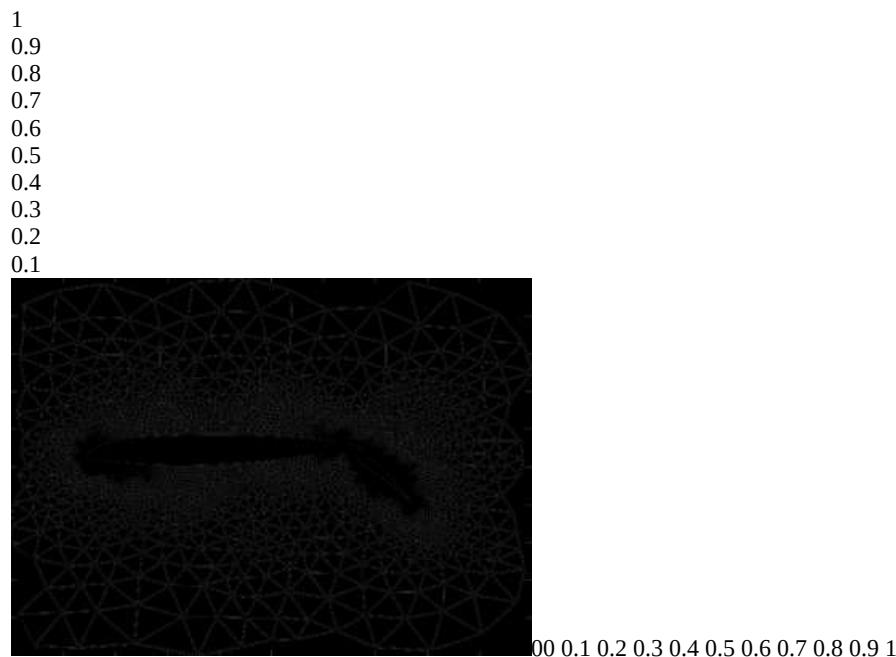
```
0 20 40 60 nz = 1380  
0 20 40 60 nz = 3540
```

Fill-in is generated by operations like matrix multiplication. The product of two or more matrices usually has more nonzero entries than the individual terms, and so requires more storage. As p increases, B^p fills in and `spy(Bp)` gets more dense.

The *distance* between two nodes in a graph is the number of steps on the graph necessary to get from one node to the other. The spy plot of the p -th power of B shows the nodes that are a distance p apart. As p increases, it is possible to get to more and more nodes in p steps. For the Bucky ball, B^8 is almost completely full. Only the antidiagonal is zero, indicating that it is possible to get from any node to any other node, except the one directly opposite it on the sphere, in eight steps.

An Airflow Model

A calculation performed at NASA's Research Institute for Applications of Computer Science involves modeling the flow over an airplane wing with two trailing flaps.



In a two-dimensional model, a triangular grid surrounds a cross section of the wing and flaps. The partial differential equations are nonlinear and involve several unknowns, including hydrodynamic pressure and two components of velocity. Each step of the nonlinear iteration requires the solution of a sparse linear system of equations. Since both the connectivity and the geometric location of the grid points are known, the `plot` function can produce the graph shown above.

In this example, there are 4253 grid points, each of which is connected to between 3 and 9 others, for a total of 28831 nonzeros in the matrix, and a density equal to 0.0016. This spy plot shows that the node numbering yields a definite band structure.

The Laplacian of the mesh.

0
500
.....
.....
.....
.....
1000 ..
.....
.....
1500 ..
.....
.....
.....
2000
.....
.....
.....
.....
.....
2500
.....
.....
.....
.....
.....
.....
.....
3000 ..
.....
.....
.....
3500 ..
.....
.....
.....
4000 ..
.....
0 1000 2000
3000 4000
.....
.....
nz = 28831
Sparse Matrix Operations

Sparse Matrix Operations

Most of MATLAB's standard mathematical functions work on sparse matrices just as they do on

full matrices. In addition, MATLAB provides a number of functions that perform operations specific to sparse matrices. This section discusses:

- Computational considerations
- Standard mathematical operations
- Permutation and reordering
- Factorization
- Simultaneous linear equations
- Eigenvalues and singular values

Computational Considerations

The computational complexity of sparse operations is proportional to nnz , the number of nonzero elements in the matrix. Computational complexity also depends linearly on the row size m and column size n of the matrix, but is independent of the product $m \times n$, the total number of zero and nonzero elements.

The complexity of fairly complicated operations, such as the solution of sparse linear equations, involves factors like ordering and fill-in, which are discussed in the previous section. In general, however, the computer time required for a sparse matrix operation is proportional to the number of arithmetic operations on nonzero quantities.

Standard Mathematical Operations

Sparse matrices propagate through computations according to these rules:

- Functions that accept a matrix and return a scalar or vector always produce output in full storage format. For example, the `size` function always returns a full vector, whether its input is full or sparse.
- Functions that accept scalars or vectors and return matrices, such as `zeros`, `ones`, `rand`, and `eye`, always return full results. This is necessary to avoid introducing sparsity unexpectedly. The sparse analog of `zeros(m,n)` is `sparse(m,n)`. The sparse analogs of `rand` and `eye` are `rand` and `speye`, respectively. There is no sparse analog for the function `ones`.
- Unary functions that accept a matrix and return a matrix or vector preserve the storage class of the operand. If S is a sparse matrix, then `chol(S)` is also a sparse matrix, and `diag(S)` is a sparse vector. Columnwise functions such as `max` and `sum` also return sparse vectors, even though these vectors may be entirely nonzero. Important exceptions to this rule are the `sparse` and `full` functions.
- Binary operators yield sparse results if both operands are sparse, and full results if both are full. For mixed operands, the result is full unless the operation preserves sparsity. If S is sparse and F is full, then $S+F$, $S*F$, and $F\backslash S$ are full, while $S.*F$ and $S\&F$ are sparse. In some cases, the result might be sparse even though the matrix has few zero elements.
- Matrix concatenation using either the `cat` function or square brackets produces sparse results for mixed operands.
- Submatrix indexing on the right side of an assignment preserves the storage format of the operand unless the result is a scalar. $T = S(i,j)$ produces a sparse result if S is sparse and either i or j is a vector. It produces a full scalar if both i and j are scalars. Submatrix indexing on the left, as in $T(i,j) = S$, does not change the storage format of the matrix on the left.

Permutation and Reordering

A permutation of the rows and columns of a sparse matrix S can be represented in two ways:

- A permutation matrix P acts on the rows of S as P^*S or on the columns as

S^*P' .

- A permutation vector p , which is a full vector containing a permutation of $1:n$, acts on the rows of S as $S(p,:)$, or on the columns as $S(:,p)$.

For example, the statements

```
p = [1 3 4 2 5]
I = eye(5,5);
P = I(p,:);
e = ones(4,1);
S = diag(11:11:55) + diag(e,1) + diag(e,-1)
```

produce

```
p =
1 3 4 2 5
P =
```

```
1 0 0 0 0
0 0 1 0 0
0 0 0 1 0
0 1 0 0 0
0 0 0 0 1
```

```
S =
```

```
11 1 0 0 0
1 22 1 0 0
0 1 33 1 0
0 0 1 44 1
0 0 0 1 55
```

You can now try some permutations using the permutation vector p and the permutation matrix P .
For example, the statements $S(p,:)$ and P^*S produce

```
ans =
```

```
11 1 0 0 0
0 1 33 1 0
0 0 1 44 1
1 22 1 0 0
0 0 0 1 55
```

Similarly, $S(:,p)$ and S^*P' produce

```
ans =
```

```
11 0 0 1 0  
1 1 0 22 0  
0 33 1 1 0  
0 1 44 0 1  
0 0 1 0 55
```

If P is a sparse matrix, then both representations use storage proportional to n and you can apply either toS in time proportional to $\text{nnz}(S)$. The vector representation is slightly more compact and efficient, so the various sparse matrix permutation routines all return full row vectors with the exception of the pivoting permutation in LU (triangular) factorization, which returns a matrix compatible with earlier versions of MATLAB.

To convert between the two representations, let $I = \text{speye}(n)$ be an identity matrix of the appropriate size. Then,

```
P = I(p,:)  
P' = I(:,p)  
p = (1:n)*P'  
p = (P*(1:n))'
```

The inverse of P is simply $R = P'$. You can compute the inverse of P with $r(p) = 1:n$.

$r(p) = 1:5$

$r =$

1 4 2 3 5

Reordering for Sparsity

Reordering the columns of a matrix can often make its LU or QR factors sparser. Reordering the rows and columns can often make its Cholesky factors sparser. The simplest such reordering is to sort the columns by nonzero count. This is sometimes a good reordering for matrices with very irregular structures, especially if there is great variation in the nonzero counts of rows or columns.

The function $p = \text{colperm}(S)$ computes this column-count permutation. The colperm M-file has only a single line.

```
[ignore,p] = sort(full(sum(spones(S))));
```

This line performs these steps:

- 1** The inner call to spones creates a sparse matrix with ones at the location of every nonzero element in S .
- 2** The sum function sums down the columns of the matrix, producing a vector that contains the count of nonzeros in each column.
- 3** full converts this vector to full storage format.
- 4** sort sorts the values in ascending order. The second output argument from sort is the permutation that sorts this vector.

Reordering to Reduce Bandwidth

The reverse Cuthill-McKee ordering is intended to reduce the profile or bandwidth of the matrix.

It is not guaranteed to find the smallest possible bandwidth, but it usually does. The function `symrcm(A)` actually operates on the nonzero structure of the symmetric matrix $A + A'$, but the result is also useful for asymmetric matrices. This ordering is useful for matrices that come from one-dimensional problems or problems that are in some sense “long and thin.”

Minimum Degree Ordering

The degree of a node in a graph is the number of connections to that node. This is the same as the number of off-diagonal nonzero elements in the corresponding row of the adjacency matrix. The minimum degree algorithm generates an ordering based on how these degrees are altered during Gaussian elimination or Cholesky factorization. It is a complicated and powerful algorithm that usually leads to sparser factors than most other orderings, including column count and reverse Cuthill-McKee.

MATLAB functions implement two methods for each of two types of matrices: `symamd` and `symmmd` for symmetric matrices, `colamd` and `colmmd` for nonsymmetric matrices. `colamd` and `colmmd` also work for symmetric matrices of the form A^*A' or A'^*A .

Because the most time-consuming part of a minimum degree ordering algorithm is keeping track of the degree of each node, all four functions use an approximation to the degree, rather than the exact degree. As a result:

- Factorizations obtained using `colmmd` and `symmmd` tend to have more nonzero elements than if the implementation used exact degrees.
- `colamd` and `symamd` use a tighter approximation than `colmmd` and `symmmd`. They generate orderings that are as good as could be obtained using exact degrees.
- `colamd` and `symamd` are faster than `colmmd` and `symmmd`, respectively. This is true particularly for very large matrices.

You can change various parameters associated with details of the algorithms using the `spparms` function.

For details on the algorithms used by `colmmd` and `symmmd`, see [4]. For details on the algorithms used by `colamd` and `symamd`, see [5]. The approximate degree used in `colamd` and `symamd` is based on [1].

Factorization

This section discusses four important factorization techniques for sparse matrices:

- LU, or triangular, factorization
- Cholesky factorization
- QR, or orthogonal, factorization
- Incomplete factorizations

LU Factorization

If S is a sparse matrix, the statement below returns three sparse matrices L , U , and P such that $P^*S = L^*U$.

`[L,U,P] = lu(S)`

`lu` obtains the factors by Gaussian elimination with partial pivoting. The permutation matrix P has only non-zero elements. As with dense matrices, the statement `[L,U] = lu(S)` returns a permuted

unit lower triangular matrix and an upper triangular matrix whose product is S . By itself, $\text{lu}(S)$ returns L and U in a single matrix without the pivot information.

The sparse LU factorization does not pivot for sparsity, but it does pivot for numerical stability. In fact, both the sparse factorization (line 1) and the full factorization (line 2) below produce the same L and U , even though the time and storage requirements might differ greatly.

```
[L,U] = lu(S) % Sparse factorization  
[L,U] = sparse(lu(full(S))) % Full factorization
```

You can control pivoting in sparse matrices using $\text{lu}(S, \text{thresh})$

where thresh is a pivot threshold in $[0,1]$. Pivoting occurs when the diagonal entry in a column has magnitude less than thresh times the magnitude of any sub-diagonal entry in that column. $\text{thresh} = 0$ forces diagonal pivoting. $\text{thresh} = 1$ is the default.

MATLAB automatically allocates the memory necessary to hold the sparse L and U factors during the factorization. MATLAB does not use any symbolic LU prefactorization to determine the memory requirements and set up the data structures in advance.

Reordering and Factorization. If you obtain a good column permutation p that reduces fill-in, perhaps from symrcm or colamd , then computing $\text{lu}(S(:,p))$ takes less time and storage than computing $\text{lu}(S)$. Two permutations are the symmetric reverse Cuthill-McKee ordering and the symmetric minimum degree ordering.

```
r = symrcm(B);  
m = symamd(B);
```

The three spy plots produced by the lines below show the three adjacency matrices of the Bucky Ball graph with these three different numberings. The local, pentagon-based structure of the original numbering is not evident in the other three.

```
spy(B)  
spy(B(r,r))  
spy(B(m,m))
```

```
Original  
0  
10  
20  
30  
40  
50  
60
```



0 10 20 30 40 50 60 nz = 180
ReverseCuthill-McKee

0
10
20
30
40
50

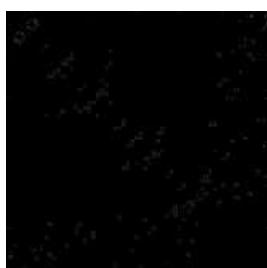
60



0 10 20 30 40 50 60 nz = 180
Minimum Degree

0
10
20
30
40
50

60



0 10 20 30 40 50 60 nz = 180

The reverse Cuthill-McKee ordering, r, reduces the bandwidth and concentrates all the nonzero elements near the diagonal. The approximate minimum degree ordering,m, produces a fractal-like structure with large blocks of zeros.

To see the fill-in generated in the LU factorization of the Bucky ball, use `speye(n,n)`, the sparse identity matrix, to insert -3s on the diagonal of B . $B = B - 3 * \text{speye}(n,n)$;

Since each row sum is now zero, this new B is actually singular, but it is still instructive to compute its LU factorization. When called with only one output argument, `lu` returns the two triangular factors, L and U , in a single sparse matrix. The number of nonzeros in that matrix is a measure of the time and storage required to solve linear systems involving B . Here are the nonzero counts for the three permutations being considered.

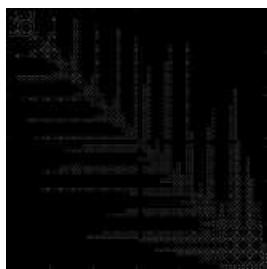
Original lu(B) 1022 Reverse Cuthill-McKee lu($B(r,r)$) 968 Approximate minimum degree lu($B(m,m)$) 636

Even though this is a small example, the results are typical. The original numbering scheme leads to the most fill-in. The fill-in for the reverse Cuthill-McKee ordering is concentrated within the band, but it is almost as extensive as the first two orderings. For the minimum degree ordering, the relatively large blocks of zeros are preserved during the elimination and the amount of fill-in is significantly less than that generated by the other orderings. Thespy plots below reflect the characteristics of each reordering.

Original

0
10
20
30
40
50

60



0 10 20 30 40 50 60 nz = 1022

ReverseCuthill-McKee

0
10
20
30
40
50

60

100



0 10 20 30 40 50 60 nz = 968
0 Minimum Degree

10
20
30
40
50

60



0 10 20 30 40 50 60 nz = 636

Cholesky Factorization If S is a symmetric (or Hermitian), positive definite, sparse matrix, the statement below returns a sparse, upper triangular matrix R so that $R^*R = S$.

$R = chol(S)$

chol does not automatically pivot for sparsity, but you can compute minimum degree and profile limiting permutations for use with $chol(S(p,p))$.

Since the Cholesky algorithm does not use pivoting for sparsity and does not require pivoting for numerical stability, chol does a quick calculation of the amount of memory required and allocates all the memory at the start of the factorization. You can use symbfact, which uses the same algorithm as chol, to calculate how much memory is allocated.

QR Factorization

MATLAB computes the complete QR factorization of a sparse matrix S with $[Q,R] = qr(S)$

but this is usually impractical. The orthogonal matrix Q often fails to have a high proportion of zero elements. A more practical alternative, sometimes known as “the Q-less QR factorization,” is available.

With one sparse input argument and one output argument

$R = qr(S)$

returns just the upper triangular portion of the QR factorization. The matrix R provides a Cholesky factorization for the matrix associated with the normal equations,

$$R'^*R = S'^*S$$

However, the loss of numerical information inherent in the computation of S'^*S is avoided. With two input arguments having the same number of rows, and two output arguments, the statement

$$[C,R] = qr(S,B)$$

applies the orthogonal transformations to B, producing $C = Q'^*B$ without computing Q.

The Q-less QR factorization allows the solution of sparse least squares problems

$$\text{minimize } Ax - b$$

with two steps

$$[c,R] = qr(A,b)$$

$$x = R \backslash c$$

If A is sparse, but not square, MATLAB uses these steps for the linear equation solving backslash operator

$$x = A \backslash b$$

Or, you can do the factorization yourself and examine R for rank deficiency.

It is also possible to solve a sequence of least squares linear systems with different right-hand sides, b, that are not necessarily known when $R = qr(A)$ is computed. The approach solves the “semi-normal equations”

$$R'^*R^*x = A'^*b$$

with

$$x = R \backslash (R \backslash (A'^*b))$$

and then employs one step of iterative refinement to reduce roundoff error

$$r = b - A * x$$

$$e = R \backslash (R \backslash (A'^*r))$$

$$x = x + e$$

Incomplete Factorizations

The luinc and cholinc functions provide approximate, *incomplete* factorizations, which are useful as preconditioners for sparse iterative methods.

The luinc function produces two different kinds of incomplete LU factorizations, one involving a drop tolerance and one involving fill-in level. If A is a sparse matrix, and tol is a small tolerance, then

$$[L,U] = luinc(A,tol)$$

computes an approximate LU factorization where all elements less than tol times the norm of the relevant column are set to zero. Alternatively, $[L,U] = luinc(A,'0')$

computes an approximate LU factorization where the sparsity pattern of L+U is a permutation of

the sparsity pattern of A.

For example,

```
load west0479
A = west0479;
nnz(A)
nnz(lu(A))
nnz(luinc(A,1e-6))
nnz(luinc(A,'0'))
```

shows that A has 1887 nonzeros, its complete LU factorization has 16777 nonzeros, its incomplete LU factorization with a drop tolerance of $1e-6$ has 10311 nonzeros, and itslu('0') factorization has 1886 nonzeros.

The luinc function has a few other options. See the luinc reference page for details.

The cholinc function provides drop tolerance and level 0 fill-in Cholesky factorizations of symmetric, positive definite sparse matrices. See the cholinc reference page for more information.

Simultaneous Linear Equations

Systems of simultaneous linear equations can be solved by two different classes of methods:

- Direct methods. These are usually variants of Gaussian elimination and are often expressed as matrix factorizations such as LU or Cholesky factorization. The algorithms involve access to the individual matrix elements.
- Iterative methods. Only an approximate solution is produced after a finite number of steps. The coefficient matrix is involved only indirectly, through a matrix-vector product or as the result of an abstract linear operator.

Direct Methods

Direct methods are usually faster and more generally applicable, if there is enough storage available to carry them out. Iterative methods are usually applicable to restricted cases of equations and depend upon properties like diagonal dominance or the existence of an underlying differential operator. Direct methods are implemented in the core of MATLAB and are made as efficient as possible for general classes of matrices. Iterative methods are usually implemented in MATLAB M-files and may make use of the direct solution of subproblems or preconditioners. The usual way to access direct methods in MATLAB is not through the lu or chol functions, but rather with the matrix division operators / and \. If A is square, the result of X = A\B is the solution to the linear system A*X = B. If A is not square, then a least squares solution is computed.

If A is a square, full, or sparse matrix, then A\B has the same storage class as B. Its computation involves a choice among several algorithms:

- If A is triangular, perform a triangular solve for each column of B.
- If A is a permutation of a triangular matrix, permute it and perform a sparse triangular solve for each column of B.

- If A is symmetric or Hermitian and has positive real diagonal elements, find a symmetric minimum degree order $p = \text{symmmd}(A)$, and attempt to compute the Cholesky factorization of $A(p,p)$. If successful, finish with two sparse triangular solves for each column of B.
- Otherwise (if A is not triangular, or is not Hermitian with positive diagonal, or if Cholesky factorization fails), find a column minimum degree order $p = \text{colmmd}(A)$. Compute the LU factorization with partial pivoting of $A(:,p)$, and perform two triangular solves for each column of B.

For a square matrix, MATLAB tries these possibilities in order of increasing cost. The tests for triangularity and symmetry are relatively fast and, if successful, allow for faster computation and more efficient memory usage than the general purpose method.

For example, consider the sequence below.

```
[L,U] = lu(A);
y = L\b;
x = U\y;
```

In this case, MATLAB uses triangular solves for both matrix divisions, since L is a permutation of a triangular matrix and U is triangular.

Using a Different Preordering. If A is not triangular or a permutation of a triangular matrix, backslash (\) uses colmmd and symmmd to determine a minimum degree order. Use the functions spparms to turn off the minimum degree preordering if you want to use a better preorder for a particular matrix.

If A is sparse and x = A\b can use LU factorization, you can use a column ordering other than colmmd to solve for x, as in the following example.

```
spparms('autommd',0);
q = colamd(A);
x = A(:,q)\b;
x(q) = x;
spparms('autommd',1);
```

If A can be factorized using Cholesky factorization, then x = A\b can be computed efficiently using

```
spparms('autommd',0);
p = symamd(A);
x = A(p,p)\b(p);
x(p) = x;
spparms('autommd',1);
```

In the examples above, the spparms('autommd',0) statement turns the automatic colmmd or symmmd ordering off. The spparms('autommd',1) statement turns it back on, just in case you use A\b later without specifying an appropriate pre-ordering. spparms with no arguments reports

the current settings of the sparse parameters.

Iterative Methods

Nine functions are available that implement iterative methods for sparse systems of simultaneous linear systems.

Functions for Iterative Methods for Sparse Systems Function Method

bicg Biconjugate gradient

bicgstab Biconjugate gradient stabilized

cgs Conjugate gradient squared

gmres Generalized minimum residual

lsqr

minres

pcg qmr

symmlq

LSQR implementation of Conjugate Gradients on the Normal Equations

Minimum residual

Preconditioned conjugate gradient

Quasiminimal residual

Symmetric LQ

These methods are designed to solve $Ax = b$ or $\min \|b - Ax\|$. For the Preconditioned Conjugate Gradient method, pcg, A must be a symmetric, positive definite matrix. minres and symmlq can be used on symmetric indefinite matrices. For lsqr, the matrix need not be square. The other five can handle nonsymmetric, square matrices.

All nine methods can make use of preconditioners. The linear system $Ax = b$ is replaced by the equivalent system

$$M^{-1}Ax = M^{-1}b$$

The preconditioner M is chosen to accelerate convergence of the iterative method. In many cases, the preconditioners occur naturally in the mathematical model. A partial differential equation with variable coefficients may be approximated by one with constant coefficients, for example. Incomplete matrix factorizations may be used in the absence of natural preconditioners.

The five-point finite difference approximation to Laplace's equation on a square, two-dimensional domain provides an example. The following statements use the preconditioned conjugate gradient method preconditioner $M = R'*R$, where R is the incomplete Cholesky factor of A .

```
A = delsq(numgrid('S',50));
b = ones(size(A,1),1);
tol = 1.e-3;
maxit = 10;
R = cholinc(A,tol);
[x,flag,err,iter,res] = pcg(A,b,tol,maxit,R',R);
```

Only four iterations are required to achieve the prescribed accuracy. Background information on these iterative methods and incomplete factorizations is available in [2] and [7].

Eigenvalues and Singular Values

Two functions are available which compute a few specified eigenvalues or singular values.`svds` is based on `oneigs` which uses ARPACK [6].

Functions to Compute a Few Eigenvalues or Singular Values Function Description

eigs	Few eigenvalues
svds	Few singular values

These functions are most frequently used with sparse matrices, but they can be used with full matrices or even with linear operators defined by M-files.

The statement

```
[V,lambda] = eigs(A,k,sigma)
```

finds the k eigenvalues and corresponding eigenvectors of the matrix `A` which are nearest the “shift” `sigma`. If `sigma` is omitted, the eigenvalues largest in magnitude are found. If `sigma` is zero, the eigenvalues smallest in magnitude are found. A second matrix, `B`, may be included for the generalized eigenvalue problem

$$Av = \pm B v$$

The statement

```
[U,S,V] = svds(A,k)
```

finds the k largest singular values of `A` and

```
[U,S,V] = svds(A,k,0)
```

finds the k smallest singular values.

For example, the statements

```
L = numgrid('L',65);
```

```
A = delsq(L);
```

set up the five-point Laplacian difference operator on a 65-by-65 grid in an L-shaped, two-dimensional domain. The statements

```
size(A)
```

```
nnz(A)
```

show that `A` is a matrix of order 2945 with 14,473 nonzero elements. The statement

```
[v,d] = eigs(A,1,0);
```

computes the smallest eigenvalue and eigenvector. Finally,

```
L(L>0) = full(v(L(L>0)));
```

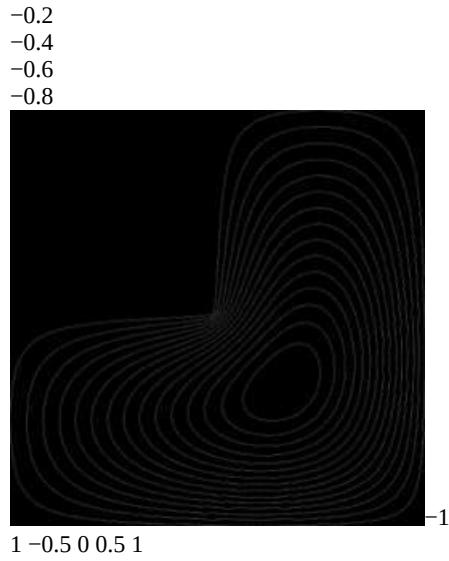
```
x = -1:1/32:1;
```

```
contour(x,x,L,15)
```

```
axis square
```

distributes the components of the eigenvector over the appropriate grid points and produces a contour plot of the result.

```
1  
0.8  
0.6  
0.4  
0.2  
0
```



The numerical techniques used in `ineigs` and `svds` are described in [6].

Selected Bibliography

[1] Amestoy, P. R., T. A. Davis, and I. S. Duff, "An Approximate Minimum Degree Ordering Algorithm," *SIAM Journal on Matrix Analysis and Applications*, Vol. 17, No. 4, Oct. 1996, pp. 886-905.

[2] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[3] Davis, T.A., Gilbert, J. R., Larimore, S.I., Ng, E., Peyton, B., "A Column Approximate Minimum Degree Ordering Algorithm," *Proc. SIAM Conference on Applied Linear Algebra*, Oct. 1997, p. 29.

[4] Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM J. Matrix Anal. Appl.*, Vol. 13, No. 1, January 1992, pp. 333-356.

[5] Larimore, S. I., *An Approximate Minimum Degree Column Ordering Algorithm*, MS Thesis, Dept. of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 1998, available at <http://www.cise.ufl.edu/tech-reports/>

[6] Lehoucq, R. B., D. C. Sorensen, C. Yang, *ARPACKUsers' Guide*, SIAM, Philadelphia, 1998.
 [7] Saad, Yousef, *Iterative Methods for Sparse Linear Equations*. PWS Publishing Company, 1996.

Programming and Data Types

MATLAB is a high-level language that includes data structures, functions, control flow statements, input/output, and object-oriented capabilities. This section presents MATLAB's programming features and techniques in the following chapters:

- M-File Programming – describes language constructs and how to create MATLAB programs

(M-files). It covers data types, flow control, array indexing, optimizing performance, and other topics.

- Character Arrays – covers MATLAB's support for string data, including how to create character arrays and cell arrays of strings, ways to represent strings, how to perform common string operations, and conversion between string and numeric formats.
- Multidimensional Arrays – discusses the use of MATLAB arrays having more than two dimensions. It covers array creation, array indexing, data organization, and how MATLAB functions operate on these arrays.
- Structures and Cell Arrays – describes two MATLAB data types that provide hierarchical storage for dissimilar kinds of data. Structures contain different kinds of data organized by named fields. Cell arrays consist of cells that themselves contain MATLAB arrays.
- Function Handles - describes how you use the MATLAB function handle to capture certain information about a function, including function references, that you can then use to execute the function from anywhere in the MATLAB environment.
- MATLAB Classes and Objects – presents MATLAB's object-oriented programming capabilities. Classes and objects enable you to add new data types and new operations to MATLAB. This section also includes examples of how to implement well-behaved classes in MATLAB.

Related Information

The following sections provide information that is also useful to MATLAB programmers. This documentation is available in MATLAB's online help.

- Graphics – describes how to plot vector and matrix data in 2-D and 3-D representations, how to annotate, print, and export plots, and how to use graphics objects and their figure and axes properties.
- Calling C and Fortran Programs from MATLAB - describes how to build C and Fortran subroutines into callable MEX files.
- Calling MATLAB from C and Fortran Programs - discusses how to use the MATLAB engine library to call MATLAB from C and Fortran programs.
- Calling Java from MATLAB - describes how to use the MATLAB interface to Java classes and objects.
- Importing and Exporting Data - describes techniques for importing data to and exporting data from the MATLAB environment.
- ActiveX and DDE Support - describes how to use ActiveX and Dynamic Data Exchange (DDE) with MATLAB.
- Serial Port I/O - describes how to communicate with peripheral devices such as modems, printers, and scientific instruments that you connect to your computer's serial port.

17

M-File Programming

MATLAB Programming: A Quick Start	17-3
Scripts	17-7
Functions	17-8
Local and Global Variables	17-19

Data Types	17-22
Key words	17-25
Operators	17-26
Flow Control	17-35
Subfunctions	17-43
Private Functions	17-45
Subscripting and Indexing	17-46
String Evaluation	17-54
Command/Function Duality	17-56
Empty Matrices	17-57
Errors and Warnings	17-59
Dates and Times	17-62
Obtaining User Input	17-69
Shell Escape Functions	17-70
Optimizing MATLAB Code	17-71

MATLAB provides a full programming language that enables you to write a series of MATLAB statements into a file and then execute them with a single command. You write your program in an ordinary text file, giving the file a name `filename.m`. The term you use for `filename` becomes the new command that MATLAB associates with the program. The file extension of `.m` makes this a MATLAB M-file.

This chapter explains the basics of how to write script and function programs in M-files. It covers the following topics:

- “MATLAB Programming: A Quick Start”
- “Scripts”
- “Functions”
- “Local and Global Variables”
- “Data Types”
- “Operators”
- “Flow Control”
- “Subfunctions”
- “Private Functions”
- “Subscripting and Indexing”
- “String Evaluation”
- “Command/Function Duality”
- “Empty Matrices”
- “Errors and Warnings”
- “Dates and Times”
- “Obtaining User Input”
- “Shell Escape Functions”
- “Optimizing MATLAB Code”

MATLAB Programming: A Quick Start

M-files can be *scripts* that simply execute a series of MATLAB statements, or they can be

functions that also accept arguments and produce output. You create M-files using a text editor, then use them as you would any other MATLAB function or command.

The process looks like this:

1 Create an M-file using a text editor.

```
function c = myfile(a,b) c = sqrt((a.^2)+(b.^2))
```

2 Call the M-file from the command line, or from within another M-file.

```
a = 7.5  
b = 3.342  
c = myfile(a,b)  
  
c =  
8.2109
```

Kinds of M-Files

There are two kinds of M-files.

Script M-Files Function M-Files

- Do not accept input arguments or return output arguments
- Can accept input arguments and return output arguments
- Operate on data in the workspace
- Useful for automating a series of steps you need to perform many times
- Internal variables are local to the function by default
- Useful for extending the MATLAB language for your application

What's in an M-File?

This section shows you the basic parts of a function M-file, so you can familiarize yourself with MATLAB programming and get started with some examples.

```
function f = fact(n) % Function definition line  
% FACT Factorial. % H1 line  
% FACT(N) returns the factorial of N, H! % Help text % usually denoted by N!  
% Put simply, FACT(N) is PROD(1:N).
```

f = prod(1:n); % Function body This function has some elements that are common to all MATLAB functions:

A *function definition line*. This line defines the function name, and the • number and order of input and output arguments.

• An *H1 line*. H1 stands for “help 1” line. MATLAB displays the H1 line for a function when you use lookfor or request help on an entire directory.

- *Help text.* MATLAB displays the help text entry together with the H1 line when you request help on a specific function.
- The *function body.* This part of the function contains code that performs the actual computations and assigns values to any output arguments.

Refer to “Functions” on page 17-8 for more detail on the parts of a MATLAB function.

Providing Help for Your Programs

You can provide user information for the programs you write by including a help text section at the beginning of your M-file. This section starts on the line following the function definition and ends at the first blank line. Each line of the help text must begin with a comment (%) character. MATLAB displays this information whenever you type

```
help m-file_name
```

You can also make help entries for an entire directory by creating a file with the special name Contents.m that resides in the directory. This file must contain only comment lines; that is, every line must begin with a percent sign. MATLAB displays the lines in aContents.m file whenever you type

```
help directory_name
```

If a directory does not contain aContents.m file, typing help directory_name displays the first help line (the H1 line) for each M-file in the directory.

Creating M-Files: Accessing Text Editors

M-files are ordinary text files that you create using a text editor. MATLAB provides a built-in editor, although you can use any text editor you like.

Note To open the editor on the PC, from the **File** menu, choose **New**, and then **M-File**.

Another way to edit an M-file is from the MATLAB command line using the edit function. For example,

```
edit foo
```

opens the editor on the filefoo.m. Omitting a filename opens the editor on an untitled file.

You can create the fact function shown on the previous page by opening your text editor, entering the lines shown, and saving the text in a file calledfact.m in your current directory.

Once you’ve created this file, here are some things you can do:

- List the names of the files in your current directory

```
what
```

- List the contents of M-filefact.m

```
type fact
```

- Call thefact function

```
fact(5)
```

```
ans =
```

```
120
```

Note Save any M-files you create and any MATLAB-supplied M-files that you edit in a directory that is not in the MATLAB directory tree. If you keep your files in the MATLAB directory tree, they may be overwritten when you install a new version of MATLAB. Also note

that the locations of files in the MATLAB/ toolbox directory tree are loaded and cached in memory at the beginning of each MATLAB session to improve performance. If you do save a new or edited file in the MATLAB/toolbox directory tree, restart MATLAB or use the `rehash` function to reload the directory and update the cache before you use the file. Scripts

Scripts

Scripts are the simplest kind of M-file because they have no input or output arguments. They're useful for automating series of MATLAB commands, such as computations that you have to perform repeatedly from the command line. Scripts operate on existing data in the workspace, or they can create new data on which to operate. Any variables that scripts create remain in the workspace after the script finishes so you can use them for further computations.

Simple Script Example

These statements calculate rho for several trigonometric functions of theta, then create a series of polar plots.

```
% An M-file script to produce % "flower petal" plots
theta = -pi:0.01:pi;
rho(1,:) = 2*sin(5*theta).^2; rho(2,:) = cos(10*theta).^3; rho(3,:) = sin(theta).^2; rho(4,:) =
5*cos(3.5*theta).^3; for k = 1:4
polar(theta,rho(k,:)) pause
end
% Comment lines
% Computations
% Graphics output
```

Try entering these commands in an M-file called `petals.m`. This file is now a MATLAB script. Typing `petals` at the MATLAB command line executes the statements in the script.

After the script displays a plot, press **Return** to move to the next plot. There are no input or output arguments; `petals` creates the variables it needs in the MATLAB workspace. When execution completes, the variables (`i`, `theta`, and `rho`) remain in the workspace. To see a listing of them, enter `whos` at the command prompt.

Functions

Functions are M-files that accept input arguments and return output arguments. They operate on variables within their own workspace. This is separate from the workspace you access at the MATLAB command prompt.

This section covers the following topics regarding functions:

- “Simple Script Example”
- “Basic Parts of a Function M-File”
- “Function Names”
- “How Functions Work”

- “Checking the Number of Function Arguments”
- “Passing Variable Numbers of Arguments”

Simple Function Example

The average function is a simple M-file that calculates the average of the elements in a vector.

```
function y = average(x)
% AVERAGE Mean of vector elements.
% AVERAGE(X), where X is a vector, is the mean of vector elements. % Non-vector input
results in an error.
[m,n] = size(x);
if (~((m == 1) | (n == 1)) | (m == 1 & n == 1))

error('Input must be a vector')
end
y = sum(x)/length(x); % Actual computation
```

If you would like, try entering these commands in an M-file called average.m. The average function accepts a single input argument and returns a single output argument. To call the average function, enter

```
z = 1:99;
average(z)
ans = 50
```

Basic Parts of a Function M-File

A function M-file consists of:

- “The Function Definition Line”
- “The H1 Line”
- “Help Text”
- “The Function Body”
- “Comments”

The Function Definition Line

The function definition line informs MATLAB that the M-file contains a function, and specifies the argument calling sequence of the function. The function definition line for the average function is

```
function y = average(x)
```

```
■
```

```
inputargument
functionname
outputargument
keyword
```

All MATLAB functions have a function definition line that follows this pattern.

If the function has multiple output values, enclose the output argument list in square brackets. Input arguments, if present, are enclosed in parentheses. Use commas to separate multiple input or output arguments. Here's a more complicated example.

```
function [x,y,z] = sphere(theta,phi,rho)
```

If there is no output, leave the output blank

```
function printresults(x)  
or use empty square brackets  
function [] = printresults(x)
```

The variables that you pass to the function do not need to have the same name as those in the function definition line.

The H1 Line

The H1 line, so named because it is the first help text line, is a comment line immediately following the function definition line. Because it consists of comment text, the H1 line begins with a percent sign, "%." For the average function, the H1 line is

```
% AVERAGE Mean of vector elements.
```

This is the first line of text that appears when a user types *help function_name* at the MATLAB prompt. Further, the *lookfor* function searches on and displays only the H1 line. Because this line provides important summary information about the M-file, it is important to make it as descriptive as possible.

Help Text

You can create online help for your M-files by entering text on one or more comment lines, beginning with the line immediately following the H1 line. The help text for the average function is

```
% AVERAGE(X), where X is a vector, is the mean of vector elements. % Nonvector input  
results in an error.
```

When you type *help function_name*, MATLAB displays the comment lines that appear between the function definition line and the first non-comment (executable or blank) line. The help system ignores any comment lines that appear after this help block.

For example, typing *help sin* results in
SIN Sine.

SIN(X) is the sine of the elements of X.

The Function Body

The function body contains all the MATLAB code that performs computations and assigns values to output arguments. The statements in the function body can consist of function calls, programming constructs like flow control and interactive input/output, calculations, assignments, comments, and blank lines.

For example, the body of the average function contains a number of simple programming statements.

```
[m,n] = size(x);
if (~((m == 1) | (n == 1)) | (m == 1 & n == 1)) % Flow control error('Input must be a vector') %
Error message display end
y = sum(x)/length(x); % Computation and assignment
```

Comments

As mentioned earlier, comment lines begin with a percent sign (%). Comment lines can appear anywhere in an M-file, and you can append comments to the end of a line of code. For example,

```
% Add up all the vector elements.
y = sum(x) % Use the sum function.
```

The first comment line immediately following the function definition line is considered the H1 line for the function. The H1 line and any comment lines immediately following it constitute the online help entry for the file.

In addition to comment lines, you can insert blank lines anywhere in an M-file. Blank lines are ignored. However, a blank line can indicate the end of the help text entry for an M-file.

Function Names

MATLAB function names have the same constraints as variable names. MATLAB uses the first 31 characters of names. Function names must begin with a letter; the remaining characters can be any combination of letters, numbers, and underscores. Some operating systems may restrict function names to shorter lengths.

The name of the text file that contains a MATLAB function consists of the function name with the extension.m appended. For example,
average.m

If the filename and the function definition line name are different, the internal name is ignored. Thus, while the function name specified on the function definition line does not have to be the same as the filename, we strongly recommend that you use the same name for both.

How Functions Work

You can call function M-files from either the MATLAB command line or from within other M-files. Be sure to include all necessary arguments, enclosing input arguments in parentheses and output arguments in square brackets.

This section provides the following information on calling MATLAB functions:

- “Function Name Resolution”
- “What Happens When You Call a Function”
- “Creating P-Code Files”

- “How MATLAB Passes Function Arguments”
- “Function Workspaces”

Function Name Resolution

When MATLAB comes upon a new name, it resolves it into a specific function by following these steps:

- 1 Checks to see if the name is a variable.
- 2 Checks to see if the name is a *subfunction* , a MATLAB function that resides in the same M-file as the calling function. Subfunctions are discussed in the section, “Subfunctions” on page 17-43.
- 3 Checks to see if the name is a *private function* , a MATLAB function that resides in a *private directory* , a directory accessible only to M-files in the directory immediately above it. Private directories are discussed in the section, “Private Functions” on page 17-45.
- 4 Checks to see if the name is a function on the MATLAB search path. MATLAB uses the first file it encounters with the specified name.

If you duplicate function names, MATLAB executes the one found first using the above rules. It is also possible to overload function names. This uses additional dispatching rules and is discussed in the section, “How MATLAB Determines Which Method to Call” on page 22-67.

What Happens When You Call a Function

When you call a function M-file from either the command line or from within another M-file, MATLAB parses the function into pseudocode and stores it in memory. This prevents MATLAB from having to reparse a function each time you call it during a session. The pseudocode remains in memory until you clear it using theclear function, or until you quit MATLAB.

You can useclear in any of the following ways to remove functions from the MATLAB workspace.

Syntax Description

clear function_name	Remove specified function from workspace	clear functions	Remove all compiled M-functions
clear all	Remove all variables and functions		

Creating P-Code Files You can save a preparsed version of a function or script, called P-code files, for later MATLAB sessions using thepcode function. For example,

PCODE average

parses average.m and saves the resulting pseudocode to the file named average.p. This saves MATLAB from reparsingaverage.m the first time you call it in each session.

MATLAB is very fast at parsing so thepcode function rarely makes much of a speed difference.

One situation where pcode does provide a speed benefit is for large GUI applications. In this case, many M-files must be parsed before the application becomes visible.

Another situation for code is when, for proprietary reasons, you want to hide algorithms you've created in your M-file.

How MATLAB Passes Function Arguments

From the programmer's perspective, MATLAB appears to pass all function arguments by value. Actually, however, MATLAB passes by value only those arguments that a function modifies. If a function does not alter an argument but simply uses it in a computation, MATLAB passes the argument by reference to optimize memory use.

Function Workspaces

Each M-file function has an area of memory, separate from MATLAB's base workspace, in which it operates. This area is called the function workspace, with each function having its own workspace context.

While using MATLAB, the only variables you can access are those in the calling context, be it the base workspace or that of another function. The variables that you pass to a function must be in the calling context, and the function returns its output arguments to the calling workspace context. You can however, define variables as global variables explicitly, allowing more than one workspace context to access them.

Checking the Number of Function Arguments

The nargin and nargout functions let you determine how many input and output arguments a function is called with. You can then use conditional statements to perform different tasks depending on the number of arguments. For example,

```
function c = testarg1(a,b)
```

```
if (nargin == 1)
```

```
c = a.^2;
```

```
elseif (nargin == 2)
```

```
c = a + b;
```

```
end
```

Given a single input argument, this function squares the input value. Given two inputs, it adds them together.

Here's a more advanced example that finds the first token in a character string. A *token* is a set of characters delimited by whitespace or some other character. Given one input, the function assumes a default delimiter of whitespace; given two, it lets you specify another delimiter if desired. It also allows for two possible output argument lists.

```
function [token,remainder] = strtok(string,delimiters) % Function requires at least one input argument if nargin < 1
```

```
error('Not enough input arguments.');
```

```
end
```

```
token = []; remainder = [];
```

```

len = length(string);
if len == 0

return
end
% If one input, use white space delimiter
if (nargin == 1)
delimiters = [9:13 32]; % White space characters end
i = 1;

% Determine where non-delimiter characters begin while (any(string(i) == delimiters))
i = i + 1;
if (i > len), return, end
end

% Find where token ends
start = i;
while (~any(string(i) == delimiters))

i = i + 1;
if (i > len), break, end
end
finish = i - 1;
token = string(start:finish);

% For two output arguments, count characters after % first delimiter (remainder)
if (nargout == 2)

remainder = string(finish + 1:end);
end

```

The strtok function is a MATLAB M-file in the strfun directory. **Note** The order in which output arguments appear in the function declaration line is important. The argument that the function returns in most cases appears first in the list. Additional, optional arguments are appended to the list.

Passing Variable Numbers of Arguments

The varargin and varargout functions let you pass any number of inputs or return any number of outputs to a function. This section describes how to use these functions and also covers:

- “Unpacking varargin Contents”
- “Packing varargout Contents”
- “varargin and varargout in Argument Lists”

MATLAB packs all specified input arguments into a *cell array*, a special kind of MATLAB array that consists of cells instead of array elements. Each cell can hold any size or kind of data – one might hold a vector of numeric data, another in the same array might hold an array of string

data, and so on. For output arguments, your function code must pack them into a cell array so that MATLAB can return the arguments to the caller.

Here's an example function that accepts any number of two-element vectors and draws a line to connect them.

```
function testvar(varargin)
for k = 1:length(varargin)
x(k) = varargin{k}(1); % Cell array indexing
y(k) = varargin{k}(2);
end
xmin = min(0,min(x));
ymin = min(0,min(y));
axis([xmin fix(max(x))+3 ymin fix(max(y))+3])
plot(x,y)
```

Coded this way, the `testvar` function works with various input lists; for example,

```
testvar([2 3],[1 5],[4 8],[6 5],[4 2],[2 3])
testvar([-1 0],[3 -5],[4 2],[1 1])
```

Unpacking varargin Contents

Because `varargin` contains all the input arguments in a cell array, it's necessary to use cell array indexing to extract the data. For example,

```
y(i) = varargin{i}(2);
Cell array indexing has two subscript components:
```

- The cell indexing expression, in curly braces
- The contents indexing expression(s), in parentheses

In the code above, the indexing expression `{i}` accesses the i^{th} cell of `varargin`. The expression `(2)` represents the second element of the cell contents.

Packing varargout Contents

When allowing any number of output arguments, you must pack all of the output into the `varargout` cell array. Use `nargout` to determine how many output arguments the function is called with. For example, this code accepts a two-column input array, where the first column represents a set of x coordinates and the second represents y coordinates. It breaks the array into separate $[x_i \ y_i]$ vectors that you can pass into the `testvar` function on the previous page.

```
function [varargout] = testvar2(arrayin)
for k = 1:nargout
varargout{k} = arrayin(k,:); % Cell array assignment end
```

The assignment statement inside the for loop uses cell array assignment syntax. The left side of the statement, the cell array, is indexed using curly braces to indicate that the data goes inside a cell. For complete information on cell array assignment, see the "Structures and Cell Arrays" section. Here's how to call `testvar2`.

```
a = {1 2;3 4;5 6;7 8;9 0}; [p1,p2,p3,p4,p5] = testvar2(a);  
varargin and varargout in Argument Lists
```

varargin or varargin must appear last in the argument list, following any required input or output variables. That is, the function call must specify the required arguments first. For example, these function declaration lines show the correct placement of varargin and varargout.

```
function [out1,out2] = example1(a,b,varargin)  
function [i,j,varargout] = example2(x1,y1,x2,y2,flag)
```

Local and Global Variables

The same guidelines that apply to MATLAB variables at the command line also apply to variables in M-files:

- You do not need to type or declare variables. Before assigning one variable to another, however, you must be sure that the variable on the right-hand side of the assignment has a value.
- Any operation that assigns a value to a variable creates the variable if needed, or overwrites its current value if it already exists.
- MATLAB variable names consist of a letter followed by any number of letters, digits, and underscores. MATLAB distinguishes between uppercase and lowercase characters, so A and a are not the same variable.
- MATLAB uses only the first 31 characters of variable names.

Ordinarily, each MATLAB function, defined by an M-file, has its own local variables, which are separate from those of other functions, and from those of the base workspace. However, if several functions, and possibly the base workspace, all declare a particular name as global, then they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the other functions declaring it global.

Suppose you want to study the effect of the interaction coefficients, α and β , in the Lotka-Volterra predator-prey model.

$$\dot{y}_1 = y_1 - \alpha y_1 y_2$$

$$\dot{y}_2 = -y_2 + \beta y_1 y_2$$

Create an M-file, lotka.m.

```
function yp = lotka(t,y)  
%LOTKA Lotka-Volterra predator-prey model. global ALPHA BETA  
yp = [y(1) - ALPHA*y(1)*y(2); -y(2) + BETA*y(1)*y(2)];
```

Then interactively enter the statements

```
global ALPHA BETA  
ALPHA = 0.01  
BETA = 0.02  
[t,y] = ode23('lotka',0,10,[1; 1]);  
plot(t,y)
```

The two global statements make the values assigned to ALPHA andBETA at the command prompt available inside the function defined bylotka.m. They can be modified interactively and new solutions obtained without editing any files.

For your MATLAB application to work with global variables:

- Declare the variable asglobal in every function that requires access to it. To enable the workspace to access the global variable, also declare it asglobal from the command line.
- In each function, issue the global command before the first occurrence of the variable name. The top of the M-file is recommended.

MATLAB global variable names are typically longer and more descriptive than local variable names, and sometimes consist of all uppercase characters. These are not requirements, but guidelines to increase the readability of MATLAB code and reduce the chance of accidentally redefining a global variable.

Persistent Variables

A variable may be defined as persistent so that it does not change value from one call to another. Persistent variables may be used within a function only. Persistent variables remain in memory until the M-file is cleared or changed.

persistent is exactly likeglobal, except that the variable name is not in the global workspace, and the value is reset if the M-file is changed or cleared. Three MATLAB functions support the use of persistent variables.

Function Description

mlock Prevents an M-file from being cleared
munlock Unlocks an M-file that had previously been locked
bymlock mislocked Indicates whether an M-file can be cleared or not

Special Values

Several functions return important special values that you can use in your M-files.

Function Return Value

ans
eps
realmax
realmin
pi
i,j
inf
NaN
computer
version

Most recent answer (variable). If you do not assign an output variable to an expression, MATLAB automatically stores the result inans.

Floating-point relative accuracy. This is the tolerance MATLAB uses in its calculations.

Largest floating-point number your computer can represent. Smallest floating-point number your

computer can represent. 3.1415926535897...
Imaginary unit.

Infinity. Calculations like $n/0$, where n is any nonzero real value, result in ∞ .

Not-a-Number, an invalid numeric value. Expressions like $0/0$ and ∞/∞ result in NaN , as do arithmetic operations involving NaN . $n/0$, where n is complex, also returns NaN .

Computer type.

MATLAB version string.

Here are several examples that use these values in MATLAB expressions.

```
x = 2*pi;  
A = [3+2i 7-8i];  
tol = 3*eps;
```

Data Types

There are 14 fundamental data types (or classes) in MATLAB. Each of these data types is in the form of an array. This array is a minimum of 0-by-0 in size and can grow to an n-dimensional array of any size. Two-dimensional versions of these arrays are called *matrices*.

All of the fundamental data types are shown in lowercase text in the diagram below. An additional, user-defined data type, shown below as *user class*, is a subset of the *structure* type.

ARRAY

char NUMERIC cell structure function handle
user class java class

int8 , uint8, single int16,uint16,
int32,uint32
double

sparse

The char data type holds characters in Unicode representation. A character string is merely a 1-by-n array of characters. You can use `char` to hold an array of strings as long as each string in the array has the same length. (This is because MATLAB arrays must be rectangular.) To hold an array of strings of unequal length, use `acell` array.

Numeric data types include signed and unsigned integers, single- and doubleprecision floating point, and sparse matrices of double-precision. The following hold true for numeric data types in MATLAB:

- All MATLAB computations are done in double-precision.
- Integer and single precision arrays offer more memory efficient storage than double-precision.
- All data types support basic array operations, such as subscripting and reshaping.

Data Types

- To perform mathematical operations on integer or single precision arrays, you must convert them to double precision using the `double` function.

A cell array provides a storage mechanism for dissimilar kinds of data. You can store arrays of different types and/or sizes within the cells of a cell array. For example, you can store a 1-by-50char array, a 7-by-13double array, and a 1-by-1uint32 in cells of the same cell array. You access data in a cell array using the same matrix indexing used on other MATLAB matrices and arrays.

The MATLAB structure data type is similar to the cell array in that it also stores dissimilar kinds of data. But, in this case, it stores the data in named fields rather than in cells. This enables you to attach a name to the groups of data stored within the structure. You access data in a structure using these same field names.

MATLAB data types are implemented as classes. You can also create MATLAB classes of your own. These user-defined classes inherit from the MATLAB structure class and are shown in the previous diagram as a subset of structure.

MATLAB provides an interface to the Java programming language that enables you to create objects from Java classes and call Java methods on these objects. A Java class is a MATLAB data type. There are built-in and third-party classes that are already available through the MATLAB interface. You can also create your own Java class definitions and bring them into MATLAB.

A function handle holds information to be used in referencing a function. When you create a function handle, MATLAB captures all the information about the function that it needs to locate and execute, or *evaluate*, it later on. Typically, a function handle is passed in an argument list to other functions. It is then used in conjunction with `feval` to evaluate the function to which the handle belongs.

The following table describes the data types in more detail.

Data Type Example Description `single` 3×10^{38} Single-precision numeric array. Single precision

requires less storage than double precision, but has less precision and a smaller range. This data type cannot be used in mathematical operations.

Data Type Example Description

`double` 3×10^{300} $5 + 6i$

`sparse speye(5)`

`int8, uint8, int16, uint16, int32, uint32` `uint8(magic(3))`

```
char 'Hello'  
cell {17 'hello' eye(2)}  
structure a.  
a.day = 12;  
a.color = 'Red'; a.mat = magic(3);
```

```
user class inline('sin(x)')
java class java.awt.Frame
function handle @humps
```

Double-precision numeric array. This is the most common MATLAB variable type.

Sparse double-precision matrix (2-D only). The sparse matrix stores matrices with only a few nonzero elements in a fraction of the space required for an equivalent full matrix. Sparse matrices invoke special methods especially tailored to solve sparse problems.

Signed and unsigned integer arrays that are 8, 16, and 32 bits in length. Enables you to manipulate integer quantities in a memory efficient manner. These data types cannot be used in mathematical operations.

Character array (each character is 16 bits long). This array is also referred to as a string.

Cell array. Elements of cell arrays contain other arrays. Cell arrays collect related data and information of a dissimilar size together.

Structure array. Structure arrays have field names. The fields contain other arrays. Like cell arrays, structures collect related data and information together.

MATLAB class. This user-defined class is created using MATLAB functions.

Java class. You can use classes already defined in the Java API or by a third party, or create your own classes in the Java language.

Handle to a MATLAB function. A function handle can be passed in an argument list and evaluated using feval.

Keywords

Key words

MATLAB reserves certain words for its own use as keywords of the language. To list the keywords, type

iskeyword

ans =

```
'break'
'case'
'catch'
'continue'
'else'
'elseif'
'end'
'for'
'function'
'global'
'if'
```

```
'otherwise'  
'persistent'  
'return'  
'switch'  
'try'  
'while'
```

See the online function reference pages for help to learn how to use any of these keywords. You should not use MATLAB keywords other than for their intended purpose. For example, a keyword should not be used as follows:

```
while = 5;  
??? while = 5;  
|
```

Error: Expected a variable, function, or constant, found "=".

Operators

MATLAB's operators fall into three categories:

- Arithmetic operators that perform numeric computations, for example, adding two numbers or raising the elements of an array to a given power.
- Relational operators that compare operands quantitatively, using operators like “less than” and “not equal to.”
- Logical operators that use the logical operators AND, OR, and NOT. This section also discusses operator precedence.

Arithmetic Operators

MATLAB provides these arithmetic operators

Operator Description + Addition

- Subtraction .
– Multiplication ./ Right division .\ Left division + Unary plus
- Unary minus : Colon operator .
– Power .' Transpose ' Complex conjugate transpose * Matrix multiplication / Matrix right division

Operator Description \ Matrix left division ^ Matrix power

Arithmetic Operators and Arrays

Except for some matrix operators, MATLAB's arithmetic operators work on corresponding elements of arrays with equal dimensions. For vectors and rectangular arrays, both operands must be the same size unless one is a scalar.

If one operand is a scalar and the other is not, MATLAB applies the scalar to every element of the other operand – this property is known as *scalar expansion*.

This example uses scalar expansion to compute the product of a scalar operand and a matrix.

```
A = magic(3)  
A =
```

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

```
3 * A  
ans =
```

```
24 3 18  
9 15 21  
12 27 6
```

Relational Operators

MATLAB provides these relational operators.

Operator Description < Less than <= Less than or equal to > Greater than >= Greater than or equal to == Equal to ~= Not equal to

Relational Operators and Arrays

MATLAB's relational operators compare corresponding elements of arrays with equal dimensions. Relational operators always operate element-by-element. In this example, the resulting matrix shows where an element of A is equal to the corresponding element of B.

```
A = [2 7 6;9 0 5;3 0.5 6];  
B = [8 7 0;3 2 5;4 -1 7];  
A == B
```

```
ans =  
  
0 1 0  
0 0 1  
0 0 0
```

For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. For the case where one operand is a scalar and the other is not, MATLAB tests the scalar against every element of the other operand. Locations where the specified relation is true receive the value 1. Locations where the relation is false receive the value 0.

Relational Operators and Empty Arrays

The relational operators work with arrays for which any dimension has size zero, as long as both arrays are the same size or one is a scalar. However, expressions such as

```
A == []  
return an error if A is not 0-by-0 or 1-by-1.  
To test for empty arrays, use the function
```

```
isempty(A)
```

Logical Operators

This section describes the MATLAB logical operators and also covers:

- “Using Logical Operators on Arrays”
- “Logical Functions”

- “Logical Expressions Using the find Function”

MATLAB provides these logical operators.

Operator Description

& AND

| OR

~ NOT

Note In addition to these logical operators, theops directory contains a number of functions that perform bitwise logical operations. See online help for more information.

Each logical operator has a specific set of rules that determines the result of a logical expression:

- An expression using the AND operator, &, is true if both operands are logically true. In numeric terms, the expression is true if both operands are nonzero. This example shows the logical AND of the elements in the vector u with the corresponding elements in the vector v.

```
u = [1 0 2 3 0 5];
v = [5 6 1 0 0 7];
u & v
ans =
1 0 1 0 0 1
```

- An expression using the OR operator, |, is true if one operand is logically true, or if both operands are logically true. An OR expression is false only if both operands are false. In numeric terms, the expression is false only if both operands are zero. This example shows the logical OR of the elements in the vector u and with the corresponding elements in the vector v.

```
u | v
ans =
1 1 1 1 0 1
```

- An expression using the NOT operator, ~, negates the operand. This produces a false result if the operand is true, and true if it is false. In numeric terms, any nonzero operand becomes zero, and any zero operand becomes one. This example shows the negation of the elements in the vector u.

```
~u
ans =
0 1 0 0 1 0
```

Using Logical Operators on Arrays

MATLAB’s logical operators compare corresponding elements of arrays with equal dimensions. For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. For the case where one operand is a scalar and the other is not, MATLAB tests the scalar against every element of the other operand. Locations where the specified relation is true receive the value 1. Locations where the relation is false receive the value 0.

Logical Functions

In addition to the logical operators, MATLAB provides a number of logical functions.

Function Description Examples xor Performs an exclusive OR on its arguments. xor returns true if one argument is true and the other false. xor(a,b) In numeric terms, the function returns 1 if one operand is nonzero and the other operand is zero. 0

all Returns 1 if all of the elements in a $A = [0 \ 1 \ 2; 3 \ 5 \ 0]$

vector are true or nonzero. all operates columnwise on matrices. $A =$
0 1 2
3 5 0

all(A)
ans = 0 1 0
any Returns 1 if any of the elements of v = [5 0 8];

argument are true or nonzero; any(v) otherwise, it returns 0. Like all, the any function operates columnwise on ans = matrices. A number of other MATLAB functions perform logical operations. For example, the isnan function returns 1 for NaNs; the isinf function returns 1 for Infs. See the ops directory for a complete listing of logical functions.

Logical Expressions Using the find Function

The find function determines the indices of array elements that meet a given logical condition. It's useful for creating masks and index matrices. In its most general form, find returns a single vector of indices. This vector can be used to index into arrays of any size or shape. For example,

```
A = magic(4)
A =
```

```
16 2 3 13
5 11 10 8
9 7 6 12
4 14 15 1
```

```
i = find(A > 8); A(i) = 100
A =
```

```
100 2 3 100
5 100 100 8
100 7 6 100
4 100 100 1
```

You can also use find to obtain both the row and column indices for a rectangular matrix, as well as the array values that meet the logical condition. Use the help facility for more information on find.

Operator Precedence

You can build expressions that use any combination of arithmetic, relational, and logical operators. Precedence levels determine the order in which MATLAB evaluates an expression. Within each precedence level, operators have equal precedence and are evaluated from left to right. The precedence rules for MATLAB operators are shown in this list, ordered from highest precedence level to lowest precedence level:

- 1** Parentheses ()
- 2** Transpose(.'), power(.[^]), complex conjugate transpose('), matrix power([^])
- 3** Unary plus (+), unary minus (-), logical negation (~)
- 4** Multiplication (*), right division (/), left division(\.), matrix multiplication (*), matrix right division (/), matrix left division ()
- 5** Addition (+), subtraction (-)
- 6** Colon operator (:)
- 7** Less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (==), not equal to (~=)
- 8** Logical AND (&)
- 9** Logical OR (|)

Overriding Default Precedence

The default precedence can be overridden using parentheses, as shown in this example.

```
A = [3 9 5];
B = [2 1 5];
C = A./B.^2

C = 0.7500 9.0000 0.2000
C = (A./B).^2
C = 2.2500 81.0000 1.0000
```

Expressions can also include values that you access through subscripts. b = sqrt(A(2)) + 2*B(1)
b = 7

Flow Control

There are eight flow control statements in MATLAB:

- if, together with else and elseif, executes a group of statements based on some logical condition.
- switch, together with case and otherwise, executes different groups of statements depending on the value of some logical condition.
- while executes a group of statements an indefinite number of times, based on some logical condition.
- for executes a group of statements a fixed number of times.
- continue passes control to the next iteration of a for or while loop, skipping any remaining statements in the body of the loop.
- break terminates execution of a for or while loop.
- try...catch changes flow control if an error is detected during execution.
- return causes execution to return to the invoking function.

All flow constructs used to indicate the end of the flow control block.

Note You can often speed up the execution of MATLAB code by replacing for and while loops with vectorized code. See “Optimizing MATLAB Code” on page 17-71.

if, else, and elseif

if evaluates a logical expression and executes a group of statements based on the value of the expression. In its simplest form, its syntax is

```
if logical_expression
  statements
end
```

If the logical expression is true (1), MATLAB executes all the statements between the if and end lines. It resumes execution at the line following the end statement. If the condition is false (0), MATLAB skips all the statements between the if and end lines, and resumes execution at the line following the end statement.

For example,

```
if rem(a,2) == 0
  disp('a is even')
  b = a/2;

end
```

You can nest any number of if statements.

If the logical expression evaluates to a nonscalar value, all the elements of the argument must be nonzero. For example, assume X is a matrix. Then the statement

```
if X
  statements
end
```

is equivalent to

```
if all(X(:)) statements
end
```

The else and elseif statements further conditionalize the if statement:

- The else statement has no logical condition. The statements associated with it execute if the preceding if (and possibly elseif condition) is false (0).
- The elseif statement has a logical condition that it evaluates if the preceding if (and possibly elseif condition) is false (0). The statements associated with it execute if its logical condition is true (1). You can have multiple elseifs within an if block.

```
if n < 0 % If n negative, display error message. disp('Input must be positive');
elseif rem(n,2) == 0 % If n positive and even, divide by 2. A = n/2;
else
  A = (n+1)/2; % If n positive and odd, increment and divide.
```

```
end
```

if Statements and Empty Arrays

An if condition that reduces to an empty array represents a false condition. That is,

```
if A
```

```
S1  
else  
S0  
end
```

will execute statement S0 when A is an empty array.

switch

switch executes certain statements based on the value of a variable or expression. Its basic form is

```
switch expression (scalar or string)  
case value1  
statements % Executes if expression is value1
```

```
case value2  
statements % Executes if expression is value2
```

```
.
```

```
.
```

```
.
```

```
otherwise
```

```
statements % Executes if expression does not % does not match any case  
end
```

This block consists of:

- The word switch followed by an expression to evaluate.
- Any number of case groups. These groups consist of the word case followed by a possible value for the expression, all on a single line. Subsequent lines contain the statements to execute for the given value of the expression. These can be any valid MATLAB statement including another switch block. Execution of a case group ends when MATLAB encounters the next case statement or the otherwise statement. Only the first matching case is executed.
- An optional otherwise group. This consists of the word otherwise, followed by the statements to execute if the expression's value is not handled by any of the preceding case groups. Execution of the otherwise group ends at the end statement.
- An end statement.

switch works by comparing the input expression to each case value. For numeric expressions, a case statement is true if(*value*==*expression*). For string expressions, a case statement is true ifstrcmp(*value*,*expression*).

The code below shows a simple example of the switch statement. It checks the variable

input_num for certain values. If input_num is -1, 0, or 1, the case statements display the value on screen as text. If input_num is none of these values, execution drops to the otherwise statement and the code displays the text 'other value'.

```
switch input_num
case -1
    disp('negative one');

case 0
    disp('zero');
case 1
    disp('positive one');
otherwise
    disp('other value');
end
```

Note For C Programmers unlike the C language switch construct, MATLAB's switch does not "fall through." That is, if the first case statement is true, other case statements do not execute. Therefore, break statements are not used.

switch can handle multiple conditions in a single case statement by enclosing the case expression in a cell array.

```
switch var
case 1
    disp('1')

case {2,3,4}
    disp('2 or 3 or 4')
case 5
    disp('5')
otherwise
    disp('something else') end
```

w hile

The while loop executes a statement or group of statements repeatedly as long as the controlling expression is true (1). Its syntax is

```
while expression
    statements
end
```

If the expression evaluates to a matrix, all its elements must be 1 for execution to continue. To reduce a matrix to a scalar value, use the all and any functions. For example, this while loop finds the first integer n for which n! (n factorial) is a 100-digit number.

```
n = 1;
while prod(1:n) < 1e100
    n = n + 1;
end
```

Exit a while loop at any time using the break statement.

while Statements and Empty Arrays

A while condition that reduces to an empty array represents a false condition. That is, while A, S1, end

never executes statement S1 when A is an empty array.

for

The for loop executes a statement or group of statements a predetermined number of times. Its syntax is:

```
for index = start:increment:end  
statements  
end
```

The default increment is 1. You can specify any increment, including a negative one. For positive indices, execution terminates when the value of the index exceeds the end value; for negative increments, it terminates when the index is less than the end value.

For example, this loop executes five times.

```
for i = 2:6  
x(i) = 2*x(i-1);  
end
```

You can nest multiple for loops.

```
for i = 1:m  
for j = 1:n  
A(i,j) = 1/(i + j - 1);  
end  
end
```

Note You can often speed up the execution of MATLAB code by replacing for and while loops with vectorized code. See “Vectorization of Loops” on page 1-91 for details.

Using Arrays as Indices

The index of a for loop can be an array. For example, consider an m-by-n array A. The statement

```
for i = A  
statements  
end
```

sets i equal to the vector A(:,k). For the first loop iteration, k is equal to 1; for the second, k is equal to 2, and so on until k equals n. That is, the loop iterates for a number of times equal to the number of columns in A. For each iteration, i is a vector containing one of the columns of A.

continue

The continue statement passes control to the next iteration of the for or while loop in which it appears, skipping any remaining statements in the body of the loop. In nested loops, continue

passes control to the next iteration of the for orwhile loop enclosing it.

The example below shows a continue loop that counts the lines of code in the file,magic.m, skipping all blank lines and comments. Acontinue statement is used to advance to the next line inmagic.m without incrementing the count whenever a blank line or comment line is encountered.

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)

line = fgetl(fid);
if isempty(line) | strncmp(line,'%',1)
continue

end
count = count + 1;
end
disp(sprintf('%d lines',count));
```

break

The break statement terminates the execution of afor loop orwhile loop. When abreak statement is encountered, execution continues with the next statement outside of the loop. In nested loops,break exits from the innermost loop only.

The example below shows a while loop that reads the contents of the filefft.m into a MATLAB character array. Abreak statement is used to exit thewhile loop when the first empty line is encountered. The resulting character array contains the M-file help for thefft program.

```
fid = fopen('fft.m','r'); s = "";
while ~feof(fid)

line = fgetl(fid); if isempty(line)
break

end
s = strvcat(s,line); end
disp(s)
```

try ... catch

The general form of atry ...catch statement sequence is

```
try,
statement,
...,
statement,
```

```
catch,  
statement,  
...,  
statement,  
end
```

In this sequence the statements between try and catch are executed until an error occurs. The statements between catch and end are then executed. Use lasterr to see the cause of the error. If an error occurs between catch and end, MATLAB terminates execution unless another try...catch sequence has been established.

return

return terminates the current sequence of commands and returns control to the invoking function or to the keyboard. return is also used to terminate keyboard mode. A called function normally transfers control to the function that invoked it when it reaches the end of the function. return may be inserted within the called function to force an early termination and to transfer control to the invoking function.

Subfunctions

Subfunctions

Function M-files can contain code for more than one function. The first function in the file is the *primary function*, the function invoked with the M-file name. Additional functions within the file are *subfunctions* that are only visible to the primary function or other subfunctions in the same file.

Each subfunction begins with its own function definition line. The functions immediately follow each other. The various subfunctions can occur in any order, as long as the primary function appears first.

```
function [avg,med] = newstats(u) % Primary function  
% NEWSTATS Find mean and median with internal functions. n = length(u);  
avg = mean(u,n); med = median(u,n);  
  
function a = mean(v,n) % Calculate average. a = sum(v)/n;  
% Subfunction  
  
function m = median(v,n) % Calculate median.  
w = sort(v);  
if rem(n,2) == 1  
  
m = w((n+1)/2);  
else  
m = (w(n/2)+w(n/2+1))/2; end  
% Subfunction
```

The subfunctions mean and median calculate the average and median of the input list. The primary function newstats determines the length of the list and calls the subfunctions, passing to them the list lengthn. Functions within the same M-file cannot access the same variables unless you declare them as global within the pertinent functions, or pass them as arguments. In addition, the help facility can only access the primary function in an M-file.

When you call a function from within an M-file, MATLAB first checks the file to see if the function is a subfunction. It then checks for a private function (described in the following section) with that name, and then for a standard M-file on your search path. Because it checks for a subfunction first, you can supersede existing M-files using subfunctions with the same name, for example, mean in the above code. Function names must be unique within an M-file, however. Private Functions

Private Functions

Private functions are functions that reside in subdirectories with the special name private. They are visible only to functions in the parent directory. For example, assume the directory newmath is on the MATLAB search path. A subdirectory of newmath called private can contain functions that only the functions in newmath can call. Because private functions are invisible outside of the parent directory, they can use the same names as functions in other directories. This is useful if you want to create your own version of a particular function while retaining the original in another directory. Because MATLAB looks for private functions before standard M-file functions, it will find a private function named test.m before a nonprivate M-file named test.m.

You can create your own private directories simply by creating subdirectories called private using the standard procedures for creating directories or folders on your computer. Do not place these private directories on your path.

Subscripting and Indexing

Subscripting

This section explains how to use subscripting to access and assign to elements of a MATLAB matrix. It covers the following:

- Accessing Single Elements of a Matrix
- Accessing Multiple Elements of a Matrix
- Expanding the Size of a Matrix
- Deleting Rows and Columns
- Concatenating Matrices

Accessing Single Elements of a Matrix

The element in row i and column j of A is denoted by A(i,j). For example, suppose A = magic(4), Then A(4,2) is the number in the fourth row and second column. For our magic square, A(4,2) is 14.

```
A = magic(4)  
A =
```

```
16 2 3 13  
5 11 10 8  
9 7 6 12  
4 14 15 1
```

```
A(4,2)  
ans =  
14
```

It is also possible to refer to the elements of a matrix with a single subscript, $A(k)$. This is the usual way of referencing row and column vectors. But it can also apply to a fully two-dimensional matrix, in which case the array is regarded as one long column vector formed from the columns of the original matrix.

So, for our magic square, $A(8)$ is another way of referring to the value 14 stored in $A(4,2)$.

```
A(8)  
ans =  
14
```

Accessing Multiple Elements of a Matrix

It is possible to compute the sum of the elements in the fourth column of A by typing $A(1,4) + A(2,4) + A(3,4) + A(4,4)$

You can reduce the size of this expression using the colon operator. Subscript expressions involving colons refer to portions of a matrix.

```
A(1:m,n)
```

refers to the elements in rows 1 through m of column n of the A matrix. Using this notation, you can compute the sum of the fourth column of A more succinctly.

```
sum(A(1:4,4))
```

The colon by itself refers to *all* the elements in a row or column of a matrix. The keyword `end` refers to the *last* row or column. Using the following syntax, you can compute this same column sum without having to specify row and column numbers.

```
sum(A(:,end))  
ans =  
34
```

By adding an additional colon operator, you can refer to nonconsecutive elements in a matrix. `Them:3:n` in this expression means to make the assignment to every third element in the matrix.

```
B = A;  
B(1:3:end) = -10  
B =  
  
-10 2 3 -10 5 11 -10 8 9 -10 6 12  
-10 14 15 -10
```

You can repeatedly access an array element using the ones function. To create a new 2-by-6 matrix out of the 9th element of A,

```
B = A(9*ones(2,6))
```

```
B =
```

```
3 3 3 3 3 3 3 3 3 3 3
```

Expanding the Size of a Matrix

If you try to access an element outside of the matrix, it is an error B = A(4,5)

Index exceeds matrix dimensions

However, if you store a value in an element outside of the matrix, the size of the matrix increases to accommodate the new element.

```
B = A;
```

```
B(4,5) = 17
```

```
B =
```

```
16 2 3 13 0
```

```
5 11 10 8 0
```

```
9 7 6 12 0
```

```
4 14 15 1 17
```

Similarly, you can expand a matrix by assigning to a series of matrix elements.

```
B(2:5,5:6) = 5
```

```
B =
```

```
16 2 3 13 0 0
```

```
5 11 10 8 5 5
```

```
9 7 6 12 5 5
```

```
4 14 15 1 5 5
```

```
0 0 0 0 5 5
```

Deleting Rows and Columns

You can delete rows and columns from a matrix using just a pair of square brackets. Start with

```
X = A;
```

Then, to delete the second column of X, use

```
X(:,2) = []
```

This changes X to

```
X =
```

```
16 3 13
```

```
5 10 8
```

```
9 6 12
```

```
4 15 1
```

If you delete a single element from a matrix, the result isn't a matrix anymore. So expressions like

```
X(1,2) = []
```

result in an error. However, using a single subscript deletes a single element, or sequence of

elements, and reshapes the remaining elements into a row vector. So

```
X(2:2:10) = []
```

results in X =

```
16 9 3 6 13 12 1
```

Concatenating Matrices

Concatenation is the process of joining small matrices together to make bigger ones. In fact, you made your first matrix by concatenating its individual elements. The pair of square brackets, [], is the concatenation operator. For an example, start with the 4-by-4 magic square,A, and form

```
B = [A A+32;A+48 A+16]
```

The result is an 8-by-8 matrix, obtained by joining the four submatrices.

```
B =
```

```
16 2 3 13 48 34 35 45  
5 11 10 8 37 43 42 40  
9 7 6 12 41 39 38 44  
4 14 15 1 36 46 47 33  
64 50 51 61 32 18 19 29  
53 59 58 56 21 27 26 24  
57 55 54 60 25 23 22 28  
52 62 63 49 20 30 31 17
```

This matrix is half way to being another magic square. Its elements are a rearrangement of the integers 1:64. Its column sums are the correct value for an 8-by-8 magic square.

```
sum(B)
```

```
ans =
```

```
260 260 260 260 260 260 260 260 But, its row sums,sum(B)', are not all the same. Further manipulation is necessary to make this a valid 8-by-8 magic square.
```

Advanced Indexing

MATLAB stores each array as a column of values regardless of the actual dimensions. This column consists of the array columns, appended end to end.

For example, MATLAB stores

```
A = [2 6 9; 4 2 8; 3 0 1]
```

as

```
2  
4  
3  
6  
2  
0  
9  
8  
1
```

Accessing A with a single subscript indexes directly into the storage column. A(3) accesses the third value in the column, the number 3.A(7) accesses the seventh value, 9, and so on.

If you supply more subscripts, MATLAB calculates an index into the storage column based on the dimensions you assigned to the array. For example, assume a two-dimensional array like A has size [d1 d2], where d1 is the number of rows in the array and d2 is the number of columns. If you supply two subscripts (i,j) representing row-column indices, the offset is

$$(j-1)*d1+i$$

Given the expression A(3,2), MATLAB calculates the offset into A's storage column as $(2-1)*3+3$, or 6. Counting down six elements in the column accesses the value 0.

Indexing Into Multidimensional Arrays

This storage and indexing scheme also extends to multidimensional arrays. In this case, MATLAB operates on a page-by-page basis to create the storage column, again appending elements columnwise.

For example, consider a 5-by-4-by-3-by-2 array C.

MATLAB displays C as MATLAB stores C as

page(1,1) = 1
2 1 4 3 5 52 1 7 9 0 5 6 3 2 30 1 5 9 43 2 7 5 1
6 1 page(2,1) = 2
3 6 2 4 2 77 1 4 9 30 0 1 5 59 4 4 2 71 8 2 5 5
9 2 page(3,1) = 9
5 2 2 8 3 62 5 1 8 75 1 5 2 0 0 9 0 9 99 4 5 3 1
2 1 page(1,2) = 0
4 9 8 2 3 80 0 3 3 46 4 9 6 41 9 2 3 10 2 8 7 4
2 2 page(2,2) = 9
5 7 0 1 3 22 4 8 1 57 5 8 6 26 8 8 4 29 4 1 2 5
0 9 page(3,2) = 2
5 1 6 6 5 12 9 1 3 97 1 1 1 48 0 1 5
3 2 7 6 .

Again, a single subscript indexes directly into this column. For example, C(4) produces the result

```
ans =  
0
```

If you specify two subscripts (i,j) indicating row-column indices, MATLAB calculates the offset as described above. Two subscripts always access the first page of a multidimensional array, provided they are within the range of the original array dimensions.

If more than one subscript is present, all subscripts must conform to the original array dimensions. For example, C(6,2) is invalid, because all pages of C have only five rows.

If you specify more than two subscripts, MATLAB extends its indexing scheme accordingly. For example, consider four subscripts(i,j,k,l) into a four-dimensional array with size[d1 d2 d3 d4]. MATLAB calculates the offset into the storage column by

$$(l-1)(d_3)(d_2)(d_1)+(k-1)(d_2)(d_1)+(j-1)(d_1)+i$$

For example, if you index the arrayC using subscripts (3,4,2,1), MATLAB returns the value 5 (index 38 in the storage column).

In general, the offset formula for an array with dimensions[d₁ d₂ d₃ ... d_n] using any subscripts(s₁ s₂ s₃ ... s_n) is

$$(s_n-1)(d_{n-1})(d_{n-2})...(d_1)+(s_{n-1}-1)(d_{n-2})...(d_1)+...+(s_2-1)(d_1)+s_1$$

Because of this scheme, you can index an array using any number of subscripts. You can append any number of 1s to the subscript list because these terms become zero. For example,

C(3,2,1,1,1,1,1)

is equivalent to

C(3,2)

String Evaluation

String evaluation adds power and flexibility to the MATLAB language, letting you perform operations like executing user-supplied strings and constructing executable strings through concatenation of strings stored in variables.

eval

The eval function evaluates a string that contains a MATLAB expression, statement, or function call. In its simplest form, the eval syntax is eval('string')

For example, this code uses eval on an expression to generate a Hilbert matrix of ordern.

```
t = '1/(i+j-1)';
for i = 1:n
for j = 1:n
a(i,j) = eval(t);
end
end
```

Here's an example that uses eval on a statement.

```
eval('t = clock');
```

Constructing Strings for Evaluation

You can concatenate strings to create a complete expression for input to eval.

This code shows how eval can create 10 variables named P1,P2, ...P10, and set each of them to a different value.

```
for i=1:10
eval(['P',int2str(i),'= i.^2'])
end
```

feval

The feval function differs from eval in that it executes a function rather than a MATLAB expression. The function to be executed is specified in the first argument by either a function handle or a string containing the function name. String Evaluation

You can use feval and the input function to choose one of several tasks defined by M-files. This example uses function handles for the sin, cos, and log functions.

```
fun = {@sin; @cos; @log};  
k = input('Choose function number: ');  
x = input('Enter value: ');  
feval(fun(k),x)
```

Note Use feval rather than eval whenever possible. M-files that use feval execute faster and can be compiled with the MATLAB Compiler.

Command / Function Duality

MATLAB commands are statements like

```
load  
help
```

Many commands accept modifiers that specify operands.

```
load August17.dat  
help magic  
type rank
```

An alternate method of supplying the command modifiers makes them string arguments of functions.

```
load('August17.dat')  
help('magic')  
type('rank')
```

This is MATLAB's *command / function duality*. Any command of the form command argument can also be written in the functional form
command('argument')

The advantage of the functional approach comes when the string argument is constructed from other pieces. The following example processes multiple data files, August1.dat, August2.dat, and so on. It uses the function int2str, which converts an integer to a character string, to help build the filename.

```
for d = 1:31  
s = ['August' int2str(d) '.dat']  
load(s)  
  
% Process the contents of the d-th file
```

Empty Matrices

Empty Matrices

A matrix having at least one dimension equal to zero is called an *empty matrix*. The simplest empty matrix is 0-by-0 in size. Examples of more complex matrices are those of dimension 0-by-5 or 10-by-0-by-20.

To create a 0-by-0 matrix, use the square bracket operators with no value specified.

```
A = [];
```

```
whos A
```

Name Size Bytes Class

A 0x0 0 double array You can create empty arrays of other sizes using the zeros, ones, rand, or eye functions. To create a 0-by-5 matrix, for example, use

```
E = zeros(0,5)
```

Operating on an Empty Matrix

The basic model for empty matrices is that any operation that is defined for m-by-n matrices, and that produces a result whose dimension is some function of m and n, should still be allowed when m or n is zero. The size of the result should be that same function, evaluated at zero.

For example, horizontal concatenation

```
C = [A B]
```

requires that A and B have the same number of rows. So if A is m-by-n and B is m-by-p, then C is m-by-(n+p). This is still true if m or n or p is zero. Many operations in MATLAB produce row vectors or column vectors. It is possible for the result to be the empty row vector

```
r = zeros(1,0)
```

or the empty column vector

```
C = zeros(0,1)
```

As with all matrices in MATLAB, you must follow the rules concerning compatible dimensions. In the following example, an attempt to add a 1-by-3 matrix to a 0-by-3 empty matrix results in an error.

```
[1 2 3] + ones(0,3)
```

??? Error using ==> +

Matrix dimensions must agree.

Some MATLAB functions, like sum and max, are *reductions*. For matrix arguments, these functions produce vector results; for vector arguments they produce scalar results. Empty inputs produce the following results with these functions:

- sum([]) is 0
- prod([]) is 1
- max([]) is []
- min([]) is []

Using Empty Matrices with If or While

When the expression part of an if or while statement reduces to an empty matrix, MATLAB evaluates the expression as being false. The following example executes statement S0, because A is an empty array.

```
A = ones(25,0,4);
if A
```

```
S1
else
S0
```

Errors and Warnings

In many cases, it's desirable to take specific actions when different kinds of errors occur. For example, you may want to prompt the user for more input, display extended error or warning information, or repeat a calculation using default values. MATLAB's error handling capabilities let your application check for particular error conditions and execute appropriate code depending on the situation.

Error Handling with eval and lasterr

The basic tools for error-handling in MATLAB are:

- The eval function, which lets you execute a function and specify a second function to execute if an error occurs in the first.
 - The lasterr function, which returns a string containing the last error generated by MATLAB.
- The eval function provides error-handling capabilities using the two-argument form eval ('trystring','catchstring')

If the operation specified by trystring executes properly, eval simply returns. If trystring generates an error, the function evaluates catchstring. Use catchstring to specify a function that determines the error generated by trystring and takes appropriate action.

The trystring/catchstring form of eval is especially useful in conjunction with the lasterr function. lasterr returns a string containing the last error message generated by MATLAB. Use lasterr inside the catchstring function to "catch" the error generated by trystring.

For example, this function uses lasterr to check for a specific error message that can occur during matrix multiplication. The error message indicates that matrix multiplication is impossible because the operands have different inner dimensions. If the message occurs, the code truncates one of the matrices to perform the multiplication.

```
function C = catchfcn(A,B)
l = lasterr;
j = findstr(l,'Inner matrix dimensions')
if (~isempty(j))
    [m,n] = size(A) [p,q] = size(B) if (n>p)
```

```

A(:,p+1:n) = [] elseif (n<p)
B(n+1:p,:) = [] end
C = A*B;
else
C = 0;
end

```

This example uses the two-argument form of eval with the catchfcn function shown above.

```

clear
A = [1 2 3; 6 7 2; 0 1 5];
B = [9 5 6; 0 4 9];
eval('A*B','catchfcn(A,B)')

A = 1:7;
B = randn(9,9);
eval('A*B','catchfcn(A,B)')

```

Displaying Error and Warning Messages

Use the error and fprintf functions to display error information on the screen. The error function has the syntax
`error('error string')`

If you call the error function from inside an M-file, error displays the text in the quoted string and causes the M-file to stop executing. For example, suppose the following appears inside the M-file myfile.m.

```

if n < 1
error('n must be 1 or greater.')

```

For n equal to 0, the following text appears on the screen and the M-file stops. ??? Error using
`==> myfile`
n must be 1 or greater.

In MATLAB, warnings are similar to error messages, except program execution does not stop. Use the warning function to display warning messages.

```
warning('warning string')
```

The function lastwarn displays the last warning message issued by MATLAB.

Dates and Times

MATLAB provides functions for time and date handling. These functions are in a directory called timefun in the MATLAB Toolbox.

Category	Function	Description
Current date and time	clock	Current date and time as date vector
Current date as date string	datestr	Convert to serial date number
Conversion	datenum	Convert to serial date number
Date components	datevec	Convert to string representation of date
Utility	calendar	Calendar
Calendar	datetick	Date formatted tick labels
End of month	eomday	Day of the week
Weekday	tic, toc	Timing
Day of the week	cputime	CPU time in seconds
Elapsed time	etime	Elapsed time
Stopwatch timer	tic, toc	Stopwatch timer

Date Formats

This section covers the following topics:

- “Types of Date Formats”
- “Conversions Between Date Formats”
- “Date String Formats”
- “Output Formats”

Types of Date Formats

MATLAB works with three different date formats: date strings, serial date numbers, and date vectors.

When dealing with dates you typically work with date strings (16-Sep-1996). MATLAB works internally with *serial date numbers* (729284). A serial date represents a calendar date as the number of days that has passed since a fixed base date. In MATLAB, serial date number 1 is January 1, 0000. MATLAB also uses serial time to represent fractions of days beginning at midnight; for example, 6 p.m. equals 0.75 serial days. So the string '16-Sep-1996, 6:00 pm' in MATLAB is date number 729284.75.

All functions that require dates accept either date strings or serial date numbers. If you are dealing with a few dates at the MATLAB command-line level, date strings are more convenient. If you are using functions that handle large numbers of dates or doing extensive calculations with dates, you will get better performance if you use date numbers.

Date vectors are an internal format for some MATLAB functions; you do not typically use them in calculations. A date vector contains the elements [year month day hour minute second].

MATLAB provides functions that convert date strings to serial date numbers, and vice versa. Dates can also be converted to date vectors.

Here are examples of the three date formats used by MATLAB.

Date Format Example

Date string 02-Oct-1996 Serial date number 729300 Date vector 1996 10 2 0 0 0

Conversions Between Date Formats

Functions that convert between date formats are shown below.

Function Description

`datenum` Convert date string to serial date number
`datestr` Convert serial date number to date string
`datevec` Split date number or date string into individual date elements

Here are some examples of conversions from one date format to another. `d1 = datenum('02-Oct-1996')`

```
d1 =
729300
d2 = datestr(d1+10)
d2 =
12-Oct-1996
dv1 = datevec(d1)
dv1 =
1996 10 2 0 0 0
```

```
dv2 = datevec(d2)
dv2 =
1996 12 2 0 0 0
```

Date String Formats

The datenum function is important for doing date calculations efficiently. datenum takes an input string in any of several formats, with 'dd-mmm-yyyy',

'mm/dd/yyyy', or 'dd-mmm-yyyy', hh:mm:ss.ss' most common. You can form up to six fields from letters and digits separated by any other characters:

- The day field is an integer from 1 to 31.
- The month field is either an integer from 1 to 12 or an alphabetic string with at least three characters.
- The year field is a non-negative integer: if only two digits are specified, then a year 19yy is assumed; if the year is omitted, then the current year is used as a default.
- The hours, minutes, and seconds fields are optional. They are integers separated by colons or followed by 'AM' or 'PM'.

For example, if the current year is 1996, then these are all equivalent

```
'17-May-1996'
'17-May-96'
'17-May'
'May 17, 1996'
'5/17/96'
'5/17'
```

and both of these represent the same time

```
'17-May-1996, 18:30'
'5/17/96/6:30 pm'
```

Note that the default format for numbers-only input follows the American convention. Thus 3/6 is March 6, not June 3.

If you create a vector of input date strings, use a column vector and be sure all strings are the same length. Fill in with spaces or zeros.

Output Formats

The function datestr(D,dateform) converts a serial date D to one of 19

different date string output formats showing date, time, or both. The default output for dates is a day-month-year string: 01-Mar-1996. You select an alternative output format by using the optional integer argument dateform. This table shows the date string formats that correspond to each dateform value.

dateform Format Description

0 01-Mar-1996 15:45:17 day-month-year hour:minute:second

1 01-Mar-1996 day-month-year

2 03/01/96 month/day/year

- 3 Mar month, three letters
- 4 M month, single letter
- 5 3 month
- 6 03/01 month/day
- 7 1 day of month
- 8 Wed day of week, three letters
- 9 W day of week, single letter
- 10 1996 year, four digits
- 11 96 year, two digits
- 12 Mar96 month year
- 13 15:45:17 hour:minute:second
- 14 03:45:17 PM hour:minute:second AM or PM
- 15 15:45 hour:minute
- 16 03:45 PM hour:minute AM or PM
- 17 Q1-96 calendar quarter-year
- 18 Q1 calendar quarter

Here are some examples of converting the date March 1, 1996 to various forms using the datestr function.

```
d = '01-Mar-1999'
```

```
d =
```

```
01-Mar-1999
```

```
datestr(d)
```

```
ans =
```

```
01-Mar-1999
```

```
datestr(d, 2)
```

```
ans =
```

```
03/01/99
```

```
datestr(d, 17)
```

```
ans =
```

```
Q1-99
```

Current Date and Time

The function date returns a string for today's date. date

```
ans =
```

```
02-Oct-1996
```

The function now returns the serial date number for the current date and time. now

```
ans =
```

```
729300.71
```

```
datestr(now)
```

```
ans =
```

```
02-Oct-1996 16:56:16
```

```
datestr(floor(now))
```

```
ans =
```

```
02-Oct-1996
```

Obtaining User Input

Obtaining User Input

There are three ways to obtain input from a user during M-file execution. You can:

- Display a prompt and obtain keyboard input.
- Pause until the user presses a key.
- Build a complete graphical user interface.

This section covers the first two topics. The third topic is discussed in online documentation under “Creating Graphical User Interfaces”.

Prompting for Keyboard Input

The `input` function displays a prompt and waits for a user response. Its syntax is
`n = input('prompt_string')`

The function displays the *prompt_string*, waits for keyboard input, and then returns the value from the keyboard. If the user inputs an expression, the function evaluates it and returns its value. This function is useful for implementing menu-driven applications.

`input` can also return user input as a string, rather than a numeric value. To obtain string input, append's' to the function's argument list. `name = input('Enter address: ','s')`

Pausing During Execution

Some M-files benefit from pauses between execution steps. For example, the `petals.mscript`, shown in the “Simple Script Example” section, pauses between the plots it creates, allowing the user to display a plot for as long as desired and then press a key to move to the next plot.

The `pause` command, with no arguments, stops execution until the user presses a key. To pause for *n* seconds, use

`pause(n)`

Shell Escape Functions

It is sometimes useful to access your own C or Fortran programs using *shell escape functions*. Shell escape functions use the shell escape command ! to make external stand-alone programs act like new MATLAB functions. A shell escape M-function is an M-file that:

- 1 Saves the appropriate variables on disk.
- 2 Runs an external program (which reads the data file, processes the data, and writes the results back out to disk).
- 3 Loads the processed file back into the workspace.

For example, look at the code `forgarfield.m`, below. This function uses an external function, `gareqn`, to find the solution to Garfield’s equation.

```
function y = garfield(a,b,q,r)
save gardata a b q r
!gareqn
load gardata
```

This M-file:

- 1 Saves the input arguments *a*, *b*, *q*, and *r* to a MAT-file in the workspace using the `save` command.
- 2 Uses the shell escape operator to access a C, or Fortran program called `gareqn` that uses the workspace variables to perform its computation. `gareqn` writes its results to the `gardata` MAT-file.

3 Loads the `thegardata` MAT-file to obtain the results.

Optimizing MATLAB Code

This section describes techniques that often improve the execution speed and memory management of MATLAB code:

- Vectorizing loops
- Preallocating Arrays

MATLAB is a matrix language, which means it is designed for vector and matrix operations. For best performance, you should take advantage of this where possible.

For information on how to conserve memory and improve memory use, see the section, “[Making Efficient Use of Memory](#)” on page 17-74.

See “[Improving M-File Performance - the Profiler](#)” for information on using the MATLAB Profiler to identify which parts of your code consume the most time.

Vectorizing Loops

You can speed up your M-file code by vectorizing algorithms. *Vectorization* means converting `for` and `while` loops to equivalent vector or matrix operations.

A Simple Example

Here is one way to compute the sine of 1001 values ranging from 0 to 10.

```
i = 0;  
for t = 0:0.01:10  
    i = i+1;  
  
    y(i) = sin(t);  
end
```

A vectorized version of the same code is:

```
t = 0:0.01:10;  
y = sin(t);
```

The second example executes much faster than the first and is the way MATLAB is meant to be used. Test this on your system by creating M-file scripts that contain the code shown, then using the `tic` and `toc` commands to time the M-files.

An Advanced Example

`repmat` is an example of a function that takes advantage of vectorization. It accepts three input arguments: an array `A`, a row dimension `M`, and a column dimension `N`.

`repmat` creates an output array that contains the elements of `arrayA`, replicated and “tiled” in an `M`-by-`N` arrangement.

```
A = [1 2 3; 4 5 6];  
B = repmat(A,2,3);  
B =
```

```
1 2 3 1 2 3 1 2 3
```

```
4 5 6 4 5 6 4 5 6  
1 2 3 1 2 3 1 2 3  
4 5 6 4 5 6 4 5 6
```

repmat

uses vectorization to create the indices that place elements in the output array.

```
function B = repmat(A,M,N)
```

```
if nargin < 2
```

```
error('Requires at least 2 inputs.')
```

```
elseif nargin == 2
```

```
N = M;
```

```
end
```

```
% Step 1 Get row and column sizes [m,n] = size(A);
```

```
% Step 2 Generate vectors of indices from 1 to row/column size mind = (1:m)';  
nind = (1:n)';
```

```
% Step 3 Creates index matrices from vectors above mind = mind(:,ones(1,M));  
nind = nind(:,ones(1,N));
```

```
% Step 4 Create output array B = A(mind,nind);
```

Step 1, above, obtains the row and column sizes of the input array.

Step 2 creates two column vectors. mind contains the integers from 1 through the row size of A. The nind variable contains the integers from 1 through the column size of A.

Step 3 uses a MATLAB vectorization trick to replicate a single column of data through any number of columns. The code is

```
B = A(:,ones(1,n_cols))
```

where n_cols is the desired number of columns in the resulting matrix.

Step 4 uses array indexing to create the output array. Each element of the row index array, mind, is paired with each element of the column index array, nind, using the following procedure:

1 The first element of mind, the row index, is paired with each element of nind. MATLAB moves through the nind matrix in a columnwise fashion, so mind(1,1) goes with nind(1,1), then nind(2,1), and so on. The result fills the first row of the output array.

2 Moving columnwise through mind, each element is paired with the elements of nind as above. Each complete pass through the nind matrix fills one row of the output array.

Preallocating Arrays

You can often improve code execution time by preallocating the arrays that store output results. Preallocation prevents MATLAB from having to resize an array each time you enlarge it. Use the appropriate preallocation function for the kind of array you are working with.

Array Type Function Examples

Numeric array

Cell array

```
zeros y = zeros(1,100);
```

```
cell B = cell(2,3); B{1,3} = 1:3; B{2,2} = 'string';
```

Array Type Function Examples

Structure array

```
struct, data = repmat(struct('x',[1 3],... repmat 'y',[5 6]), 1, 3);
```

Preallocation also helps reduce memory fragmentation if you work with large matrices. In the course of a MATLAB session, memory can become fragmented due to dynamic memory allocation and deallocation. This can result in plenty of free memory, but not enough contiguous space to hold a large variable. Preallocation helps prevent this by allowing MATLAB to “grab” sufficient space for large data constructs at the beginning of a computation.

Making Efficient Use of Memory

This section discusses the following ways to conserve memory and improve memory use:

- “Memory Management Functions”
- “Removing a Function From Memory”
- “Nested Function Calls”
- “Variables and Memory”
- “PC-Specific Topics”
- “UNIX-Specific Topics”
- “What Does “Out of Memory” Mean?”

Memory Management Functions

MATLAB has five functions to improve how memory is handled:

- clear removes variables from memory.
- pack saves existing variables to disk, then reloads them contiguously. Because of time considerations, you should not use pack within loops or M-file functions.
- quit exits MATLAB and returns all allocated memory to the system.
- save selectively stores variables to disk.
- load reloads a data file saved with the save command.

Note save and load are faster than MATLAB low-level file I/O routines. save and load have been optimized to run faster and reduce memory fragmentation.

On some systems, the whos function displays the amount of free memory remaining. However, be aware that:

- If you delete a variable from the workspace, the amount of free memory indicated by whos usually does not get larger unless the deleted variable occupied the highest memory addresses. The number actually indicates the amount of contiguous, unused memory. Clearing the highest variable makes the number larger, but clearing a variable beneath the highest variable has no effect. This means that you might have more free memory than is indicated by whos.
- Computers with virtual memory do not display the amount of free memory remaining because

neither MATLAB nor the hardware imposes limitations.

Removing a Function From Memory

MATLAB creates a list of M- and MEX-filenames at startup for all files that reside below thematlab/toolbox directories. This list is stored in memory and is freed only when a new list is created during a call to the path function. Function M-file code and MEX-file relocatable code are loaded into memory when the corresponding function is called. The M-file code or relocatable code is removed from memory when:

- The function is called again and a new version now exists.
- The function is explicitly cleared with theclear command.
- All functions are explicitly cleared with theclear functions command.
- MATLAB runs out of memory.

Nested Function Calls

The amount of memory used by nested functions is the same as the amount used by calling them on consecutive lines. These two examples require the same amount of memory.

```
result = function2(function1(input99));  
result = function1(input99); result = function2(result);
```

Variables and Memory

Memory is allocated for variables whenever the left-hand side variable in an assignment does not exist. The statement

```
x = 10  
allocates memory, but the statement  
x(10) = 1  
does not allocate memory if the 10th element of x exists.  
To conserve memory:
```

- Avoid creating large temporary variables, and clear temporary variables when they are no longer needed.
- When working with arrays of fixed size, pre-allocate them rather than having MATLAB resize the array each time you enlarge it.
- Set variables equal to the empty matrix[] to free memory, or clear them using clear variable_name
- Reuse variables as much as possible.

Global Variables. Declaring variables asglobal merely puts a flag in a symbol table. It does not use any more memory than defining nonglobal variables. Consider the following example.

```
global a  
a = 5;
```

Now there is one copy of a stored in the MATLAB workspace. Typing clear a removes a from the MATLAB workspace, but it still exists in theglobal workspace.

`clear global a` removes `a` from the global workspace.

PC-Specific Topics

- There are no functions implemented to manipulate the way MATLAB handles Microsoft Windows system resources. Windows uses system resources to track fonts, windows, and screen objects. Resources can be depleted by using multiple figure windows, multiple fonts, or several Uicontrols. The best way to free up system resources is to close all inactive windows. Iconified windows still use resources.
- The performance of a permanent swap file is typically better than a temporary swap file.
- Typically a swap file twice the size of the installed RAM is sufficient.

UNIX-Specific Topics

- Memory that MATLAB requests from the operating system is not returned to the operating system until the MATLAB process is finished.
- MATLAB requests memory from the operating system when there is not enough memory available in the MATLAB heap to store the current variables. It reuses memory in the heap as long as the size of the memory segment required is available in the MATLAB heap.

For example, on one machine these statements use approximately 15.4 MB of RAM.

```
a = rand(1e6,1);  
b = rand(1e6,1);
```

These statements use approximately 16.4 MB of RAM.

```
c = rand(2.1e6,1);
```

These statements use approximately 32.4 MB of RAM.

```
a = rand(1e6,1);  
b = rand(1e6,1);  
clear  
c = rand(2.1e6,1);
```

This is because MATLAB is not able to fit a 2.1 MB array in the space previously occupied by two 1 MB arrays. The simplest way to prevent overallocation of memory, is to preallocate the largest vector. This series of statements uses approximately 32.4 MB of RAM

```
a = rand(1e6,1);  
b = rand(1e6,1);  
clear  
c = rand(2.1e6,1);
```

while these statements use only about 16.4 MB of RAM

```
c = rand(2.1e6,1);  
clear  
a = rand(1e6,1);  
b = rand(1e6,1);
```

Allocating the largest vectors first allows for optimal use of the available memory.

What Does“OutofMemory” Mean?

Typically the Out of Memory message appears because MATLAB asked the operating system for a segment of memory larger than what is currently available. Use any of the techniques discussed in this section to help optimize the available memory. If the Out of Memory message still appears:

- Increase the size of the swap file.
- Make sure that there are no external constraints on the memory accessible to MATLAB (on UNIX systems use the limit command to check).
- Add more memory to the system.
- Reduce the size of your data.

18

Character Arrays (Strings)

Character Arrays	18-5
Creating Character Arrays	18-5
Creating Two-Dimensional Character Arrays.....	18-6
Converting Characters to Numeric Values	18-7
Cell Arrays of Strings	18-8 Converting to a Cell
Array of Strings.....	18-8

String Comparisons	18-10
Comparing Strings For Equality.....	18-10
Comparing for Equality Using Operators	18-11
Categorizing Characters Within a String	18-12

Searching and Replacing	18-13
--------------------------------------	--------------

String/Numeric Conversion.....	18-15
Array/String Conversion	18-16

This chapter explains MATLAB’s support for string data. It describes the two ways that MATLAB represents strings:

- “Character Arrays”
- “Cell Arrays of Strings”

It also describes the operations that you can perform on string data under the following topics:

- “String Comparisons”
- “Searching and Replacing”
- “String/Numeric Conversion”

This table shows the string functions, which are located in the directory named strfun in the MATLAB Toolbox.

Category Function Description General blanks String of blanks cellstr Create cell array of

strings from character array
char
deblank
eval String Tests iscellstr
ischar
isletter
isspace

Create character array (string)
Remove trailing blanks
Execute string with MATLAB expression True for cell array of strings
True for character array
True for letters of alphabet.
True for whitespace characters.

Category Function Description String

Operations

findstr Find one string within another lower Convert string to lowercase
strcat Concatenate strings
strcmp Compare strings strcmpi Compare strings, ignoring case
strjust Justify string
strmatch Find matches for string strncmp Compare first N characters of strings
strncmpi Compare first N characters, ignoring case
strrep Replace string with another
strtok Find token in string
strvcat Concatenate strings vertically upper Convert string to uppercase
String to Number Conversion
double Convert string to numeric codes
int2str Convert integer to string
mat2str Convert matrix to eval'able string
num2str Convert number to string
sprintf Write formatted data to string
str2double Convert string to double-precision value
str2num Convert string to number
scanf Read string under format control

Category Function Description Base Number Conversion

base2dec Convert base B string to decimal integer
bin2dec Convert binary string to decimal integer
dec2base Convert decimal integer to base B string
dec2bin Convert decimal integer to binary string
dec2hex Convert decimal integer to hexadecimal string
hex2dec Convert hexadecimal string to decimal integer
hex2num Convert IEEE hexadecimal to double-precision number

Character Arrays

In MATLAB, the term *string* refers to an array of characters. MATLAB represents each character internally as its corresponding numeric value. Unless you want to access these values, however, you can simply work with the characters as they display on screen.

This section covers:

- “Creating Character Arrays”
- “Creating Two-Dimensional Character Arrays”
- “Converting Characters to Numeric Values”

Creating Character Arrays

Specify character data by placing characters inside a pair of single quotes. For example, this line creates a 1-by-13 character array called name.

```
name = 'Thomas R. Lee';  
In the workspace, the output of whos shows  
Name Size Bytes Class
```

name 1x13 26 char array

You can see that a character uses two bytes of storage internally. The class and ischar functions show name's identity as a character array.

```
class(name)  
ans =  
char  
ischar(name)  
ans =  
1
```

You can also join two or more character arrays together to create a new character array. Use either the string concatenation function, strcat, or the MATLAB concatenation operator, [], to do this. The latter preserves any trailing spaces found in the input arrays.

```
name = 'Thomas R. Lee';  
title = ' Sr. Developer';  
strcat(name,',',title)  
  
ans =  
Thomas R. Lee, Sr. Developer  
You can also concatenate strings vertically with strvcat.
```

Creating Two-Dimensional Character Arrays

When creating a two-dimensional character array, be sure that each row has the same length. For example, this line is legal because both input rows have exactly 13 characters.

```
name = ['Thomas R. Lee' ; 'Sr. Developer']  
name =  
Thomas R. Lee  
Sr. Developer
```

When creating character arrays from strings of different lengths, you can pad the shorter strings with blanks to force rows of equal length.

```
name = ['Thomas R. Lee' ; 'Senior Developer'];
```

A simpler way to create string arrays is to use the char function. char automatically pads all strings to the length of the longest input string. In this example, char pads the 13-character input string 'Thomas R. Lee' with three trailing blanks so that it will be as long as the second string.

```
name = char('Thomas R. Lee','Senior Developer')  
name =  
Thomas R. Lee Senior Developer When extracting strings from an array, use the deblank function to remove any trailing blanks.  
trimname = deblank(name(1,:))
```

```
trimname =  
Thomas R. Lee  
size(trimname)  
ans =  
1 13
```

Converting Characters to Numeric Values

Character arrays store each character as a 16-bit numeric value. Use the double function to convert strings to their numeric values, and char to revert to character representation.

```
name = double(name)  
name =  
84 104 111 109 97 115 32 82 46 32 76 101 101  
name = char(name) name =  
Thomas R. Lee Use str2num to convert a character array to the numeric value represented by that string.  
str = '37.294e-1';  
val = str2num(str)  
val =
```

Cell Arrays of Strings

It's often convenient to store groups of strings in cell arrays instead of standard character arrays. This prevents you from having to pad strings with blanks to create character arrays with rows of equal length. A set of functions enables you to work with cell arrays of strings:

- You can convert between standard character arrays and cell arrays of strings.
- You can apply string comparison operations to cell arrays of strings. For details on cell arrays see the “Structures and Cell Arrays” chapter.

Converting to a Cell Array of Strings

The cellstr function converts a character array into a cell array of strings. Consider the character array

```
data = ['Allison Jones'; 'Development'; 'Phoenix']; Each row of the matrix is padded so that all have equal length (in this case, 13 characters).
```

Now use cellstr to create a column vector of cells, each cell containing one of the strings from the data array.

```
celldata = cellstr(data)  
celldata =
```

```
'Allison Jones' 'Development' 'Phoenix'
```

Note that the cellstr function strips off the blanks that pad the rows of the input string matrix.

```
length(celldata{3})  
ans =  
7
```

Cell Arrays of Strings

The iscellstr function determines if the input argument is a cell array of strings. It returns a logical true (1) in the case of celldata. iscellstr(celldata)

```
ans =
1 Usechar to convert back to a standard padded character array.
strings = char(celldata)
strings =
Allison Jones
Development
Phoenix
```

String / Numeric Conversion

The `str2double` function converts a cell array of strings to the double-precision values represented by the strings.

```
c = {'37.294e-1'; '-58.375'; '13.796'};
str2double(c)
ans =
3.7294
-58.3750
```

String Comparisons

There are several ways to compare strings and substrings:

- You can compare two strings, or parts of two strings, for equality.
- You can compare individual characters in two strings for equality.
- You can categorize every element within a string, determining whether each element is a character or whitespace.

These functions work for both character arrays and cell arrays of strings.

Comparing Strings For Equality

There are four functions that determine if two input strings are identical:

- `strcmp` determines if two strings are identical.
- `strncmp` determines if the first n characters of two strings are identical.
- `strcmpi` and `strncmpi` are the same as `strcmp` and `strncmp`, except that they ignore case.

Consider the two strings

```
str1 = 'hello';
```

```
str2 = 'help';
```

Strings `str1` and `str2` are not identical, so invoking `strcmp` returns 0 (false). For example,

```
C = strcmp(str1,str2)
```

```
C =
```

```
0
```

Note For C programmers, this is an important difference between MATLAB's `strcmp` and C's `strcmp()`, which returns 0 if the two strings are the same. String Comparisons

The first three characters of `str1` and `str2` are identical, so invoking `strncmp` with any value up to 3 returns 1.

```
C = strncmp(str1,str2,2)
```

```
C =
```

1 These functions work cell-by-cell on a cell array of strings. Consider the two cell arrays of

```
strings  
A = {'pizza';'chips';'candy'};  
B = {'pizza';'chocolate';'pretzels'};  
Now apply the string comparison functions.  
strcmp(A,B)  
ans =
```

```
1  
0  
0
```

```
strncmp(A,B,1)  
ans =
```

```
1  
1  
0
```

Comparing for Equality Using Operators

You can use MATLAB relational operators on character arrays, as long as the arrays you are comparing have equal dimensions, or one is a scalar. For example, you can use the equality operator (==) to determine which characters in two strings match.

```
A = 'fate'; B = 'cake'; A == B
```

```
ans =  
0 1 0 1
```

All of the relational operators (>,>=,<,<=,==,!=) compare the values of corresponding characters.

Categorizing Characters Within a String

There are two functions for categorizing characters inside a string:

- isletter determines if a character is a letter
- isspace determines if a character is whitespace (blank, tab, or new line)

For example, create a string named mystring.

```
mystring = 'Room 401';
```

isletter examines each character in the string, producing an output vector of the same length as mystring.

```
A = isletter(mystring)
```

```
A =
```

1 1 1 1 0 0 0 0 The first four elements in A are 1 (true) because the first four characters of mystring are letters.

Searching and Replacing

Searching and Replacing

MATLAB provides several functions for searching and replacing characters in a string. Consider a string named label.

```
label = 'Sample 1, 10/28/95';
```

The `strrep` function performs the standard search-and-replace operation. Use `strrep` to change the date from '10/28' to '10/30'.

```
newlabel = strrep(label,'28','30')
```

```
newlabel =
```

Sample 1, 10/30/95 `findstr` returns the starting position of a substring within a longer string. To find all occurrences of the string 'amp' inside `label`

```
position = findstr('amp',label)
```

```
position =
```

2 The position within `label` where the only occurrence of 'amp' begins is the second character.

The `strtok` function returns the characters before the first occurrence of a delimiting character in an input string. The default delimiting characters are the set of whitespace characters. You can use the `strtok` function to parse a sentence into words; for example,

```
function all_words = words(input_string)
```

```
remainder = input_string;
```

```
all_words = ":";
```

```
while (any(remainder))
```

```
[chopped,remainder] = strtok(remainder); all_words = strvcat(all_words,chopped);
```

```
end
```

The `strmatch` function looks through the rows of a character array or cell array of strings to find strings that begin with a given series of characters. It returns the indices of the rows that begin with these characters.

```
maxstrings = strvcat('max','minimax','maximum')
```

```
maxstrings =
```

```
max
```

```
minimax maximum
```

```
strmatch('max',maxstrings)
```

```
ans =
```

```
1 3
```

String / Numeric Conversion

String / Numeric Conversion

MATLAB's string/numeric conversion functions change numeric values into character strings. You can store numeric values as digit-by-digit string representations, or convert a value into a hexadecimal or binary string. Consider a the scalar

```
x = 5317;
```

By default, MATLAB stores the number `x` as a 1-by-1 double array containing the value 5317. The `int2str` (integer to string) function breaks this scalar into a 1-by-4 vector containing the

```
string'5317'.
```

```
y = int2str(x);
size(y)
ans =
1 4
```

A related function, num2str, provides more control over the format of the output string. An optional second argument sets the number of digits in the output string, or specifies an actual format.

```
p = num2str(pi,9)
p =
3.14159265 Both int2str and num2str are handy for labeling plots. For example, the following lines use num2str to prepare automated labels for the x-axis of a plot.
```

```
function plotlabel(x,y)
plot(x,y)
str1 = num2str(min(x));
str2 = num2str(max(x));
out = ['Value of f from ' str1 ' to ' str2];
xlabel(out);
```

Another class of numeric/string conversion functions changes numeric values into strings representing a decimal value in another base, such as binary or hexadecimal representation. For example, the dec2hex function converts a decimal value into the corresponding hexadecimal string.

```
dec_num = 4035;
hex_num = dec2hex(dec_num)
hex_num =
FC3 See the strfun directory for a complete listing of string conversion functions.
```

Arra y / String Conversion

The MATLAB function mat2str changes an array to a string that MATLAB can evaluate. This string is useful input for a function such as eval, which evaluates input strings just as if they were typed at the MATLAB command line.

Create a 2-by-3 array A.

```
A = [1 2 3; 4 5 6]
```

```
A =
```

1 2 3 4 5 6 mat2str returns a string that contains the text you would enter to create A at the command line.

```
B = mat2str(A)
B = [1 2 3; 4 5 6]
```

Multidimensional Arrays

Multidimensional Arrays	19-3
Creating Multidimensional Arrays	19-4
Accessing Multidimensional Array Properties.....	19-8
Indexing	19-9
Reshaping.....	19-10
Permuting Array Dimensions	19-12
Computing w ith Multidimensional Arrays	19-14
Operating on Vectors.....	19-14
Operating Element-by-Element.....	19-14
Operating on Planes and Matrices.....	19-15
Organizing Data in Multidimensional Arrays.....	19-16
Multidimensional Cell Arrays.....	19-18
Multidimensional Structure Arrays	19-19 Applying Functions to Multidimensional Structure Arrays . 19-20 This chapter discusses <i>multidimensional arrays</i> , MATLAB arrays with more than two dimensions. Multidimensional arrays can be numeric, character, cell, or structure arrays. These arrays are broadly useful—for example, in the representation of multivariate data, or multiple pages of two-dimensional data.

This chapter covers the following topics:

- “Multidimensional Arrays”
- “Computing with Multidimensional Arrays”
- “Organizing Data in Multidimensional Arrays”
- “Multidimensional Cell Arrays”
- “Multidimensional Structure Arrays”

MATLAB provides a number of functions that directly support multidimensional arrays. You can extend this support by creating M-files that work with your data architecture.

Function Description cat Concatenate arrays. ipermute Inverse permute array dimensions.
ndgrid Generate arrays for N-D functions and interpolation. ndims Number of array dimensions.
permute Permute array dimensions. shiftdim Shift dimensions. squeeze Remove singleton dimensions.

Multidimensional Arrays

Multidimensional arrays in MATLAB are an extension of the normal two-dimensional matrix. Matrices have two dimensions: the row dimension and the column dimension.

column
(1,1) (1,2) (1,3) (1,4)
row (2,1) (2,2) (2,3) (2,4)
(3,1) (3,2) (3,3) (3,4)

(4,1) (4,2) (4,3) (4,4)

You can access a two-dimensional matrix element with two subscripts: the first representing the row index, and the second representing the column index. Multidimensional arrays use additional subscripts for indexing. A three-dimensional array, for example, uses three subscripts:

- The first references array dimension 1, the row.
- The second references dimension 2, the column.
- The third references dimension 3. This guide uses the concept of a *page* to represent dimensions 3 and higher.

(1,1,3) (1,2,3) (1,3,3) (1,4,3)

(2,1,3) (2,2,3) (2,3,3) (2,4,3) page

(3,1,3) (3,2,3) (3,3,3) (3,4,3)

(1,1,2) (1,2,2) (1,3,2) (1,4,2) 4,2,3) (4,3,3) (4,4,3)

(2,1,2) (2,2,2) (2,3,2) (2,4,2)

column (3,1,2) (3,2,2) (3,3,2) (3,4,2)

(1,1,1) (1,2,1) (1,3,1) (1,4,1) (4,1,2) (4,2,2) (4,3,2) (4,4,2)

(2,1,1) (2,2,1) (2,3,1) (2,4,1)_{row}

(3,1,1) (3,2,1) (3,3,1) (3,4,1) (4,1,1) (4,2,1) (4,3,1) (4,4,1)

To access the element in the second row, third column of page 2, for example, you use the subscripts(2,3,2).

$A(:,:,1) = A(2,3,2)$

1 0 3 4 -1 2

6 8 3₈ 2 1

4 3 6

5 9 2

1 0 3 A(:,:,2) = 4 -1 2

8 2 1₆ 8 3 4 3 6 5 9 2

As you add dimensions to an array, you also add subscripts. A four-dimensional array, for example, has four subscripts. The first two reference a row-column pair; the second two access the third and fourth dimensions of data.

Note The general multidimensional array functions reside in the `datatypes` directory.

Creating Multidimensional Arrays

You can use the same techniques to create multidimensional arrays that you use for two-dimensional matrices. In addition, MATLAB provides a special concatenation function that is useful for building multidimensional arrays.

This section discusses:

- Generating arrays using indexing
- Generating arrays using MATLAB functions
- Using the cat function to build multidimensional arrays

Generating Arrays Using Indexing

One way to create a multidimensional array is to create a two-dimensional array and extend it. For example, begin with a simple two-dimensional array A.

A = [5 7 8; 0 1 9; 4 3 6]; A is a 3-by-3 array, that is, its row dimension is 3 and its column dimension is 3. To add a third dimension to A,

A(:,:,2) = [1 0 4; 3 5 6; 9 8 7]

MATLAB responds with

A(:,:,1) =

```
5 7 8
0 1 9
4 3 6
```

A(:,:,2) =

```
1 0 4
3 5 6
9 8 7
```

You can continue to add rows, columns, or pages to the array using similar assignment statements.

Extending Multidimensional Arrays. To extend A in any dimension:

- Increment or add the appropriate subscript and assign the desired values.
- Assign the same number of elements to corresponding array dimensions. For numeric arrays, all rows must have the same number of elements, all pages must have the same number of rows and columns, and so on.

You can take advantage of MATLAB's scalar expansion capabilities, together with the colon operator, to fill an entire dimension with a single value.

A(:,:,3) = 5;

A(:,:,3)

ans =

```
5 5 5
5 5 5
5 5 5
```

To turn A into a 3-by-3-by-3-by-2, four-dimensional array, enter

A(:,:,1,2) = [1 2 3; 4 5 6; 7 8 9];

A(:,:,2,2) = [9 8 7; 6 5 4; 3 2 1];

A(:,:,3,2) = [1 0 1; 1 1 0; 0 1 1];

Note that after the first two assignments MATLAB padsA with zeros, as needed, to maintain the corresponding sizes of dimensions.

Generating Arrays Using MATLAB Functions

You can use MATLAB functions such as randn, ones, and zeros to generate multidimensional arrays in the same way you use them for two-dimensional arrays. Each argument you supply represents the size of the corresponding dimension in the resulting array. For example, to create a 4-by-3-by-2 array of normally distributed random numbers.

```
B = randn(4,3,2)
```

To generate an array filled with a single constant value, use the repmat function. repmat replicates an array (in this case, a 1-by-1 array) through a vector of array dimensions.

```
B = repmat(5,[3 4 2])
```

```
B(:,:,1) =
```

```
5 5 5
```

```
5 5 5
```

```
5 5 5
```

```
B(:,:,2) =
```

```
5 5 5
```

```
5 5 5
```

```
5 5 5
```

Note Any dimension of an array can have size zero, making it a form of empty array. For example, 10-by-0-by-20 is a valid size for a multidimensional array.

Building Multidimensional Arrays with the cat Function

The cat function is a simple way to build multidimensional arrays; it concatenates a list of arrays along a specified dimension.

```
B = cat(dim,A1,A2...)
```

where A1, A2, and so on are the arrays to concatenate, and dim is the dimension along which to concatenate the arrays. For example, to create a new array with cat

```
B = cat(3,[2 8; 0 5],[1 3; 7 9])
```

```
B(:,:,1) =
```

```
2 8 0 5
```

```
B(:,:,2) =
```

```
1 3 7 9
```

The cat function accepts any combination of existing and new data. In addition, you can nest calls to cat. The lines below, for example, create a four-dimensional array.

```
A = cat(3,[9 2; 6 5],[7 1; 8 4])
```

```
B = cat(3,[3 5; 0 1],[5 6; 2 1])
D = cat(4,A,B,cat(3,[1 2; 3 4],[4 3;2 1]))
```

cat

automatically adds subscripts of 1 between dimensions, if necessary. For example, to create a 2-by-2-by-1-by-2 array, enter

```
C = cat(4,[1 2; 4 5],[7 8; 3 2])
```

In the previous case, cat inserts as many singleton dimensions as needed to create a four-dimensional array whose last dimension is not a singleton dimension. If the dim argument had been 5, the previous statement would have produced a 2-by-2-by-1-by-1-by-2 array. This adds additional 1s to indexing expressions for the array. To access the value 8 in the four-dimensional case, use

```
C(1,2,1,2)
```

Singleton dimension index

Accessing Multidimensional Array Properties

You can use the following MATLAB functions to get information about multidimensional arrays you have created.

Information Function Example Array size size(C)

```
ans =
2 2 1 2 rows columns dim3 dim4 Array ndims dimensions
ndims(C)
ans =
4
```

Array whos storage and
format

```
whos
Name Size Bytes Class
```

```
A 2x2x2 64 double array B 2x2x2 64 double array C 4-D 64 double array D 4-D 192 double
array
```

Grand total is 48 elements using 384 bytes

Indexing

Many of the concepts that apply to two-dimensional matrices extend to multidimensional arrays as well.

To access a single element of a multidimensional array, use integer subscripts. Each subscript indexes a dimension—the first indexes the row dimension, the second indexes the column dimension, the third indexes the first page dimension, and so on.

Consider a 10-by-5-by-3 array nddata of random integers:

```
nddata = fix(8*randn(10,5,3));
```

To access element(3,2) on page 2 of nddata, for example, use nddata(3,2,2).

You can use vectors as array subscripts. In this case, each vector element must be a valid subscript, that is, within the bounds defined by the dimensions of the array. To access elements(2,1), (2,3), and (2,4) on page 3 of nndata, use

```
nndata(2,[1 3 4],3);
```

The Colon and Multidimensional Array Indexing

MATLAB's colon indexing extends to multidimensional arrays. For example, to access the entire third column on page 2 of nndata, use nndata(:,3,2).

The colon operator is also useful for accessing other subsets of data. For example, nndata(2:3,2:3,1) results in a 2-by-2 array, a subset of the data on page 1 of nndata. This matrix consists of the data in rows 2 and 3, columns 2 and 3, on the first page of the array.

The colon operator can appear as an array subscript on both sides of an assignment statement. For example, to create a 4-by-4 array of zeros C = zeros(4,4)

Now assign a 2-by-2 subset of array nndata to the four elements in the center of C.

```
C(2:3,2:3) = nndata(2:3,1:2,2)
```

Avoiding Ambiguity in Multidimensional Indexing Some assignment statements, such as A(:,:,2) = 1:10

are ambiguous because they do not provide enough information about the shape of the dimension to receive the data. In the case above, the statement tries to assign a one-dimensional vector to a two-dimensional destination. MATLAB produces an error for such cases. To resolve the ambiguity, be sure you provide enough information about the destination for the assigned data, and that both data and destination have the same shape. For example,

```
A(1,:,:2) = 1:10;
```

Reshaping

Unless you change its shape or size, a MATLAB array retains the dimensions specified at its creation. You change array size by adding or deleting elements. You change array shape by respecifying the array's row, column, or page dimensions while retaining the same elements. The reshape function performs the latter operation. For multidimensional arrays, its form is

```
B = reshape(A,[s1 s2 s3 ...])
```

s1, s2, and so on represent the desired size for each dimension of the reshaped matrix. Note that a reshaped array must have the same number of elements as the original array (that is, the product of the dimension sizes is constant).

M reshape(M, [6 5])

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

```
1 2 3 4 5^2 4 9 8 2  
9 0 6 3 7^0 3 3 8 1
```

```
8 1 5 0 21 0 6 4 3
```

The reshape function operates in a columnwise manner. It creates the reshaped matrix by taking consecutive elements down each column of the original data construct.

C reshape(C, [6 2])

```
9 10 1 6  
11 12 3 8  
5 6 2 9  
7 8 4 11  
1 2 5 10  
3 4 7 12
```

Here are several new arrays from reshapingnddata.

```
B = reshape(nddata,[6 25])  
C = reshape(nddata,[5 3 10])  
D = reshape(nddata,[5 3 2 5])
```

Removing Singleton Dimensions

MATLAB creates singleton dimensions if you explicitly specify them when you create or reshape an array, or if you perform a calculation that results in an array dimension of one.

```
B = repmat(5,[2 3 1 4]);  
size(B)  
ans =  
2 3 1 4  
The squeeze function removes singleton dimensions from an array. C = squeeze(B);  
size(C)  
ans =  
2 3 4
```

The squeeze function does not affect two-dimensional arrays; row vectors remain rows.

Permuting Array Dimensions

The permute function reorders the dimensions of an array.

```
B = permute(A,dims);
```

dims is a vector specifying the new order for the dimensions of A, where 1 corresponds to the first dimension (rows), 2 corresponds to the second dimension (columns), 3 corresponds to pages, and so on.

```
A B= permute (A,[2 1 3]) C = permute(A,[3 2 1]) A(:,:,1) = B(:,:,1) = C(:,:,1) =
```

```
1 2 3 1 4 7 Rowandcolumn  
4 5 62 5 8 subscriptsare  
7 8 93 6 9 reversed
```

$A(:,:,2) =$
 $B(:,:,2) =$
(page-by-page transposition).

0 5 4^{0 2 9}
2 7 6^{5 7 3}
9 3 1^{4 6 1}

1 2 3 Row and page
0 5 4 subscripts are reversed. $C(:,:,2) =$

4 5 6 2 7 6
 $C(:,:,3) =$
7 8 9 9 3 1

For a more detailed look at the `permute` function, consider a four-dimensional array A of size 5-by-4-by-3-by-2. Rearrange the dimensions, placing the column dimension first, followed by the second page dimension, the first page dimension, then the row dimension. The result is a 4 by 2 by 3 by 5 array.

$B = \text{permute}(A,[2\ 4\ 3\ 1])$

Moved dimension 2 of A to first subscript position of B ,
dimension 4 to second subscript position, and so on.

Input Dimension 1 2 array $A_{\text{Size } 5\ 4\ 3\ 4\ 3\ 2}$

Output Dimension 1 2

array $B_{\text{Size } 4\ 2}$
3 4
3 5

The order of dimensions in `permute`'s argument list determines the size and shape of the output array. In this example, the second dimension moves to the first position. Because the second dimension of the original array had size four, the output array's first dimension also has size four.

You can think of `permute`'s operation as an extension of the `transpose` function, which switches the row and column dimensions of a matrix. For `permute`, the order of the input dimension list determines the reordering of the subscripts. In the example above, element(4,2,1,2) of A becomes element (2,2,1,4) of B , element(5,4,3,2) of A becomes element(4,2,3,5) of B , and so on.

Inverse Permutation

The `ipermute` function is the inverse of `permute`. Given an input array A and a vector of dimensions v , `ipermute` produces an array B such that `permute(B,v)` returns A .

For example, these statements create an array E that is equal to the input array C .

```
D = ipermute(C,[1 4 2 3]);  
E = permute(D,[1 4 2 3])
```

You can obtain the original array after permuting it by calling ipermute with the same vector of dimensions.

Computing with Multidimensional Arrays

Many of MATLAB's computational and mathematical functions accept multidimensional arrays as arguments. These functions operate on specific dimensions of multidimensional arrays; that is, they operate on individual elements, on vectors, or on matrices.

Operating on Vectors

Functions that operate on vectors, like sum, mean, and so on, by default typically work on the first nonsingleton dimension of a multidimensional array. Most of these functions optionally let you specify a particular dimension on which to operate. There are exceptions, however. For example, the cross function, which finds the cross product of two vectors, works on the first nonsingleton dimension having length three.

Note In many cases, these functions have other restrictions on the input arguments – for example, some functions that accept multiple arrays require that the arrays be the same size. Refer to the online help for details on function arguments.

Operating Element-by-Element

MATLAB functions that operate element-by-element on two-dimensional arrays, like the trigonometric and exponential functions in the elfun directory, work in exactly the same way for multidimensional cases. For example, the sin function returns an array the same size as the function's input argument. Each element of the output array is the sine of the corresponding element of the input array.

Similarly, the arithmetic, logical, and relational operators all work with corresponding elements of multidimensional arrays that are the same size in every dimension. If one operand is a scalar and one an array, the operator applies the scalar to each element of the array.

Computing with Multidimensional Arrays

Operating on Planes and Matrices

Functions that operate on planes or matrices, such as the linear algebra and matrix functions in the matfun directory, do not accept multidimensional arrays as arguments. That is, you cannot use the functions in the matfun directory, or the array operators \circ , \wedge , \backslash , or $/$, with multidimensional arguments. Supplying multidimensional arguments or operands in these cases results in an error.

You can use indexing to apply a matrix function or operator to matrices within a multidimensional array. For example, create a three-dimensional array A = cat(3,[1 2 3;9 8 7;4 6 5],[0 3 2;8 8 4;5 3 5],[6 4 7;6 8 5;... 5 4 3]);

Applying the eig function to the entire multidimensional array results in an error.

```
eig(A)
```

```
??? Error using ==> eig  
Input arguments must be 2-D.
```

You can, however, apply eig to planes within the array. For example, use colon notation to index just one page (in this case, the second) of the array.

```
eig(A(:,:,2))  
ans =
```

```
-2.6260  
12.9129  
2.7131
```

Note In the first case, subscripts are not colons, you must use squeeze to avoid an error. For example, eig(A(2,:,:)) results in an error because the size of the input is [1 3 3]. The expression eig(squeeze(A(2,:,:))), however, passes a valid two-dimensional matrix to eig.

Organizing Data in Multidimensional Arrays

You can use multidimensional arrays to represent data in two ways:

- As planes or pages of two-dimensional data. You can then treat these pages as matrices.
- As multivariate or multidimensional data. For example, you might have a four-dimensional array where each element corresponds to either a temperature or air pressure measurement taken at one of a set of equally spaced points in a room.

For example, consider an RGB image. For a single image, a multidimensional array is probably the easiest way to store and access data. ArrayRGB

Page3

—

```
0.689 0.706 0.118 0.884 ... 0.535 0.532 0.653 0.925 ... blue 0.314 0.265 0.159 0.101 ... intensity 0.553  
0.633 0.528 0.493 ...
```

```
0.441 0.465 0.512 0.512 ... page2- 0.342 0.647 0.515 0.816 ... 0.398 0.401 0.421 0.398 ... green  
0.111 0.300 0.205 0.526 ... 0.912 0.713 ...
```

intensity

```
0.523 0.428 0.712 0.929 ... 0.119 0.182 0.219 0.328 ...  
0.214 0.604 0.918 0.344 ... 0.125 0.495 0.128 0.133 ... values 0.100 0.121 0.113 0.126 ... . page1-  
0.112 0.986 0.234 0.432 ... 0.204 0.175 ... . red 0.765 0.128 0.863 0.521 ... 0.760 0.531 ... .
```

intensity

```
1.000 0.985 0.761 0.698 ... 0.997 0.910 ...  
0.455 0.783 0.224 0.395 ... 0.995 0.726 ...  
0.021 0.500 0.311 0.123 ... .  
1.000 1.000 0.867 0.051 ... .
```

```
1.000 0.945 0.998 0.893 ... .  
0.990 0.941 1.000 0.876 ...  
0.902 0.867 0.834 0.798 ...  
. . .
```

To access an entire plane of the image, use `red_plane = RGB(:,:,1);`
Organizing Data in Multidimensional Arrays

To access a subimage, use
`subimage = RGB(20:40,50:85,:);`

The RGB image is a good example of data that needs to be accessed in planes for operations like display or filtering. In other instances, however, the data itself might be multidimensional. For example, consider a set of temperature measurements taken at equally spaced points in a room. Here the location of each value is an integral part of the data set – the physical placement in three-space of each element is an aspect of the information. Such data also lends itself to representation as a multidimensional array.

```
ArrayTEMP  
67.923892 68.023892 67.923892  
67.823892 67.823892 67.923892  
  
67.923892 68.023892 68.0  
67.6 67.523892 67.723892  
67.723892 67.823892 67.723892  
68.023892 68.023892 67.8  
67.723892 67.523892 67.523892  
67.923892 67.823892 67.623892  
67.823892 67.623892 67.623892
```

Now to find the average of all the measurements, use
`mean(mean(mean(TEMP)));`

To obtain a vector of the “middle” values (element (2,2)) in the room on each page, use
`B = TEMP(2,2,:);`

Multidimensional Cell Arrays

Like numeric arrays, the framework for multidimensional cell arrays in MATLAB is an extension of the two-dimensional cell array model. You can use the cat function to build multidimensional cell arrays, just as you use it for numeric arrays.

For example, create a simple three-dimensional cell arrayC.

```
A{1,1} = [1 2;4 5];  
A{1,2} = 'Name';
```

```

A{2,1} = 2-4i;
A{2,2} = 7;
B{1,1} = 'Name2';
B{1,2} = 3;
B{2,1} = 0:1:3;
B{2,2} = [4 5]';
C = cat(3,A,B);

```

The subscripts for the cells of C look like

cell 1,1,2 **cell 1,2,2**
'Name2' 3
cell 2,1,2 **cell 2,2,2**

```

cell 1,1,1 cell 1,2,1[0 1 2 3] 4
5
1 2 'Name'
4 5

```

cell 2,1,1 **cell 2,2,1** 2-4i₇

Multidimensional Structure Arrays

Multidimensional Structure Arrays

Multidimensional structure arrays are extensions of rectangular structure arrays. Like other types of multidimensional arrays, you can build them using direct assignment or the cat function.

```

patient(1,1,1).name = 'John Doe'; patient(1,1,1).billing = 127.00; patient(1,1,1).test = [79 75 73;
180 178 177.5; 220 210 205]; patient(1,2,1).name = 'Ann Lane'; patient(1,2,1).billing = 28.50;
patient(1,2,1).test = [68 70 68; 118 118 119; 172 170 169]; patient(1,1,2).name = 'Al
Smith'; patient(1,1,2).billing = 504.70; patient(1,1,2).test = [80 80 80; 153 153 154; 181 190
182]; patient(1,2,2).name = 'Dora Jones'; patient(1,2,2).billing = 1173.90;
patient(1,2,2).test = [73 73 75; 103 103 102; 201 198 200];

```

```

patient(1,1,2) patient(1,2,2)
.name 'Al Smith' .billing 504.70

.test 80 80 80
153 153 154
181 190 182
.name 'Dora Jones' .billing 1173.90

.test 73 75 75
103 103 102
201 198 200

```

patient(1,1,1)) **patient(1,2,1)**

```

.name
.billing .test
'John Doe'.name 'Ann Lane' 127.00 .billing 28.50

```

```

79 75 73 .test 68 70 68
180 178 177.5 118 118 119
220 210 205 172 170 169

```

Applying Functions to Multidimensional Structure Arrays

To apply functions to multidimensional structure arrays, operate on fields and field elements using indexing. For example, find the sum of the columns of the test array `inpatient(1,1,2)`.

```
sum((patient(1,1,2).test));
```

Similarly, add all the `billing` fields in the `patient` array. `total = sum([patient.billing]);`

20

Structures and Cell Arrays

Structures	20-4
Building Structure Arrays.....	20-5
Accessing Data in Structure Arrays.....	20-7
Finding the size of Structure Arrays.....	20-10
Adding Fields to Structures	20-10
Deleting Fields from Structures.....	20-10
Applying Functions and Operators	20-10
Writing Functions to Operate on Structures.....	20-11
Organizing Data in Structure Arrays	20-12
Nesting Structures.....	20-17
Cell Arrays.....	20-19
Creating Cell Arrays	20-20
Obtaining Data from Cell Arrays.....	20-23
Deleting Cells.....	20-24
Reshaping Cell Arrays	20-25
Replacing Lists of Variables with Cell Arrays	20-25
Applying Functions and Operators	20-27
Organizing Data in Cell Arrays	20-27
Nesting Cell Arrays.....	20-29
Converting Between Cell and Numeric Arrays.....	20-30
Cell Arrays of Structures.....	20-31

Structures are collections of different kinds of data organized by named fields. Cell arrays are a special class of MATLAB array whose elements consist of cells that themselves contain MATLAB arrays. Both structures and cell arrays provide a hierarchical storage mechanism for dissimilar kinds of data. They differ from each other primarily in the way they organize data. You access data in structures using named fields, while in cell arrays, data is accessed through matrix indexing operations.

This table describes the MATLAB functions for working with structures and cell arrays..

Category	Function Description
Structure functions	<code>deal</code> Deal inputs to outputs
Structure field names.	<code>fieldnames</code> Get structure field names.
<code>getfield</code>	
<code>isfield</code>	
<code>issstruct</code>	
<code>rmfield</code>	
<code>setfield</code>	

```
struct  
struct2cell Cell array functions cell  
cell2struct  
celldisp  
cellfun  
cellplot  
deal  
iscell
```

Get structure field contents.

True if field is in structure array. True for structures.

Remove structure field.

Set structure field contents.

Create or convert to structure array. Convert structure array into cell array. Create cell array.

Convert cell array into structure array. Display cell array contents.

Apply a cell function to a cell array. Display graphical depiction of cell array. Deal inputs to outputs.

True for cell array.

Category	Function Description
strings functions	num2cell Convert numeric array into cell array. Cell array of strings
deblank	Remove trailing blanks from a string
intersect	ismember
setdiff	setxor
sort	strcat
strcmp	strmatch
union	unique

Set the intersection of two vectors

Detect members of a set

Return the set difference of two vectors

Set the exclusive-or of two vectors

Sort elements in ascending order

Concatenate strings

Compare strings

Find possible matches for a string

Set the union of two vectors

Set the unique elements of a vector

Structures

Structures are MATLAB arrays with named “data containers” called *fields*. The fields of a structure can contain any kind of data. For example, one field might contain a text string representing a name, another might contain a scalar representing a billing amount, a third might hold a matrix of medical test results, and so on.

```
patient  
.name 'John Doe' .billing 127.00
```

```
. test 79 75 73
```

```
180 178 177.5
```

```
220 210 205
```

Like standard arrays, structures are inherently array oriented. A single structure is a 1-by-1 structure array, just as the value 5 is a 1-by-1 numeric array. You can build structure arrays with any valid size or shape, including multidimensional structure arrays.

Note The examples in this section focus on two-dimensional structure arrays. For examples of higher-dimension structure arrays, see Chapter 19, “Multidimensional Arrays”

The following list summarizes the contents of this section:

- “Building Structure Arrays”
- “Accessing Data in Structure Arrays”
- “Finding the size of Structure Arrays”
- “Adding Fields to Structures”
- “Deleting Fields from Structures”
- “Applying Functions and Operators”
- “Writing Functions to Operate on Structures”
- “Organizing Data in Structure Arrays”
- “Nesting Structures”

Building Structure Arrays

You can build structures in two ways:

- Using assignment statements
- Using the `struct` function

Building Structure Arrays Using Assignment Statements

You can build a simple 1-by-1 structure array by assigning data to individual fields. MATLAB automatically builds the structure as you go along. For example, create the 1-by-1 `patient` structure array shown at the beginning of this section.

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

Now entering
`patient`
at the command line results in

```
name: 'John Doe'
billing: 127
test: [3x3 double]
```

`patient`
is an array containing a structure with three fields. To expand the structure array, add subscripts after the structure name.

```
patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68 70 68; 118 118 119; 172 170 169];
```

The patient structure array now has size[1 2]. Note that once a structure array contains more than a single element, MATLAB does not display individual field contents when you type the array name. Instead, it shows a summary of the kind of information the structure contains.

```
patient
patient =
```

1x2 struct array with fields:

```
name
billing
test
```

You can also use thefieldnames function to obtain this information. fieldnames returns a cell array of strings containing field names. As you expand the structure, MATLAB fills in unspecified fields with empty matrices so that:

- All structures in the array have the same number of fields.
- All fields have the same field names.

For example, entering patient(3).name = 'Alan Johnson' expands the patient array to size [1 3]. Now both patient(3).billing and patient(3).test contain empty matrices.

Note Field sizes do not have to conform for every element in an array. In the patient example, the name fields can have different lengths, the test fields can be arrays of different sizes, and so on.

Building Structure Arrays Using the struct Function

You can preallocate an array of structures with thestruct function. Its basic form is

```
str_array = struct('field1','val1','field2','val2', ...)
```

where the arguments are field names and their corresponding values. A field value can be a single value, represented by any MATLAB data construct, or a cell array of values. All field values in the argument list must be of the same scale (single value or cell array).

You can use different methods for preallocating structure arrays. These methods differ in the way in which the structure fields are initialized. As an example, consider the allocation of a 1-by-3 structure array, weather, with the structure fields temp and rainfall. Three different methods for allocating such an array are shown in this table.

Method Syntax Initialization

```
struct weather(3) =
struct('temp',72,'rainfall',0.0);
```

weather(3) is initialized with the field values shown. The fields for the other structures in the array, weather(1) and weather(2), are initialized to the empty matrix.

```
struct with weather =
repmat repmat(struct('temp',72, 'rainfall',0.0),1,3);
```

All structures in the weather array are initialized using one set of field values.

```
struct with weather = struct('temp',{68,80,72}, cell array 'rainfall',{0.2,0.4,0.0}); syntax  
The structures in theweather array are initialized with distinct field values specified with cell  
arrays.
```

Accessing Data in Structure Arrays

Using structure array indexing, you can access the value of any field or field element in a structure array. Likewise, you can assign a value to any field or field element. For the examples in this section, consider this structure array.

```
patient array  
patient(1) patient(2) patient(3)  
  
.name 'John Doe'  
.billing 127.00  
  
.test 79 75 73  
180 178 177.5  
220 210 205  
.name 'Ann Lane' .billing 28.50  
  
.test 68 70 68 118 118 119 172 170 169  
.name 'Al Smith'  
.billing 504.70  
  
.test 80 80 80  
153 153 154  
181 190 182
```

You can access subarrays by appending standard subscripts to a structure array name. For example, the line below results in a 1-by-2 structure array.

```
mypatients = patient(1:2)
```

```
1x2 struct array with fields: name  
billing  
test
```

The first structure in themypatients array is the same as the first structure in thepatient array.

```
mypatients(1)  
ans =
```

```
name: 'John Doe'  
billing: 127  
test: [3x3 double]
```

To access a field of a particular structure, include a period (.) after the structure name followed by the field name.

```
str = patient(2).name  
str =  
Ann Lane
```

To access elements within fields, append the appropriate indexing mechanism to the field name. That is, if the field contains an array, use array subscripting; if the field contains a cell array, use cell array subscripting, and so on.

```
test2b = patient(3).test(2,2)
```

```
test2b =
```

153 Use the same notations to assign values to structure fields, for example, patient(3).test(2,2) = 7;

You can extract field values for multiple structures at a time. For example, the line below creates a 1-by-3 vector containing all of the billing fields. bills = [patient.billing]

```
bills =
```

127.0000 28.5000 504.7000 Similarly, you can create a cell array containing the test data for the first two structures.

```
tests = {patient(1:2).test}
```

```
tests =
```

```
[3x3 double] [3x3 double]
```

Accessing Field Values Using setfield and getfield

Direct indexing is usually the most efficient way to assign or retrieve field values. If, however, you only know the field name as a string – for example, if you have used the fieldnames function to obtain the field name within an M-file – you can use the setfield and getfield functions to do the same thing. getfield obtains a value or values from a field or field element

```
f = getfield(array,{array_index},'field',{field_index}) where the field_index is optional,  
and array_index is optional for a 1-by-1 structure array. The function syntax corresponds to  
f = array(array_index).field(field_index);
```

For example, to access the name field in the second structure of the patient array, use:

```
str = getfield(patient,{2},'name');
```

Similarly, setfield lets you assign values to fields using the syntax f = setfield(array,{array_index},'field',{field_index},value)

Finding the size of Structure Arrays

Use the size function to obtain the size of a structure array, or of any structure field. Given a structure array name as an argument, size returns a vector of array dimensions. Given an argument in the form array(n).field, the size function returns a vector containing the size of the field contents.

For example, for the 1-by-3 structure array patient, size(patient) returns the vector [1 3]. The statement size(patient(1,2).name) returns the length of the name string for element(1,2) of patient.

Adding Fields to Structures

You can add a field to every structure in an array by adding the field to a single structure. For example, to add a social security number field to the patient array, use an assignment like

```
patient(2).ssn = '000-00-0000';
```

Now patient(2).ssn has the assigned value. Every other structure in the array also has the ssn field,

but these fields contain the empty matrix until you explicitly assign a value to them.

Deleting Fields from Structures

You can remove a given field from every structure within a structure array using `rmfield` function. Its most basic form is

```
struc2 = rmfield(array,'field')
```

where `array` is a structure array and `'field'` is the name of a field to remove from it. To remove the `name` field from the `patient` array, for example, enter: `patient = rmfield(patient,'name');`

Applying Functions and Operators

Operate on fields and field elements the same way you operate on any other MATLAB array. Use indexing to access the data on which to operate. For example, this statement finds the mean across the rows of the `test` array in `patient(2)`.

```
mean((patient(2).test)');
```

There are sometimes multiple ways to apply functions or operators across fields in a structure array. One way to add all the `billing` fields in the `patient` array is

```
total = 0;
for k = 1:length(patient)
    total = total + patient(k).billing;
end
```

To simplify operations like this, MATLAB enables you to operate on all like-named fields in a structure array. Simply enclose the `array.field` expression in square brackets within the function call. For example, you can sum all the `billing` fields in the `patient` array using

```
total = sum ([patient.billing]);
```

This is equivalent to using the comma-separated list.

```
total = sum ([patient(1).billing, patient(2).billing...]);
```

This syntax is most useful in cases where the operand field is a scalar field.

Writing Functions to Operate on Structures

You can write functions that work on structures with specific field architectures. Such functions can access structure fields and elements for processing.

Note When writing M-file functions to operate on structures, you must perform your own error checking. That is, you must ensure that the code checks for the expected fields.

As an example, consider a collection of data that describes measurements, at different times, of the levels of various toxins in a water source. The data consists of fifteen separate observations, where each observation contains three separate measurements.

You can organize this data into an array of 15 structures, where each structure has three fields, one for each of the three measurements taken. The function `concen`, shown below, operates on an

array of structures with specific characteristics. Its arguments must contain the fields lead, mercury, and chromium.

```
function [r1,r2] = concen(toxtest);
% Create two vectors. r1 contains the ratio of mercury to lead % at each observation. r2 contains
the ratio of lead to chromium. r1 = [toxtest.mercury]./[toxtest.lead];
r2 = [toxtest.lead]./[toxtest.chromium];
% Plot the concentrations of lead, mercury, and chromium % on the same plot, using different
colors for each. lead = [toxtest.lead];
mercury = [toxtest.mercury];
chromium = [toxtest.chromium];
plot(lead,'r'); hold on
plot(mercury,'b')
plot(chromium,'y'); hold off
```

Try this function with a sample structure array like test.

```
test(1).lead = .007; test(2).lead = .031; test(3).lead = .019; test(1).mercury = .0021;
test(2).mercury = .0009;
test(3).mercury = .0013;
test(1).chromium = .025; test(2).chromium = .017;
test(3).chromium = .10;
```

Organizing Data in Structure Arrays

The key to organizing structure arrays is to decide how you want to access subsets of the information. This, in turn, determines how you build the array that holds the structures, and how you break up the structure fields. For example, consider a 128-by-128 RGB image stored in three separate arrays; RED, GREEN, and BLUE.

Blueintensity

```
0.689 0.706 0.118 0.884 ...
0.535 0.532 0.653 0.925 ... values 0.314 0.265 0.159 0.101 ...
0.553 0.633 0.528 0.493 ...
0.441 0.465 0.512 0.512 ...
0.398 0.401 0.421 0.398 ... 0.342 0.647 0.515 0.816 ... 0.320 0.988 0.912 0.713 ... Greenintensity 0.111 0.300 0.205 0.526 ... 0.119
0.182 0.219 0.328 ...
```

values 0.523 0.428 0.712 0.929 ... 0.125 0.495 0.128 0.133 ...

```
0.214 0.604 0.918 0.344 ...
0.100 0.121 0.113 0.126 ...
0.288 0.187 0.204 0.175 ...
```

0.112 0.986 0.234 0.432 ... 0.208 0.576 0.760 0.531 ... Redintensity 0.765 0.128 0.863 0.521 ... 0.109 0.995 0.997 0.910 ...

values 1.000 0.985 0.761 0.698 ... 0.426 0.727 0.995 0.726 ...
0.455 0.783 0.224 0.395 ...
0.021 0.500 0.311 0.123 ...
1.000 1.000 0.867 0.051 ...
1.000 0.945 0.998 0.893 ...

```
0.990 0.941 1.000 0.876 ...
0.902 0.867 0.834 0.798 ...
.
```

There are at least two ways you can organize such data into a structure array.

Planeorganization Element-by-elementorganization

A B

```
0.112 0.986 0.234 0.432 ...
.r
```

```
0.765 0.128 0.863 0.521 ...
1.000 0.985 0.761 0.698 ...
```

· B(1,1) B(1,2) B(1,3)

...

· .r 0.112 .r 0.986 .r 0.234

```
0.342 0.647 0.515 0.816 ... .g 0.342 .g 0.647 .g 0.515 .g
0.111 0.300 0.205 0.526 ...
```

```
0.523 0.428 0.712 0.929 ...
· .b 0.689 .b 0.706 .b 0.118 .
```

· B(2,1) B(2,2) B(2,3)

...

```
0.689 0.706 0.118 0.884 ....r 0.765 .r 0.128 .r 0.863 .b
```

```
0.535 0.532 0.653 0.925 ...
0.314 0.265 0.159 0.101 ...
```

· .g 0.111 .g 0.300 .g 0.205 .

· .b 0.535 .b 0.532 .b 0.653

0.653

by-1structurearraywhereeachfieldisa128-by-128array ...

...

128-by-128structurearraywhereeachfieldisasingledataelement

Plane Organization

In case 1 above, each field of the structure is an entire plane of the image. You can create this structure using

```
A.r = RED;
A.g = GREEN;
A.b = BLUE;
```

This approach allows you to easily extract entire image planes for display, filtering, or other tasks that work on the entire image at once. To access the entire red plane, for example, use

```
red_plane = A.r;
```

Plane organization has the additional advantage of being extensible to multiple images in this case. If you have a number of images, you can store them as A(2), A(3), and so on, each containing an entire image. The disadvantage of plane organization is evident when you need to access subsets of the planes. To access a subimage, for example, you need to access each field separately.

```
red_sub = A.r(2:12,13:30);
grn_sub = A.g(2:12,13:30);
blue_sub = A.b(2:12,13:30);
```

Element-by-Element Organization

Case 2 has the advantage of allowing easy access to subsets of data. To set up the data in this organization, use

```
for m = 1:size(RED,1)
for n = 1:size(RED,2)
B(m,n).r = RED(m,n);
B(m,n).g = GREEN(m,n);
B(m,n).b = BLUE(m,n);

end
end
```

With element-by-element organization, you can access a subset of data with a single statement.
Bsub = B(1:10,1:10);

To access an entire plane of the image using the element-by-element method, however, requires a loop.

```
red_plane = zeros(128,128);
for k = 1:(128°F128)
red_plane(k) = B(k).r;
end
```

Element-by-element organization is not the best structure array choice for most image processing applications; however, it can be the best for other applications wherein you will routinely need to access corresponding subsets of structure fields. The example in the following section demonstrates this type of application.

Example - A Simple Database Consider organizing a simple database.

A¹ Planeorganization^B Element-by-elementorganization
'Ann Jones' ...

```
.
```

nam
e

'Dan Smith' ... 'Kim Lee' ...

.

.

.

B(1) B(2) B(3)

```

...
.name 'Ann Jones' .name 'Dan Smith' .name 'Kim Lee'
'80 Park St.' ...
.address
'5 Lake Ave.' ... '116 Elm St.' ...
.address '80 Park St.' .address '5 Lake Ave.'.address'116 Elm St.'
12.50 ...
.
amount
81.29 ...
30.00 ...
.
.
.amount12.50 .amount81.29 .amount
30.00

B(1).name = 'Ann Jones'; B(1).address = '80 Park St.'; B(1).amount = 12.5;
A.name = strvcat('Ann Jones','Dan Smith',...); B(2).name = 'Dan Smith';
A.address = strvcat('80 Park St.', '5 Lake Ave.', B(2).address = '5 Lake Ave.';
..);
A.amount = [12.5;81.29;30; ...];
B(2).amount = 81.29;
.
.
.
```

Each of the possible organizations has advantages depending on how you want to access the data:

- Plane organization makes it easier to operate on all field values at once. For example, to find the average of all the values in the amount field,
 - Using plane organization

```
avg = mean(A.amount);
– Using element-by-element organization
avg = mean([B.amount]);
```

- Element-by-element organization makes it easier to access all the information related to a single client. Consider an M-file, client.m, which displays the name and address of a given client on screen.

Using plane organization, pass individual fields.

```
function client(name,address)
disp(name)
```

```
disp(address)
```

To call the client function,

```
client(A.name(2,:),A.address(2,:))
```

Using element-by-element organization, pass an entire structure.

```
function client(B)
```

```
    disp(B)
```

To call the client function,

```
client(B(2))
```

- Element-by-element organization makes it easier to expand the string array fields. If you do not know the maximum string length ahead of time for plane organization, you may need to frequently recreate the name or address field to accommodate longer strings.

Typically, your data does not dictate the organization scheme you choose. Rather, you must consider how you want to access and operate on the data.

Nesting Structures

A structure field can contain another structure, or even an array of structures. Once you have created a structure, you can use the struct function or direct assignment statements to nest structures within existing structure fields.

Building Nested Structures with the struct Function

To build nested structures, you can nest calls to the struct function. For example, create a 1-by-1 structure array.

```
A = struct('data',[3 4 7; 8 0 1],'nest',...
struct('testnum','Test 1', 'xdata',[4 2 8],... 'ydata',[7 1 6]));
```

You can build nested structure arrays using direct assignment statements. These statements add a second element to the array.

```
A(2).data = [9 3 2; 7 6 5];
A(2).nest.testnum = 'Test 2';
A(2).nest.xdata = [3 4 2];
A(2).nest.ydata = [5 0 9];
```

```
A
A(1) A(2)
```

```
.data
3 4 7 .data 9 3 2
8 0 1 7 6 5
.nest .testnum 'Test 1' .nest .testnum 'Test 2' .xdata [4 2 8] .xdata [3 4 2] .ydata [7 1 6] .ydata [5 0 9]
```

Indexing Nested Structures

To index nested structures, append nested field names using dot notation. The first text string in the indexing expression identifies the structure array, and subsequent expressions access field names that contain other structures.

For example, the arrayA created earlier has three levels of nesting:

- To access the nested structure insideA(1), useA(1).nest.
- To access the xdata field in the nested structure inA(2), use A(2).nest.xdata.
- To access element 2 of the ydata field inA(1), useA(1).nest.ydata(2).

Cell Arrays

A *cell array* is a MATLAB array for which the elements are *cells*, containers that can hold other MATLAB arrays. For example, one cell of a cell array might contain a real matrix, another an array of text strings, and another a vector of complex values.

cell 1,1 cell 1,2 cell 1,3

'Anne Smith'

3 4 2 '9/12/94' .25+3i 8-16i

9 7 6 'Class II'

8 5 1 'Obs. 1' 34+5i 7+.92i

'Obs. 2'

cell 2,1 cell 2,2 cell 2,3

7

2 14

'text'

4 2

1 5

[1.43 2.98 8 3 45

5.67] 52 16 3

[4 2 7] .02 + 8i

You can build cell arrays of any valid size or shape, including multidimensional structure arrays.

Note The examples in this section focus on two-dimensional cell arrays. For examples of higher-dimension cell arrays, see Chapter 19, “Multidimensional Arrays”.

The following list summarizes the contents of this section:

- “Creating Cell Arrays”
- “Obtaining Data from Cell Arrays”
- “Deleting Cells”
- “Reshaping Cell Arrays”
- “Replacing Lists of Variables with Cell Arrays”
- “Applying Functions and Operators”
- “Organizing Data in Cell Arrays”
- “Nesting Cell Arrays”

- “Converting Between Cell and Numeric Arrays”
- “Cell Arrays of Structures”

Creating Cell Arrays

You can create cell arrays by:

- Using assignment statements
- Preallocating the array using the `cell` function, then assigning data to cells

Using Assignment Statements

You can build a cell array by assigning data to individual cells, one cell at a time. MATLAB automatically builds the array as you go along. There are two ways to assign data to cells:

- Cell indexing

Enclose the cell subscripts in parentheses using standard array notation. Enclose the cell contents on the right side of the assignment statement in curly braces, “{}.” For example, create a 2-by-2 cell array `A`.

```
A(1,1) = {[1 4 3; 0 5 8; 7 2 9]};
A(1,2) = {'Anne Smith'};
A(2,1) = {3+7i};
A(2,2) = {-pi:pi/10:pi};
```

Note The notation “{}” denotes the empty cell array, just as “[]” denotes the empty matrix for numeric arrays. You can use the empty cell array in any cell array assignments.

- Content indexing

Enclose the cell subscripts in curly braces using standard array notation. Specify the cell contents on the right side of the assignment statement. `A{1,1} = [1 4 3; 0 5 8; 7 2 9];`

```
A{1,2} = 'Anne Smith';
A{2,1} = 3+7i;
A{2,2} = -pi:pi/10:pi;
```

The various examples in this guide do not use one syntax throughout, but attempt to show representative usage of cell and content addressing. You can use the two forms interchangeably.

Note If you already have a numeric array of a given name, don’t try to create a cell array of the same name by assignment without first clearing the numeric array. If you do not clear the numeric array, MATLAB assumes that you are trying to “mix” cell and numeric syntaxes, and generates an error. Similarly, MATLAB does not clear a cell array when you make a single assignment to it. If any of the examples in this section give unexpected results, clear the cell array from the workspace and try again.

MATLAB displays the cell array `A` in a condensed form. `A = [3x3 double] 'Anne Smith' [3.0000+ 7.0000i] [1x21 double]` To display the full cell contents, use the `celldisp` function. For a high-level graphical display of cell architecture, use `cellplot`.

If you assign data to a cell that is outside the dimensions of the current array, MATLAB automatically expands the array to include the subscripts you specify. It fills any intervening

cells with empty matrices. For example, the assignment below turns the 2-by-2 cell arrayA into a 3-by-3 cell array.

```
A(3,3) = {5};  
cell 1,1 cell 1,2 cell 1,3
```

```
1 4 3  
0 5 8 'Anne Smith' [ ]  
7 2 9
```

```
cell 2,1 cell 2,2 cell 2,3
```

```
3+7i [-3.14...3.14] [ ]
```

```
cell 3,1 cell 3,2 cell 3,3
```

```
[ ] [ ] 5
```

Cell Array Syntax: Using Braces

The curly braces, “{}”, are cell array constructors, just as square brackets are numeric array constructors. Curly braces behave similarly to square brackets, except that you can nest curly braces to denote nesting of cells (see “Nesting Cell Arrays” for details).

Curly braces use commas or spaces to indicate column breaks and semicolons to indicate row breaks between cells. For example,

```
C = {[1 2], [3 4]; [5 6], [7 8]};
```

results in

```
cell 1,1 cell 1,2
```

```
[1 2] [3 4]
```

```
cell 2,1 cell 2,2
```

```
[5 6] [7 8]
```

Use square brackets to concatenate cell arrays, just as you do for numeric arrays.

Preallocating Cell Arrays with the cell Function

The `cell` function allows you to preallocate empty cell arrays of the specified size. For example, this statement creates an empty 2-by-3 cell array.

```
B = cell(2,3);
```

Use assignment statements to fill the cells of B.

```
B(1,3) = {1:3};
```

Obtaining Data from Cell Arrays

You can obtain data from cell arrays and store the result as either a standard array or a new cell array. This section discusses:

- Accessing cell contents using content indexing
- Accessing a subset of cells using cell indexing

Accessing Cell Contents Using Content Indexing

You can use content indexing on the right side of an assignment to access some or all of the data in a single cell. Specify the variable to receive the cell contents on the left side of the assignment. Enclose the cell index expression on the right side of the assignment in curly braces. This indicates that you are assigning cell contents, not the cells themselves.

Consider the 2-by-2 cell array N.

```
N{1,1} = [1 2; 4 5];
N{1,2} = 'Name';
N{2,1} = 2-4i;
N{2,2} = 7;
```

You can obtain the string in N{1,2} using

```
c = N{1,2}
c =
Name
```

Note In assignments, you can use content indexing to access only a single cell, not a subset of cells. For example, the statements `A{1,:} = value` and `B = A{1,:}` are both invalid. However, you can use a subset of cells any place you would normally use a comma-separated list of variables (for example, as function inputs or when building an array). See the “Replacing Lists of Variables with Cell Arrays” section for details.

To obtain subsets of a cell’s contents, concatenate indexing expressions. For example, to obtain element(2,2) of the array in cell N{1,1}, use: `d = N{1,1}(2,2)`

```
d =
5
```

Accessing a Subset of Cells Using Cell Indexing

Use cell indexing to assign any set of cells to another variable, creating a new cell array. Use the colon operator to access subsets of cells within a cell array.

```
cell 1,1 cell 1,2 cell 1,3
3 5 9 cell 1,1 cell 1,2
cell 2,1 cell 2,2 cell 2,3 6 0
```

```
5 6 0 B = A(2:3,2:3)
```

```
cell 2,1 cell 2,2
7 2
```

```
cell 3,1 cell 3,2 cell 3,3
4 7 2
```

Deleting Cells

You can delete an entire dimension of cells using a single statement. Like standard array deletion, use vector subscripting when deleting a row or column of cells and assign the empty matrix to the dimension.

```
A(cell_subscripts) = []
```

When deleting cells, curly braces do not appear in the assignment statement at all.

Reshaping Cell Arrays

Like other arrays, you can reshape cell arrays using the `reshape` function. The number of cells must remain the same after reshaping; you cannot use `reshape` to add or remove cells.

```
A = cell(3,4);
size(A)
ans =
3 4
B = reshape(A,6,2); size(B)
ans =
6 2
```

Replacing Lists of Variables with Cell Arrays

Cell arrays can replace comma-separated lists of MATLAB variables in:

- Function input lists
- Function output lists
- Display operations
- Array constructions (square brackets and curly braces)

If you use the colon to index multiple cells in conjunction with the curly brace notation, MATLAB treats the contents of each cell as a separate variable. For example, assume you have a cell array T where each cell contains a separate vector. The expression T{1:5} is equivalent to a comma-separated list of the vectors in the first five cells of T.

Consider the cell array C.

```
C(1) = {[1 2 3]};
C(2) = {[1 0 1]};
C(3) = {1:10};
C(4) = {[9 8 7]}; C(5) = {3};
```

To convolve the vectors in C(1) and C(2) using conv, d = conv(C{1:2})

```
d =
1 2 4 2 3
Display vectors two, three, and four with C{2:4}
ans =
1 0 1
ans =
1 2 3 4 5 6 7 8 9 10
ans =
9 8 7
```

Similarly, you can create a new numeric array using the statement

```
B = [C{1}; C{2}; C{4}]
B =
```

```
1 2 3
1 0 1
9 8 7
```

You can also use content indexing on the left side of an assignment to create a new cell array where each cell represents a separate output argument. [D{1:2}] = eig(B)

```
D =
[3x3 double] [3x3 double]
You can display the actual eigenvalues and eigenvectors using D{1}
```

andD{2}.

Note The varargin and varargin arguments allow you to specify variable numbers of input and output arguments for MATLAB functions that you create. Both varargin and varargin are cell arrays, allowing them to hold various sizes and kinds of MATLAB data. See “Passing Variable Numbers of Arguments” on page 17-16 for details.

Applying Functions and Operators

Use indexing to apply functions and operators to the contents of cells. For example, use content indexing to call a function with the contents of a single cell as an argument.

```
A{1,1} = [1 2; 3 4];
A{1,2} = randn(3,3);
A{1,3} = 1:5;
B = sum(A{1,1})
```

```
B =
4 6
```

To apply a function to several cells of a non-nested cell array, use a loop. for k = 1:length(A)
M{k} = sum(A{1,k});
end

Organizing Data in Cell Arrays

Cell arrays are useful for organizing data that consists of different sizes or kinds of data. Cell arrays are better than structures for applications where:

- You need to access multiple fields of data with one statement.
- You want to access subsets of the data as comma-separated variable lists.
- You don't have a fixed set of field names.
- You routinely remove fields from the structure.

As an example of accessing multiple fields with one statement, assume that your data consists of:

- A 3-by-4 array consisting of measurements taken for an experiment.
- A 15-character string containing a technician's name.
- A 3-by-4-by-5 array containing a record of measurements taken for the past five experiments.

For many applications, the best data construct for this data is a structure. However, if you routinely access only the first two fields of information, then a cell array might be more convenient for indexing purposes.

This example shows how to access the first and second elements of the cell array TEST.
[newdata,name] = deal(TEST{1:2})

This example shows how to access the first and second elements of the structure TEST.
newdata = TEST.measure
name = TEST.name

The varargin and varargin arguments are examples of the utility of cell arrays as substitutes for

comma-separated lists. Create a 3-by-3 numeric arrayA.

```
A = [0 1 2;4 0 7;3 1 2];
```

Now apply thenormest(2-norm estimate) function toA, and assign the function output to individual cells ofB.

```
[B{1:2}] = normest(A)
```

```
B =
```

```
[8.8826] [4]
```

All of the output values from the function are stored in separate cells ofB.B(1) contains the norm estimate;B(2) contains the iteration count.

N esting Cell Arra ys

A cell can contain another cell array, or even an array of cell arrays. (Cells that contain noncell data are called *leaf cells*.) You can use nested curly braces, the cell function, or direct assignment statements to create nested cell arrays. You can then access and manipulate individual cells, subarrays of cells, or cell elements.

Building Nested Arrays with Nested Curly Braces

You can nest pairs of curly braces to create a nested cell array. For example,

```
clear A
```

```
A(1,1) = {magic(5)};
```

```
A(1,2) = {{[5 2 8; 7 3 0; 6 7 3] 'Test 1'; [2-4i 5+7i] {17 []}}}
```

```
A =
```

```
[5x5 double] {2x2 cell}
```

Note that the right side of the assignment is enclosed in two sets of curly braces. The first set represents cell(1,2) of cell arrayA. The second “packages” the 2-by-2 cell array inside the outer cell.

Building Nested Arrays with the cell Function

To nest cell arrays with thecell function, assign the output ofcell to an existing cell:

1 Create an empty 1-by-2 cell array. A = cell(1,2);

2 Create a 2-by-2 cell array insideA(1,2). A(1,2) = {cell(2,2)};

3 FillA, including the nested array, using assignments.

```
A(1,1) = {magic(5)};
```

```
A{1,2}{1,1} = {[5 2 8; 7 3 0; 6 7 3]}; A{1,2}{1,2} = {'Test 1'}; A{1,2}{2,1} = {[2-4i 5+7i]};
```

```
A{1,2}{2,2} = {cell(2,2)} A{1,2}{2,2}{1} = {17};
```

Note the use of curly braces until the final level of nested subscripts. This is required because you need to access cell contents to access cells within cells. You can also build nested cell arrays with direct assignments using the statements shown in step 3 above.

Indexing Nested Cell Arrays

To index nested cells, concatenate indexing expressions. The first set of subscripts accesses the top layer of cells, and subsequent sets of parentheses access successively deeper layers.

For example, arrayA has three levels of nesting:

cell 1,1 cell 1,2

```
17 24 1 8 15 5 2 8  
23 5 7 14 16 7 3 0 'Test 1' 4 6 13 20 22 6 7 3  
10 12 19 21 3  
11 18 25 2 9  
[2-4i 5+7i] 17
```

- To access the 5-by-5 array in cell(1,1), use `A{1,1}`.
- To access the 3-by-3 array in position(1,1) of cell(1,2), use `A{1,2}{1,1}`.
- To access the 2-by-2 cell array in cell(1,2), use `A{1,2}`.
- To access the empty cell in position(2,2) of cell(1,2), use

`A{1,2}{2,2}{1,2}`.

Converting Between Cell and Numeric Arrays

Use `for` loops to convert between cell and numeric formats. For example, create a cell array `F`.

```
F{1,1} = [1 2; 3 4];  
F{1,2} = [-1 0; 0 1];  
F{2,1} = [7 8; 4 1];  
F{2,2} = [4i 3+2i; 1-8i 5];
```

Now use three `for` loops to copy the contents of `F` into a numeric array `NUM`. for $k = 1:4$

```
for m = 1:2  
for n = 1:2  
NUM(m,n,k) = F{k}(m,n);  
end  
end  
end
```

Similarly, you must use `for` loops to assign each value of a numeric array to a single cell of a cell array.

```
G = cell(1,16);  
for m = 1:16  
G{m} = NUM(m);  
end
```

Cell Arrays of Structures

Use cell arrays to store groups of structures with different field architectures.

```
c_str = cell(1,2);  
c_str{1}.label = '12/2/94 - 12/5/94';  
c_str{1}.obs = [47 52 55 48; 17 22 35 11];  
c_str{2}.xdata = [-0.03 0.41 1.98 2.12 17.11];  
c_str{2}.ydata = [-3 5 18 0 9];  
c_str{2}.zdata = [0.6 0.8 1 2.2 3.4];
```

cell 1 cell 2

`c_str(1)c_str(2)`

`.label '12/2/94 - 12/5/94' .name [-0.03 0.41 1.98 2.12 17.11] .test 47 52 55 48 .billing [-3 5 18 0 9] .test 17 22 35 11 .test [0.6 0.8 1 2.2`

3.4]

Cell 1 of the c_str array contains a structure with two fields, one a string and the other a vector. Cell 2 contains a structure with three vector fields. When building cell arrays of structures, you must use content indexing. Similarly, you must use content indexing to obtain the contents of structures within cells. The syntax for content indexing is:

cell_array{index}.field

For example, to access the label field of the structure in cell 1, use c_str{1}.label.

21

Function Handles

Benefits of Using Function Handles	21-3
Handle.....	21-5
Constructing a Function Handle.....	21-7
Maximum Length of a Function Name	21-7
Evaluating a Function Through Its Handle	21-9
Function Evaluation and Overloading	21-9
Examples of Function Handle Evaluation.....	21-10
Displaying Function Handle Information	21-13
Fields Returned by the Functions Command.....	21-14
Types of Function Handles	21-17
Function Handle Operations.....	21-21
Converting Function Handles to Function Names.....	21-21
Converting Function Names to Function Handles.....	21-22
Testing for Data Type.....	21-23
Testing for Equality	21-23
Saving and Loading Function Handles.....	21-25
Handling Error Conditions	21-26
Handles to Nonexistent Functions.....	21-26
Including Path In the Function Handle Constructor	21-26
Evaluating a Nonscalar Function Handle.....	21-27
Historical Note - Evaluating Function Names	21-28

A *function handle* is a MATLAB data type that contains information used in referencing a function. When you create a function handle, MATLAB stores in the handle all the information about the function that it needs to execute, or *evaluate*, it later on. Typically, a function handle is passed in an argument list to other functions. It is then used in conjunction with feval to evaluate the function to which the handle belongs.

A MATLAB function handle is more than just a reference to a function. It often represents a

collection of function methods, overloaded to handle different argument types. When you create a handle to a function, MATLAB takes a snapshot of all built-in and M-file methods of that name that are on the MATLAB path and in scope at that time, and stores access information for all of those methods in the handle.

When you evaluate a function handle, MATLAB considers only those functions that were stored within the handle when it was created. Other functions that might now be on the path or in scope are not considered. It is the combination of which function methods are mapped to by the handle and what arguments the handle is evaluated with that determines which is the actual function that MATLAB dispatches to.

This chapter addresses the following topics:

- “Benefits of Using Function Handles”
- “Constructing a Function Handle”
- “Evaluating a Function Through Its Handle”
- “Displaying Function Handle Information”
- “Function Handle Operations”
- “Saving and Loading Function Handles”
- “Handling Error Conditions”
- “Historical Note - Evaluating Function Names”

Benefits of Using Function Handles

Function handles enable you to do all of the following:

- Pass function access information to other functions
- Capture all methods of an overloaded function
- Allow wider access to subfunctions and private functions
- Ensure reliability when evaluating functions
- Reduce the number of files that define your functions
- Improve performance in repeated operations
- Manipulate handles in arrays, structures, and cell arrays

This section also includes an example of using a simple function handle. See “A Simple Function Handle” on page 21-5.

Pass Function Access Information to Other Functions

You can pass a function handle as an argument in a call to another function. The handle contains access information that enables the receiving function to call the function for which the handle was constructed.

You can evaluate a function handle from within another function even if the handle’s function is not in the scope of the evaluating function. This is because the function performing the evaluation has all the information it needs within the function handle.

For the same reason, you can also evaluate a function handle even when the handle’s function is no longer on the MATLAB search path.

You must use the MATLAB feval command to evaluate the function in a function handle. When you pass a function handle as an argument into another function, then the function receiving the handle usesfeval to evaluate the function handle.

Capture All Methods of An Overloaded Function

Because many MATLAB functions are overloaded, a function handle often maps to a number of code sources (e.g., built-in code, M-files), that implement the function. A function handle stores the access to all of the overloaded sources, or *methods*, that are on the MATLAB path at the time the handle is created.

When you evaluate an overloaded function handle, MATLAB follows the usual rules of selecting which method to evaluate, basing the selection on the argument types passed in the function call. See “How MATLAB Determines Which Method to Call” on page 22-67, for more information on how MATLAB selects overloaded functions.

For example, there are three built-in functions and one M-file function that define the abs function on the standard MATLAB path. A function handle created for theabs function contains access information on all four of these function sources. If you evaluate the function with an argument of thedouble type, then the built-in function that takes adouble argument is executed.

Allow Wider Access to Subfunctions and Private Functions

By definition, all MATLAB functions have a certain scope. They are visible to other MATLAB entities within that scope, but not visible outside of it. You can invoke a function directly from another function that is within its scope, but not from a function outside that scope.

Subfunctions and private functions are, by design, limited in their visibility to other MATLAB functions. You can invoke a subfunction only by another function that is defined within the same M-file. You can invoke a private function only from a function in the directory immediately above the\private subdirectory.

When you create a handle to a function that has limited scope, the function handle stores all the information MATLAB needs to evaluate the function from any location in the MATLAB environment. If you create a handle to a subfunction while the subfunction is in scope, (that is, you create it from within the M-file that defines the subfunction), you can then pass the handle to code that resides outside of that M-file and evaluate the subfunction from beyond its usual scope. The similar case holds true for private functions.

Ensure Reliability When Evaluating Functions

Function handles allow you more control over what methods get executed when a function is evaluated. If you create a function handle for a function with overloaded methods, making sure that only the intended methods are within scope when the handle is created gives you control over which methods are executed when MATLAB evaluates the handle. This can isolate you from methods that might be in scope at the time of evaluation that you didn’t know about.

Reduce the Number of Files That Define Your Functions

You can use function handles to help reduce the number of M-files required to define your

functions. The problem with grouping a number of functions in one M-file is that this defines them as subfunctions, and thus reduces their scope in MATLAB. Using function handles to access these subfunctions removes this limitation. This enables you to group functions as you want and reduce the number of files you have to manage.

Improve Performance in Repeated Operations

MATLAB performs a lookup on a function at the time you create a function handle and then stores this access information in the handle itself. Once defined, you can use this handle in repeated evaluations without incurring the performance delay associated with function lookup each time.

Manipulate Handles in Arrays, Structures, and Cell Arrays As a standard MATLAB data type, a function handle can be manipulated and operated on in the same manner as other MATLAB data types. You can create arrays, structures, or cell arrays of function handles. Access individual function handles within these data structures in the same way that you access elements of a numeric array or structure.

Create n-dimensional arrays of handles using either of the concatenation methods used to form other types of MATLAB arrays, [] or cat. All operations involving matrix manipulation are supported for function handles.

A Simple Function Handle

The repmat function is an elementary matrix function, which is defined in MATLAB with the single M-file, repmat.m. If you create a function handle to the repmat function, MATLAB stores in the handle the information it will need later to evaluate the function. Included in this information is the repmat.m source file path, toolbox\matlab\elmat\repmat.m.

Once you create a function handle, it is not affected by certain changes you might make in your MATLAB environment. For example, if you construct a handle to the repmat function and, later, write additional repmat.m methods to overload the function, the handle still sees only the original function. Also, if you remove the path to repmat.m from the search path, MATLAB is still able to locate and evaluate the function using the handle that you created prior to the path change. Since repmat is not an overloaded function in this case, evaluation of the function through its handle is fairly simple. You call feval on the function handle, also passing any arguments the function should act upon. MATLAB executes the one function whose access information is stored in the handle. Constructing a Function Handle

Constructing a Function Handle

Construct a function handle in MATLAB using the *at* sign, @, before the function name. The following example creates a function handle for thehumps function and assigns it to the variable fhandle.

```
fhandle = @humps;
```

Pass the handle to another function in the same way you would pass any argument. This example passes the function handle just created to fminbnd, which then minimizes over the interval [0.3, 1].

```
x = fminbnd(fhandle, 0.3, 1)
x =
0.6370
```

The fminbnd function evaluates the @humps function handle using feval. A small portion of the fminbnd M-file is shown below. In line 1, the funfcn input parameter receives the function handle, @humps, that was passed in. The feval statement, in line 113, evaluates the handle.

```
1 function [xf,fval,exitflag,output] = ...
fminbnd(funfcn,ax,bx,options,varargin)
.
.
.
113 fx = feval(funfcn,x,varargin{:});
```

Note When creating a function handle, you may only use the function *name* after the @ sign. This must not include any path information. The following syntax is invalid: fhandle = @\home\user4\humps.

M a ximum Length of a Function N a me

Function names used in handles are unique up to 31 characters. If the function name exceeds that length, MATLAB truncates the latter part of the name. fhandle = @function_name_that_exceeds_thirty_one_characters fhandle = @function_name_that_exceeds_thir

For function handles created for Java constructors, the length of any segment of the package name or class name must not exceed 31 characters. (The term *segment* refers to any portion of the name that lies before, between, or after a dot. For example, there are three segments in java.lang.String.) There is no limit to the overall length of the string specifying the package and class.

The following statement is valid, even though the length of the overall package and class specifier exceeds 31 characters.

```
fhandle = @java.awt.datatransfer.StringSelection
```

Evaluating a Function Through Its Handle

Execute the target function of a function handle using the MATLAB feval command. The syntax for using this command with a function handle is feval(fhandle, arg1, arg2, ..., argn)

This acts similarly to a direct call to the function represented by fhandle, passing arguments arg1 through argn. The principal differences are:

- A function handle can be evaluated from within any function that you pass it to.
- The code source that MATLAB selects for evaluation depends upon which overloaded methods of the function were on the MATLAB path and in scope at the time the handle was constructed. (Argument types also affect method selection.) Path and scope are not considered at the time of evaluation.
- MATLAB does the work of initial function lookup at the time the function handle is

constructed. This does not need to be done each time MATLAB evaluates the handle.

Note The feval command does not operate on nonscalar function handles. Passing a nonscalar function handle to feval results in an error.

Function Evaluation and Overloading

To understand the relationship between function handles and overloading, it is helpful to review, briefly, the nature of MATLAB function calls. Because of overloading, it is useful to think of a single MATLAB function as comprising a number of code sources (for example, built-in code, M-files). When you call a MATLAB function without feval, the choice of which source is called depends upon two factors:

- The methods that are visible on the path at the time of the call
- The classes of the arguments to the function

MATLAB evaluates function handles in a similar manner. In most cases, a function handle represents a collection of methods that overload the function. When you evaluate a function handle using feval, the choice of the particular method called depends on:

- The methods that were visible on the path at the time the handle was constructed
- The classes of the arguments passed with the handle to the feval command

Examples of Function Handle Evaluation

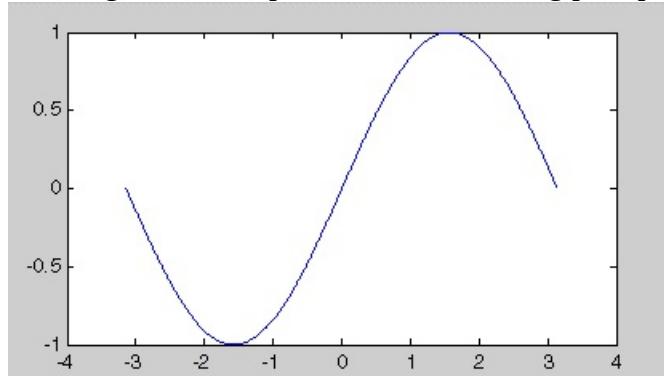
This section provides two examples of how function handles are used and evaluated.

Example 1 - A Simple Function Handle

The following example defines a function, called plot_fhandle, that receives a function handle and data, and then performs an evaluation of the function handle on that data.

```
function x = plot_fhandle(fhandle, data)
plot(data, feval(fhandle, data))
```

When you call plot_fhandle with a handle to the sin function and the argument shown below, the resulting evaluation produces the following plot. plot_fhandle(@sin, -pi:0.01:pi)



Example 2 - Function Handles and Subfunctions

The M-file in this example defines a primary function, fitcurvedemo, and a subfunction called expfun. The subfunction, by definition, is visible only within the scope of its own M-file. This, of course, means that it is available for use only by other functions within that M-file.

The author of this code would like to use expfun outside the confines of this one M-file. This example creates a function handle to theexpfun subfunction, storing access information for the subfunction so that it can be called from anywhere in the MATLAB environment. The function handle is passed to fminsearch, which successfully evaluates the subfunction outside of its usual scope.

The code shown below defines fitcurvedemo and subfunction,expfun. Line 6 constructs a function handle to expfun and assigns it to the variable,fun. In line 16, a call tofminsearch passes the function handle outside the normal scope of a subfunction. The fminsearch function usesfeval to evaluate the subfunction through its handle.

```

1 function Estimates = fitcurvedemo
2 % FITCURVEDEMO
3 % Fit curve to data where user chooses equation to fit.
4
5 % Define function and starting point of fitting routine.
6 fun = @expfun;
7 Starting = rand(1, 2);
8
9 % First, we create the data.
10 t = 0:.1:10; t=t(:); % to make 't' a column vector
11 Data = 40 * exp(-.5 * t) + randn(size(t));
12 m = [t Data];
13
14 % Now, we can call FMINSEARCH:
15 options = optimset('fminsearch'); % Use FMINSEARCH defaults
16 Estimates = fminsearch(fun, Starting, options, t, Data);
17
18 % To check the fit
19 plot(t, Data, '*')
20 hold on
21 plot(t, Estimates(1) * exp(-Estimates(2) * t), 'r')
22 xlabel('t')
23 ylabel('f(t)')
24 title(['Fitting to function ', func2str(fun)]);
25 legend('data', ['fit using ', func2str(fun)])
26 hold off
27
28 %
29
30 function sse = expfun(params, t, Data)
31 % Accepts curve parameters as inputs, and outputs fitting the
32 % error for the equation y = A * exp(-lambda * t);
33 A = params(1);
34 lambda = params(2);
35
36 Fitted_Curve = A .* exp(-lambda * t);

```

```

37 Error_Vector = Fitted_Curve - Data;
38
39 % When curve fitting, a typical quantity to minimize is the sum
40 % of squares error
41 sse = sum(Error_Vector .^ 2);

```

Displaying Function Handle Information

The functions command returns information about a function handle that might be useful for debugging. Calling functions on a function handle returns the function name, type, filename, and all of the methods for the function that were in scope at the time the function handle was created.

The information returned from functions is in the form of a MATLAB structure. The fields of this structure are listed in the following table.

Field Name Field Description

function Function name. type

file

methods

Function type. See the table in section, “Function Type” on page 21-14.

The file to be executed when the function handle is evaluated with a nonoverloaded data type. For built-in functions, it reads ‘MATLAB built-in function.’

All overloaded methods of the function that are bound to the function handle. This field exists only for functions of type overloaded.

For example, to obtain information on a function handle for the display function,

```

f = functions(@display)
ans =
function: 'display'
type: 'overloaded'
file: 'MATLAB built-in function'
methods: [1x1 struct]

```

Individual fields of the structure are accessible using the dot selection notation used to access MATLAB structure fields.

```

f.type
ans =
'overloaded'

```

The methods field is a separate structure containing one fieldname for each class that overloads the function. The value of each field is the path and name of the file that defines the function.

```

f.methods
ans =
polynom: '\home\user4\@polynom\display.m'

inline: 'matlabroot\toolbox\matlab\funfun\@inline\display.m' serial:
'matlabroot\toolbox\matlab\iofun\@serial\display.m'

```

```
avifile: 'matlabroot\toolbox\matlab\iofun \@avifile\display.m'
```

```
f.methods.polynom
```

```
ans =
```

```
polynom: '\home\user4\@polynom\display.m'
```

Note The functions command does not operate on nonscalar function handles. Passing a nonscalar function handle to functions results in an error.

Fields Returned by the Functions Comm and

The functions command returns a MATLAB structure with the fields function, type, file, and, for overloaded functions, methods. This section describes each of those fields.

Function Name

The function field is a character array that holds the name of the function corresponding to the function handle.

Function Type

The type field is a character array that holds one of five possible strings listed in the following table.

Function Type Type Description

simple Nonoverloaded MATLAB built-in or M-file, or any function for which the type cannot be determined until it is evaluated

subfunction MATLAB subfunction private MATLAB private function constructor Constructor to a MATLAB class overloaded Overloaded built-in, M-file function, or constructor

The contents of the next two fields, file and methods, depend upon the function type.

Function File

The file field is a character array that holds one of the following:

- The string, 'MATLAB built-in function', for built-in functions
- The path and name of the file that implements the default function, for nonbuilt-in functions

The *default* function is the one function implementation that is not specialized to operate on any particular data type. Unless the arguments in the function call specify a class that has a specialized version of the function defined, it is the default function that gets called.

The example below operates on a function handle for the deblank function. The function has a default implementation in the strfun directory. This is shown inf.file. It also has an overloaded method in the @cell directory. This is shown inf.methods.cell.

```
f = functions(@deblank)
```

```
f =
```

```
function: 'deblank'
```

```
type: 'overloaded'
```

```
file: 'matlabroot\toolbox\matlab\strfun\deblank.m' methods: [1x1 struct] f.methods
```

```
ans =
```

```
cell: 'matlabroot\toolbox\matlab\strfun \@cell\deblank.m'
```

If you evaluate the @deblank function handle with a cell argument, MATLAB calls the deblank method in the @cell directory. But, for any other argument types, MATLAB calls the default M-

file shown in the file field.

Function Methods

The methods field exists only for functions of type overloaded. This field is a separate MATLAB structure that identifies all overloaded methods that are bound to the function handle.

The structure contains one field for each method of the function handle. The field names are the classes that overload the function. Each field value is a character array holding the path and name of the source file that defines the method.

For example, a function handle for the display function may be bound to the M-files shown below. The functions command returns a methods structure having a field for each class overloading the function: polynom, inline, serial, and avifile. For each class, it shows the path and name of the method source file.

```
f = functions(@display);
f.methods
ans =
polynom: 'home\user4\@polynom\display.m'

inline: 'matlabroot\toolbox\matlab\funfun\@inline\display.m' serial:
'matlabroot\toolbox\matlab\iofun\@serial\display.m'
avifile: 'matlabroot\toolbox\matlab\iofun\@avifile\display.m'
```

Note The set of methods returned by functions is determined at the time a function handle is created. This depends on the state of the MATLAB path and also which functions are in scope when the handle is created.

Types of Function Handles

The information returned by functions varies depending on the type of function represented by the function handle. This section explains what is returned for each type of function. The categories of function handles are:

- Simple function handles
- Overloaded function handles
- Constructor function handles
- Subfunction handles
- Private function handles

Simple Function Handles

These are handles to nonoverloaded MATLAB built-in or M-file functions. Any function handles for which the function type has not yet been determined (e.g., Java methods, nonexistent functions), also fall into this category.

Structure fields:

```
function: function name  
type: 'simple'  
file: 'MATLAB built-in function' for built-ins
```

path and name of the default M-file for nonbuilt-ins (there is no methods field)

Examples:

Using functions on a function handle to a built-in function

```
functions(@ones)
```

```
ans =
```

```
function: 'ones'
```

```
type: 'simple'
```

```
file: 'MATLAB built-in function'
```

Using functions on a function handle to a nonbuilt-in function

```
functions(@fzero)
```

```
ans =
```

```
function: 'fzero'
```

```
type: 'simple'
```

```
file: 'matlabroot\toolbox\matlab\funfun\fzero.m'
```

Overloaded Function Handles

These are handles to MATLAB built-in or M-file functions that are overloaded implementations for different classes.

Structure fields:

```
function: function name  
type: 'overloaded'  
file: 'MATLAB built-in function' for built-ins
```

path and name of the default M-file for nonbuilt-ins methods: [1x1 struct]

Examples:

Using functions on a function handle to a built-in function

```
functions(@display)
```

```
ans =
```

```
function: 'display'
```

```
type: 'overloaded'
```

```
file: 'MATLAB built-in function'
```

```
methods: [1x1 struct]
```

Using functions on a function handle to a nonbuilt-in function

```
functions(@deblank)
```

```
ans =
```

```
function: 'deblank'
```

```
type: 'overloaded'
```

```
file: 'matlabroot\toolbox\matlab\strfun\deblank.m' methods: [1x1 struct]
```

Constructor Function Handles

These are handles to functions that construct objects of MATLAB classes. **Structure fields:**

function: function name
type: 'constructor'
file: path and name of the constructor M-file

(there is no methods field)

Example:

Using functions on a function handle to a constructor function functions(@inline)
ans =
function: 'inline'
type: 'constructor'
file: 'matlabroot\toolbox\matlab\funfun\@inline\inline.m'

Subfunction Handles

These are handles to MATLAB subfunctions, which are functions defined within an M-file that are only visible to the primary function of that M-file. When you use functions on a subfunction handle, the file field of the return structure contains the path and name of the M-file in which the subfunction is defined.

Structure fields:

function: function name
type: 'subfunction'
file: path and name of the M-file defining the subfunction

(there is no methods field)

Example:

The getLocalHandle M-file, shown below, defines a primary function and a subfunction, named subfunc.

```
% -- File GETLOCALHANDLE.M -  
function subhandle = getLocalHandle()  
subhandle = @subfunc; % return handle to subfunction
```

function subfunc() disp 'running subfunc'

A call to getLocalHandle returns a function handle to the subfunction. When you pass that handle to functions, it returns the following information. fhandle = getLocalHandle;
functions(fhandle)

ans =

function: 'subfunc' type: 'subfunction'
file: '\home\user4\getLocalHandle.m'

Private Function Handles

These are handles to MATLAB private functions, which are functions defined in a private subdirectory that are only visible to functions in the parent directory. When you use functions on a private function handle, the file field of the return structure contains the path and name of the M-file in the private subdirectory that defines the function.

Structure fields:

function: function name
type: 'private'
file: path and name of the M-file in \private

(there is no methods field)

Example:

The getPrivateHandle function, shown below, returns a handle to a private function named privatefunc.

```
% -- File GETPRIVATEHANDLE.M -  
function privhandle = getPrivateHandle()  
privhandle = @privatefunc; % return handle to private function
```

The following function, privatefunc, resides in the \private subdirectory.

```
% -- File \PRIVATE\PRIVATEFUNC.M -  
function privatefunc()  
disp 'running privatefunc'
```

A call to getPrivateHandle returns a handle to the function, privatefunc, defined in \private. When you pass that handle to functions, it returns the following information.

```
fhandle = getPrivateHandle;  
functions(fhandle)  
ans =  
function: 'privatefunc'  
type: 'private'  
file: '\home\user4\private\privatefunc.m'
```

Function Handle Operations

MATLAB provides two functions that enable you to convert between a function handle and a function name string. It also provides functions for testing to see if a variable holds a function handle, and for comparing function handles.

Converting Function Handles to Function Names

If you need to perform string operations, such as string comparison or display, on a function handle, you can use func2str to obtain the function name in string format. To convert a function handle to a string

```
fhandle = @sin;  
func2str(fhandle) ans =  
sin
```

Note The func2str command does not operate on nonscalar function handles. Passing a nonscalar function handle to func2str results in an error.

Example - Displaying the Function Name in an Error Message The catcherr function shown here accepts function handle and data arguments and attempts to evaluate the function through its handle. If the function fails to execute, catcherr uses sprintf to display an error message giving the name of the failing function. The function name must be a string for sprintf to display it. The code derives the function name from the function handle using func2str.

```
function catcherr(func, data)
try
ans = feval(func, data);
disp('Answer is:');
ans
catch
sprintf('Error executing function "%s"\n', func2str(func)) end
```

The first call to catcherr, shown below, passes a handle to the round function and a valid data argument. This call succeeds and returns the expected answer. The second call passes the same function handle and an improper data type (a MATLAB structure). This time, round fails, causing catcherr to display an error message that includes the failing function name.

```
catcherr(@round, 5.432)
ans =
Answer is 5

xstruct.value = 5.432;
catcherr(@round, xstruct)
Error executing function "round"
```

Converting Function Names to Function Handles

Using the str2func function, you can construct a function handle from a string containing the name of a MATLAB function. To convert the string, 'sin', into a handle for that function

```
fh = str2func('sin')
fh =
@sin
```

If you pass a function name string in a variable, the function that receives the variable can convert the function name to a function handle using str2func. The example below passes the variable, funcname, to function makeHandle, which then creates a function handle.

```
function fh = makeHandle(funcname)
fh = str2func(funcname);
% -- end of makeHandle.m file -

makeHandle('sin') ans =
@sin
```

You can also perform the str2func operation on a cell array of function name strings. In this

```
case,str2func returns an array of function handles. fh_array = str2func({'sin' 'cos' 'tan'})  
fh_array =  
@sin @cos @tan
```

Example - More Flexible Parameter Checking

In the following example, themyminbnd function expects to receive either a function handle or string in the first argument. If you pass a string,myminbnd constructs a function handle from it usingstr2func, and then uses that handle in a call to fminbnd.

```
function myminbnd(fhandle, lower, upper)  
if ischar(fhandle)  
    disp 'converting function string to function handle ...'  
    fhandle = str2func(fhandle);  
end  
fminbnd(fhandle, lower, upper)
```

Whether you callmyminbnd with a function handle or function name string, it is able to handle the argument appropriately.

```
myminbnd('humps', 0.3, 1)  
converting function string to function handle ...  
ans =
```

```
0.6370
```

Testing for Data Type

The isa function identifies the data type or class of a MATLAB variable or object. You can see if a variable holds a function handle by usingisa with the function_handle tag. The following function tests an argument passed in to see if it is a function handle before attempting to evaluate it.

```
function evaluate_handle(arg1, arg2)  
if isa(arg1, 'function_handle')  
    feval(arg1, arg2);  
  
else  
    disp 'You need to pass a function handle';  
end
```

Testing for Equality

You can use the isequal function to compare two function handles for equality. For example, you want to execute one particular method you have written for an overloaded function. You create a function handle within a confined scope so that it provides access to that method alone. The function shown below, test_myfun, receives this function handle in the first argument,arg1, and evaluates it.

Before evaluating the handle in arg1, test_myfun checks it against another function handle that has a broader definition. This other handle,@myfun, provides access to all methods for the function. If the function handle inarg1 represents more than the one intended method, an error message is displayed and the function is not evaluated.

```
function test_myfun(arg1, arg2)
if isequal(arg1, @myfun)
    disp 'Function handle holds unexpected context'

else
    feval(arg1, arg2);
end
```

Saving and Loading Function Handles

Saving and Loading Function Handles

You can use the MATLAB save andload functions to save function handles to MAT files, and then load them back into your MATLAB workspace later on. This example shows an array of function handles saved to the file,savefile, and then restored.

```
fh_array = [@sin @cos @tan];
save savefile fh_array;
clear

load savefile
whos
Name Size Bytes Class
fh_array 1x3 48 function_handle array Grand total is 3 elements using 48 bytes
```

Possible Effects of Changes Made Between Save and Load If you load a function handle that you saved in an earlier MATLAB session, the following conditions could cause unexpected behavior:

- Any of the M-files that define the function have been moved, and thus no longer exist on the path stored in the handle.
- You load the function handle into an environment different from that in which it was saved. For example, the source for the function either doesn't exist or is located in a different directory than on the system on which the handle was saved.
- You have overloaded the function with additional methods since the save was done. The function handle you just loaded doesn't know about the new methods.

In the first two cases, the function handle is now invalid, since it no longer maps to any existing source code. Although the handle is invalid, MATLAB still performs the load successfully and without displaying a warning. An attempt to evaluate the handle however results in an error.

Handling Error Conditions

The following are error conditions associated with the use of function handles.

Handles to Nonexistent Functions

If you create a handle to a function that does not exist, MATLAB catches the error when the handle is evaluated by feval. MATLAB allows you to assign an invalid handle and use it in such operations as func2str, but will catch and report an error when you attempt to use it in a runtime operation. For example,

```
fhandle = @no_such_function;  
  
func2str(fhandle)  
ans =  
no_such_function  
  
feval(fhandle)  
??? Error using ==> feval  
Undefined function 'no_such_function'.
```

Including Path In the Function Handle Constructor

You construct a function handle using the *at* sign, `@`, or the str2func function. In either case, you specify the function using only the simple function name. The function name cannot include path information. Either of the following successfully creates a handle to the deblank function.

```
fhandle = @deblank;  
fhandle = str2func('deblank');
```

The next example includes the path to deblank.m, and thus returns an error.

```
fhandle = str2func(which('deblank'))  
??? Error using ==> str2func  
Invalid function name  
  
'matlabroot\toolbox\matlab\strfun\deblank.m'.
```

Handling Error Conditions

Evaluating a Nonscalar Function Handle

The feval function evaluates function handles only if they are scalar. Calling feval with a nonscalar function handle results in an error.

```
feval([@sin @cos], 5)  
??? Error using ==> feval  
Function_handle argument must be scalar.
```

Historical Note - Evaluating Function Names

Evaluating a function by means of a function handle replaces the former MATLAB mechanism of evaluating a function through a string containing the function name. For example, of the following two lines of code that evaluate the humps function, the second supersedes the first and is considered to be the preferable mechanism to use.

```
feval('humps', 0.5674);
feval(@humps, 0.5674); % uses a function name string
% uses a function handle
```

To support backward compatibility, feval still accepts a function name string as a first argument and evaluates the function named in the string. However, function handles offer you the additional performance, reliability, and source file control benefits listed in the section “Benefits of Using Function Handles” on page 21-3.

22

MATLAB Classes and Objects

Classes and Objects: An Overview	22-3
Designing User Classes in MATLAB	22-9
Overloading Operators and Functions	22-21
Example - A Polynomial Class	22-24
Building on Other Classes	22-35
Example - Assets and Asset Subclasses	22-38
Example - The Portfolio Container	22-54
Saving and Loading Objects	22-60
Example - Defining saveobj and loadobj for Portfolio	22-61
Object Precedence	22-65
How MATLAB Determines Which Method to Call	22-67

This chapter describes how to define classes in MATLAB. Classes and objects enable you to add new data types and new operations to MATLAB. The *class* of a variable describes the structure of the variable and indicates the kinds of operations and functions that can apply to the variable. An *object* is an instance of a particular class. The phrase *object-oriented programming* describes an approach to writing programs that emphasizes the use of classes and objects.

The following topics and examples are presented in this chapter:

- “Classes and Objects: An Overview”
- “Designing User Classes in MATLAB”
- “Overloading Operators and Functions”
- “Example - A Polynomial Class”
- “Building on Other Classes”
- “Example - Assets and Asset Subclasses”
- “Example - The Portfolio Container”
- “Saving and Loading Objects”
- “Example - Defining saveobj and loadobj for Portfolio”
- “Object Precedence”
- “How MATLAB Determines Which Method to Call”

Classes and Objects: An Overview

You can view classes as new data types having specific behaviors defined for the class. For

example, a polynomial class might redefine the addition operator (+) so that it correctly performs the operation of addition on polynomials. Operations defined to work with objects of a particular class are known as *methods* of that class.

You can also view classes as new items that you can treat as single entities. An example is an arrow object that MATLAB can display on graphs (perhaps composed of MATLAB line and patch objects) and that has properties like a Handle Graphics object. You can create an arrow simply by instantiating the arrow class.

You can add classes to your MATLAB environment by specifying a MATLAB structure that provides data storage for the object and creating a class directory containing M-files that operate on the object. These M-files contain the methods for the class. The class directory can also include functions that define the way various MATLAB operators, including arithmetic operations, subscript referencing, and concatenation, apply to the objects. Redefining how a built-in operator works for your class is known as *overloading* the operator.

Features of Object-Oriented Programming

When using well-designed classes, object-oriented programming can significantly increase code reuse and make your programs easier to maintain and extend. Programming with classes and objects differs from ordinary structured programming in these important ways:

- **Function and operator overloading** . You can create methods that override existing MATLAB functions. When you call a function with a user-defined object as an argument, MATLAB first checks to see if there is a method defined for the object's class. If there is, MATLAB calls it, rather than the normal MATLAB function.
- **Encapsulation of data and methods** . Object properties are not visible from the command line; you can access them only with class methods. This protects the object properties from operations that are not intended for the object's class.
- **Inheritance** . You can create class hierarchies of parent and child classes in which the child class inherits data fields and methods from the parent. A child class can inherit from one parent (*single inheritance*) or many parents (*multiple inheritance*). Inheritance can span one or more generations. Inheritance enables sharing common parent functions and enforcing common behavior amongst all child classes.
- **Aggregation** . You can create classes using *aggregation*, in which an object contains other objects. This is appropriate when an object type is part of another object type. For example, a savings account object might be a part of a financial portfolio object.

MATLAB Data Class Hierarchy

All MATLAB data types are designed to function as classes in object-oriented programming. The diagram below shows the fourteen fundamental data types (or classes) defined in MATLAB. You can add new data types to MATLAB by extending the class hierarchy.

ARRAY

```
char NUMERIC cell structure function handle  
user class java class  
  
int8 , uint8, single int16,uint16,  
int32,uint32  
double  
  
sparse
```

The diagram shows a *user class* that inherits from the structure class. All classes that you create are structure based since this is the point in the class hierarchy where you can insert your own classes. (For more information about MATLAB data types, see the section on “Data Types.”)

Creating Objects

You create an object by calling the class constructor and passing it the appropriate input arguments. In MATLAB, constructors have the same name as the class name. For example, the statement,

```
p = polynom([1 0 -2 5]);
```

creates an object named p belonging to the classpolynom. Once you have created a polynom object, you can operate on the object using methods that are defined for thepolynom class. See “Example - A Polynomial Class” on page 22-24 for a description of thepolynom class.

Invoking Methods on Objects

Class methods are M-file functions that take an object as one of the input arguments. The methods for a specific class must be placed in the class directory for that class (the@*class_name* directory). This is the first place that MATLAB looks to find a class method.

The syntax for invoking a method on an object is similar to a function call. Generally, it looks like

```
[out1,out2,...] = method_name(object,arg1,arg2, ...);
```

For example, suppose a user-defined class called polynom has achar method defined for the class. This method converts a polynom object to a character string and returns the string. This statement calls the char method on the polynom objectp.

```
s = char(p);
```

Using theclass function, you can confirm that the returned values is a character string.

```
class(s)
```

```
ans =
```

```
char
```

```
s
```

```
s =
```

$x^3 - 2x^5$ You can use themethods command to produce a list of all of the methods that are defined for a class.

Private Methods

Private methods can be called only by other methods of their class. You define private methods by placing the associated M-files in a private subdirectory of the `@class_name` directory. In the example,

```
@class_name/private/update_obj.m
```

the method `update_obj` has scope only within the `class_name` class. This means that `update_obj` can be called by any method that is defined in the `@class_name` directory, but it cannot be called from the MATLAB command line or by methods outside of the class directory, including parent methods.

Private methods and private functions differ in that private methods (in fact all methods) have an object as one of their input arguments and private functions do not. You can use private functions as helper functions, such as described in the next section.

Helper Functions

In designing a class, you may discover the need for functions that perform support tasks for the class, but do not directly operate on an object. These functions are called *helper functions*. A helper function can be a subfunction in a class method file or a private function. When determining which version of a particular function to call, MATLAB looks for these functions in the order listed above. For more information about the order in which MATLAB calls functions and methods, see “How MATLAB Determines Which Method to Call” on page 22-67.

Debugging Class Methods

You can use the MATLAB debugging commands with object methods in the same way that you use them with other M-files. The only difference is that you need to include the class directory name before the method name in the command call, as shown in this example using `dbstop`.

```
dbstop @polynom/char
```

While debugging a class method, you have access to all methods defined for the class, including inherited methods, private methods, and private functions.

Changing Class Definition

If you change the class definition, such as the number or names of fields in a class, you must issue a

```
clear classes
```

command to propagate the changes to your MATLAB session. This command also clears all objects from the workspace. See the `clear` command help entry for more information.

Setting Up Class Directories

The M-files defining the methods for a class are collected together in a directory referred to as

the class directory. The directory name is formed with the class name preceded by the character @. For example, one of the examples used in this chapter is a class involving polynomials in a single variable. The name of the class, and the name of the class constructor, ispolynom. The M-files defining a polynomial class would be located in directory with the name @polynom.

The class directories are subdirectories of directories on the MATLAB search path, but are not themselves on the path. For instance, the new@polynom directory could be a subdirectory of MATLAB's working directory or your own personal directory that has been added to the search path.

Adding the Class Directory to the MATLAB Path

After creating the class directory, you need to update the MATLAB path so that MATLAB can locate the class source files. The class directory should not be directly on the MATLAB path. Instead, you should add the parent directory to the MATLAB path. For example, if the@polynom class directory is located at

c:\my_classes\@polynom

you add the class directory to the MATLAB path with the addpath command
addpath c:\my_classes;

If you create a class directory with the same name as another class, MATLAB treats the two class directories as a single directory when locating class methods. For more information, see “How MATLAB Determines Which Method to Call” on page 22-67.

Data Structure

One of the first steps in the design of a new class is the choice of the data structure to be used by the class. Objects are stored in MATLAB structures. The fields of the structure, and the details of operations on the fields, are visible only within the methods for the class. The design of the appropriate data structure can affect the performance of the code.

Tips for C++ and Java Programmers

If you are accustomed to programming in other object-oriented languages, such as C++ or Java, you will find that the MATLAB programming language differs from these languages in some important ways:

- In MATLAB, method dispatching is not syntax based, as it is in C++ and Java. When the argument list contains objects of equal precedence, MATLAB uses the left-most object to select the method to call.
- In MATLAB, there is no equivalent to a destructor method. To remove an object from the workspace, use the clear function.
- Construction of MATLAB data types occurs at runtime rather than compile time. You register an object as belonging to a class by calling theclass function.
- When using inheritance in MATLAB, the inheritance relationship is established in the child class by creating the parent object, and then calling the class function. For more information on writing constructors for inheritance relationships, see “Building on Other Classes” on page 22-35.

- When using inheritance in MATLAB, the child object contains a parent object in a property with the name of the parent class.
- In MATLAB, there is no passing of variables by reference. When writing methods that update an object, you must pass back the updated object and use an assignment statement. For instance, this call to the set method updates the name field of the objectA and returns the updated object.

A = set(A,'name','John Smith');

- In MATLAB, there is no equivalent to an abstract class.
- In MATLAB, there is no equivalent to the C++ scoping operator.
- In MATLAB, there is no virtual inheritance or virtual base classes.
- In MATLAB, there is no equivalent to C++ templates.

Designing User Classes in MATLAB

This section discusses how to approach the design of a class and describes the basic set of methods that should be included in a class.

The M ATLAB Canonical Class

When you design a MATLAB class, you should include a standard set of methods that enable the class to behave in a consistent and logical way within the MATLAB environment. Depending on the nature of the class you are defining, you may not need to include all of these methods and you may include a number of other methods to realize the class's design goals.

This table lists the basic methods included in MATLAB classes.

Class Method Description

class constructor display

set andget

subsref andsubsasgn

end

subsindex converters likedouble andchar

Creates an object of the class

Called whenever MATLAB displays the contents of an object (e.g., when an expression is entered without terminating with a semicolon)

Accesses class properties

Enables indexed reference and assignment for user objects

Supports send syntax in indexing expressions using an object; e.g., A(1:end)

Supports using an object in indexing expressions Methods that convert an object to a MATLAB data type

The following sections discuss the implementation of each type of method, as well as providing references to examples used in this chapter.

The Class Constructor Method

The @ directory for a particular class must contain an M-file known as the *constructor* for that class. The name of the constructor is the same as the name of the directory (excluding the @ prefix and.mextension) that defines the name of the class. The constructor creates the object by initializing the data structure and instantiating an object of the class.

Guidelines for Writing a Constructor

Class constructors must perform certain functions so that objects behave correctly in the MATLAB environment. In general, a class constructor must handle three possible combinations of input arguments:

- No input arguments
- An object of the same class as an input argument
- The input arguments used to create an object of the class (typically data of some kind)

No Input Arguments. If there are no input arguments, the constructor should create a default object. Since there are no inputs, you have no data from which to create the object, so you simply initialize the object's data structures with empty or default values, call the class function to instantiate the object, and return the object as the output argument. Support for this syntax is required for two reasons:

- When loading objects into the workspace, the load function calls the class constructor with no arguments.
- When creating arrays of objects, MATLAB calls the class constructor to add objects to the array.

Object Input Argument. If the first input argument in the argument list is an object of the same class, the constructor should simply return the object. Use the isa function to determine if an argument is a member of a class. See “Overloading the + Operator” on page 22-29 for an example of a method that uses this constructor syntax.

Data Input Arguments. If the input arguments exist and are not objects of the same class, then the constructor creates the object using the input data. Of course, as in any function, you should perform proper argument checking in your constructor function. A typical approach is to use varargin input argument and a switch statement to control program flow. This provides an easy way to accommodate the three cases: no inputs, object input, or the data inputs used to create an object.

It is in this part of the constructor that you assign values to the object's data structure, call the class function to instantiate the object, and return the object as the output argument. If necessary, place the object in an object hierarchy using the superiorto and inferiorto functions.

Using the class Function in Constructors

Within a constructor method, you use the class function to associate an object structure with a particular class. This is done using an internal class tag that is only accessible using the class and isa functions. For example, this call to the class function identifies the object p to be of type polynom.

```
p = class(p,'polynom');
```

Examples of Constructor Methods

See the following sections for examples of constructor methods:

- “The Polynom Constructor Method” on page 22-24

- “The Asset Constructor Method” on page 22-39
- “The Stock Constructor Method” on page 22-46
- “The Portfolio Constructor Method” on page 22-55

Identifying Objects Outside the Class Directory

The `class` and `isa` functions used in constructor methods can also be used outside of the class directory. The expression

```
isa(a,'class_name');
```

checks whether `a` is an object of the specified class. For example, if `p` is a `polynom` object, each of the following expressions is true.

```
isa(pi,'double');
isa('hello','char');
isa(p,'polynom');
```

Outside of the class directory, the `class` function takes only one argument (it is only within the constructor that `class` can have more than one argument).

The expression

```
class(a)
```

returns a string containing the class name of `a`. For example,

```
class(pi),
class('hello'),
class(p)
```

```
return
```

```
'double',
'char',
'polynom'
```

Use the `whos` function to see what objects are in the MATLAB workspace. `whos`

Name	Size	Bytes	Class
p	1x1	156	polynom object

The `display` Method

MATLAB calls a method named `display` whenever an object is the result of a statement that is not terminated by a semicolon. For example, creating the variable `a`, which is a double, calls MATLAB's `display` method for doubles.

```
a = 5
a =
5
```

You should define a `display` method so MATLAB can display values on the command line when referencing objects from your class. In many classes, `display` can simply print the variable name, and then use the `char` converter method to print the contents or value of the variable, since

MATLAB displays output as strings. You must define thechar method to convert the object's data to a character string.

Examples of display Methods

See the following sections for examples of display methods:

- “The Polynom display Method” on page 22-28
- “The Asset display Method” on page 22-44
- “The Stock display Method” on page 22-52
- “The Portfolio display Method” on page 22-56

Accessing Object Data

You need to write methods for your class that provide access to an object's data. Accessor methods can use a variety of approaches, but all methods that change object data always accept an object as an input argument and return a new object with the data changed. This is necessary because MATLAB does not support passing arguments by reference (i.e., pointers). Functions can change only their private, temporary copy of an object. Therefore, to change an existing object, you must create a new one, and then replace the old one.

The following sections provide more detail about implementation techniques for the set, get, subsasgn, andsubsref methods.

The set and get Methods

The set and get methods provide a convenient way to access object data in certain cases. For example, suppose you have created a class that defines an arrow object that MATLAB can display on graphs (perhaps composed of existing MATLAB line and patch objects).

To produce a consistent interface, you could define set and get methods that operate on arrow objects the way the MATLABset andget functions operate on built-in graphics objects. The set and get verbs convey what operations they perform, but insulate the user from the internals of the object.

Examples of set and get Methods

See the following sections for examples of set andget methods:

- “The Asset get Method” on page 22-41 and “The Asset set Method” on page 22-41
- “The Stock get Method” on page 22-48 and “The Stock set Method” on page 22-48

Property Name Methods

As an alternative to a generalset method, you can write a method to handle the assignment of an individual property. The method should have the same name as the property name.

For example, if you defined a class that creates objects representing employee data, you might have a field in an employee object called salary. You could

then define a method called salary.m that takes an employee object and a value as input

arguments and returns the object with the specified value set.

Indexed Reference Using `subsref` and `subsasgn`

User classes implement new data types in MATLAB. It is useful to be able to access object data via an indexed reference, as is possible with MATLAB's built-in data types. For example, if `A` is an array of class `double`, `A(i)` returns the i^{th} element of `A`.

As the class designer, you can decide what an index reference to an object means. For example, suppose you define a class that creates polynomial objects and these objects contain the coefficients of the polynomial.

An indexed reference to a polynomial object,
`p(3)`

could return the value of the coefficient of x^3 , the value of the polynomial at $x = 3$, or something different depending on the intended design.

You define the behavior of indexing for a particular class by creating two class methods – `subsref` and `subsasgn`. MATLAB calls these methods whenever a subscripted reference or assignment is made on an object from the class. If you do not define these methods for a class, indexing is undefined for objects of this class.

In general, the rules for indexing objects are the same as the rules for indexing structure arrays. For details, see “[Structures](#)” on page 20-4.

Handling Subscripted Reference

The use of a subscript or field designator with an object on the right-hand side of an assignment statement is known as a *subscripted reference*. MATLAB calls a method named `subsref` in these situations. Object subscripted references can be of three forms – an array index, a cell array index, and a structure field name:

`A(I)`
`A{I}`
`A.field`

Each of these results in a call by MATLAB to the `subsref` method in the class directory. MATLAB passes two arguments to `subsref`.

`B = subsref(A,S)`

The first argument is the object being referenced. The second argument, `S`, is a structure array with two fields:

- `S.type` is a string containing '()' , '{}', or '.' specifying the subscript type. The parentheses represent a numeric array; the curly braces, a cell array; and the dot, a structure array.
- `S.subs` is a cell array or string containing the actual subscripts. A colon used as a subscript is passed as the string ':'.

For instance, the expression

`A(1:2,:)`

causes MATLAB to callsubsref(A,S), where S is a 1-by-1 structure with

```
S.type = '()'  
S.subs = {1:2,:'}
```

Similarly, the expression

```
A{1:2}
```

uses

```
S.type ='{'  
S.subs = {1:2}  
The expression  
A.field  
callssubsref(A,S) where  
S.type = '.'  
S.subs = 'field'
```

These simple calls are combined for more complicated subscripting expressions. In such cases, length(S) is the number of subscripting levels. For example,

```
A(1,2).name(3:4)
```

callssubsref(A,S), where S is a 3-by-1 structure array with the values:

```
S(1).type = '()' S(2).type = '.' S(3).type = '()' S(1).subs = '{1,2}' S(2).subs = 'name' S(3).subs = '{3:4}'
```

How to Write subsref

The subsref method must interpret the subscripting expressions passed in by MATLAB. A typical approach is to use the switch statement to determine the type of indexing used and to obtain the actual indices. The following three code fragments illustrate how to interpret the input arguments. In each case, the function must return the value B.

For an array index:

```
switch S.type  
case '()'  
B = A(S.subs{:});  
end
```

For a cell array:

```
switch S.type  
case '{}'  
B = A(S.subs{:}); % A is a cell array  
end
```

For a structure array:

```
switch S.type  
case '.'  
switch S.subs  
case 'field1'
```

```
B = A.field1;  
case 'field2'  
B = A.field2;  
end  
end
```

Examples of the subsref Method

See the following sections for examples of thesubsref method:

- “The Polynom subsref Method” on page 22-28
- “The Asset subsref Method” on page 22-42
- “The Stock subsref Method” on page 22-49
- “The Portfolio subsref Method” on page 22-64

Handling Subscripted Assignment

The use of a subscript or field designator with an object on the left -hand side of an assignment statement is known as a *subscripted assignment*. MATLAB calls a method namedsubsasgn in these situations. Object subscripted assignment can be of three forms – an array index, a cell array index, and a structure field name.

```
A(I) = B  
A{I} = B  
A.field = B
```

Each of these results in a call to subsasgn of the form

```
A = subsasgn(A,S,B)
```

The first argument, A, is the object being referenced. The second argument,S, has the same fields as those used withsubsref. The third argument,B, is the new value.

Examples of the subsasgn Method

See the following sections for examples of thesubsasgn method:

- “The Asset subsasgn Method” on page 22-43
- “The Stock subsasgn Method” on page 22-51

Object Indexing Within Methods

If a subscripted reference is made within a class method, MATLAB uses its built-insubsref function to access data within the method’s own class. If the method accesses data from another class, MATLAB calls the overloaded subsref function in that class. The same holds true for subscripted assignment and subsasgn.

The following example shows a method, testref, that is defined in the class, employee. This method makes a reference to a field,address, in an object of its own class. For this, MATLAB uses the built-insubsref function. It also references the same field in another class, this time using the overloaded subsref of that class.

```
% ---- EMPLOYEE class method: testref.m ---
```

```

function testref(myclass,otherclass)
myclass.address % use built-in subsref otherclass.address % use overloaded subsref The example
creates an employee object and a company object. emp1 = employee('Johnson','Chicago');
comp = company('The MathWorks','Natick');
The employee class method, testref, is called. MATLAB uses an overloaded subsref only to
access data outside of the method's own class. testref(emp1,comp)
ans = % built-in subsref was called
Chicago
ans = % @company\subsref was called
Executing @company\subsref ...
Natick

```

Defining end Indexing for an Object

When you use end in an object indexing expression, MATLAB calls the object's end class method. If you want to be able to use end in indexing expressions involving objects of your class, you must define an end method for your class.

The end method has the calling sequence

end(a,k,n)

where a is the user object, k is the index in the expression where the end syntax is used, and n is the total number of indices in the expression.

For example, consider the expression

A(end,1,:)

MATLAB calls the end method defined for the object A using the arguments end(A,1,2)

That is, the end statement occurs in the first index element and there are two index elements. The class method for end must then return the index value for the last element of the first dimension. When you implement the end method for your class, you must ensure it returns a value appropriate for the object.

Indexing an Object with Another Object

When MATLAB encounters an object as an index, it calls the subsindex method defined for the object. For example, suppose you have an object a and you want to use this object to index into another object b.

c = b(a);

A subsindex method might do something as simple as convert the object to double format to be used as an index, as shown in this sample code.

```

function d = subsindex(a)
%SUBSINDEX
% convert the object a to double format to be used
% as an index in an indexing expression
d = double(a);

```

subsindex

values are 0-based, not 1-based.

Converter Methods

A converter method is a class method that has the same name as another class, such as `char` or `double`. Converter methods accept an object of one class as input and return an object of another class. Converters enable you to:

- Use methods defined for another class
- Ensure that expressions involving objects of mixed class types execute properly

A converter function call is of the form

`b = class_name(a)`

where `a` is an object of a class other than `class_name`. In this case, MATLAB looks for a method called `class_name` in the class directory for object `a`. If the input object is already of type `class_name`, then MATLAB calls the constructor, which just returns the input argument.

Examples of Converter Methods

See the following sections for examples of converter methods:

- “The Polynom to Double Converter” on page 22-25
- “The Polynom to Char Converter” on page 22-26

Overloading Operators and Functions

In many cases, you may want to change the behavior of MATLAB’s operators and functions for cases when the arguments are objects. You can accomplish this by *overloading* the relevant functions. Overloading enables a function to handle different types and numbers of input arguments and perform whatever operation is appropriate for the highest-precedence object. See “Object Precedence” on page 22-65 for more information on object precedence.

Overloading Operators

Each built-in MATLAB operator has an associated function name (e.g., the `+` operator has an associated `plus.m` function). You can overload any operator by creating an M-file with the appropriate name in the class directory. For example, if either `p` or `q` is an object of type `class_name`, the expression

`p + q`

generates a call to a function `@class_name/plus.m`, if it exists. If `p` and `q` are both objects of different classes, then MATLAB applies the rules of precedence to determine which method to use.

Examples of Overloaded Operators

See the following sections for examples of overloaded operators:

- “Overloading the `+` Operator” on page 22-29
- “Overloading the `\` Operator” on page 22-30
- “Overloading the `\`` Operator” on page 22-30

The following table lists the function names for most of MATLAB's operators.

Operation M-File Description	a + b	plus(a,b)	Binary addition	a - b	minus(a,b)	Binary subtraction			
-a	uminus(a)	Unary minus	+a	uplus(a)	Unary plus	a.*b	times(a,b)	Element-wise multiplication	
a*b	mtimes(a,b)	Matrix multiplication	a./b	rdivide(a,b)	Right element-wise division	a.\b			
ldivide(a,b)	Left element-wise division	a\b	mrddivide(a,b)	Matrix right division	a\b	mldivide(a,b)			
Matrix left division	a.^b	power(a,b)	Element-wise power	a^b	mpower(a,b)	Matrix power	a < b		
lt(a,b)	Less than	a > b	gt(a,b)	Greater than	a <= b	le(a,b)	Less than or equal to	a >= b	ge(a,b)
Greater than or equal to	a ~= b	ne(a,b)	Not equal to	a == b	eq(a,b)	Equality	a & b	and(a,b)	
Logical AND	a b	or(a,b)	Logical OR	~a	not(a)	Logical NOT			
Operation M-File Description									
a:d:b	colon(a,d,b)	a:b	colon(a,b)						
Colon operator									
a'	ctranspose(a)	Complex conjugate transpose	a.'	transpose(a)	Matrix transpose command				
window	display(a)	display	output						
Display method									
[a b]	horz								
cat(a,b,...)									
Horizontal concatenation									
[a; b]	vert								
cat(a,b,...)									
Vertical concatenation									
a(s1,s2,...sn)	subsref(a,s)	Subscripted reference	a(s1,...,sn) = b	subsasgn(a,s,b)	Subscripted assignment	b(a)	subsindex(a)	Subscript index	

Overloading Functions

You can overload any function by creating a function of the same name in the class directory. When a function is invoked on an object, MATLAB always looks in the class directory before any other location on the search path. To overload theplot function for a class of objects, for example, simply place your version of plot.m in the appropriate class directory.

Examples of Overloaded Functions

See the following sections for examples of overloaded functions:

- “Overloading Functions for the Polynom Class” on page 22-31
- “The Portfolio pie3 Method” on page 22-57

Example - A Polynomial Class

This example implements a MATLAB data type for polynomials by defining a new class called polynom. The class definition specifies a structure for data storage and defines a directory (@polynom) of methods that operate on polynom objects.

Polynom Data Structure

The polynom class represents a polynomial with a row vector containing the coefficients of powers of the variable, in decreasing order. Therefore, a polynom object is a structure with a single field,p.c, containing the coefficients. This field is accessible only within the methods in

the @polynom directory.

Polynom Methods

To create a class that is well behaved within the MATLAB environment and provides useful functionality for a polynomial data type, the polynom class implements the following methods:

- A constructor method polynom.m
- A polynom to double converter
- A polynom to char converter
- A display method
- A subsref method
- Overloaded +, -, *, and / operators
- Overloaded roots, polyval, plot, and diff functions

The Polynom Constructor Method

Here is the polynom class constructor, @polynom/polynom.m.

```
function p = polynom(a)
%POLYNOM Polynomial class constructor.
% p = POLYNOM(v) creates a polynomial object from the vector v, % containing the
coefficients of descending powers of x. if nargin == 0

p.c = [];
p = class(p,'polynom');
elseif isa(a,'polynom')

p = a;
else
p.c = a(:)';
p = class(p,'polynom'); end
```

Constructor Calling Syntax

You can call the polynom constructor method with one of three different arguments:

- No Input Argument – If you call the constructor function with no arguments, it returns a polynom object with empty fields.
- Input Argument is an Object – If you call the constructor function with an input argument that is already a polynom object, MATLAB returns the input argument. The isafunction (pronounced “is a”) checks for this situation.
- Input Argument is a coefficient vector – If the input argument is a variable that is not a polynom object, reshape it to be a row vector and assign it to the .c field of the object’s structure. The class function creates the polynom object, which is then returned by the constructor.

An example use of the polynom constructor is the statement

p = polynom([1 0 -2 -5])

This creates a polynomial with the specified coefficients.

Converter Methods for the Polynom Class

A converter method converts an object of one class to an object of another class. Two of the most important converter methods contained in MATLAB classes are double and char. Conversion to double produces MATLAB's traditional matrix, although this may not be appropriate for some classes. Conversion to char is useful for producing printed output.

The Polynom to Double Converter

The double converter method for the polynom class is a very simple M-file, @polynom/double.m, which merely retrieves the coefficient vector.

```
function c = double(p)
% POLYNOM/DOUBLE Convert polynom object to coefficient vector. % c = DOUBLE(p)
% converts a polynomial object to the vector c % containing the coefficients of descending powers
% of x. c = p.c;
```

On the object p,

```
p = polynom([1 0 -2 -5])
```

the statement

```
double(p)
```

returns

```
ans =
```

```
1 0 -2 -5
```

The Polynom to Char Converter

The converter to char is a key method because it produces a character string involving the powers of an independent variable, x. Therefore, once you have specified x, the string returned is a syntactically correct MATLAB expression, which you can then evaluate.

Here is @polynom/char.m.

```
function s = char(p)
% POLYNOM/CHAR
% CHAR(p) is the string representation of p.c
if all(p.c == 0)

    s = '0';
else
    d = length(p.c) - 1;
    s = [];
    for a = p.c;
        if a ~= 0;
            if ~isempty(s)
                if a > 0
                    s = [s ' + '];
                else
                    s = [s ' - '];
                end
                a = -a;
            end
        end
    end
end
```

```

end
end
if a ~= 1 | d == 0
s = [s num2str(a)]; if d > 0
s = [s '*']; end
end
if d >= 2
s = [s 'x^' int2str(d)]; elseif d == 1
s = [s 'x'];
end
d = d - 1;
end
end

```

Evaluating the Output

If you create the polynom object p

```
p = polynom([1 0 -2 -5]);
```

and then call the char method on p

```
char(p)
```

MATLAB produces the result

```

ans =
x^3 - 2*x - 5

```

The value returned by char is a string that you can pass to eval once you have defined a scalar value for x. For example,

```

x = 3;
eval(char(p))
ans =
16

```

See “The Polynom subsref Method” on page 22-28 for a better method to evaluate the polynomial.

The Polynom display Method

Here is @polynom/display.m. This method relies on the char method to produce a string representation of the polynomial, which is then displayed on the screen. This method produces output that is the same as standard MATLAB output. That is, the variable name is displayed followed by an equal sign, then a blank line, then a new line with the value.

```

function display(p)
% POLYNOM/DISPLAY Command window display of a polynom disp(' ');
disp(['inputname(1)' '=' ])
disp(' ');
disp([' ' char(p)])
disp(' ');

```

The statement

```
p = polynom([1 0 -2 -5])
```

creates a polynom object. Since the statement is not terminated with a semicolon, the resulting output is

```
p =
x^3 - 2*x - 5
```

The Polynom subsref Method

Suppose the design of the polynom class specifies that a subscripted reference to a polynom object causes the polynomial to be evaluated with the value of the independent variable equal to the subscript. That is, for a polynom object *p*,

```
p = polynom([1 0 -2 -5]);
```

the following subscripted expression returns the value of the polynomial at $x = 3$ and $x = 4$.

```
p([3 4])
```

```
ans =
```

```
16 51
```

subsref Implementation Details

This implementation takes advantage of the *char* method already defined in the polynom class to produce an expression that can then be evaluated.

```
function b = subsref(a,s)
% SUBSREF
switch s.type
case '()'

ind = s.subs{;};
for k = 1:length(ind)
b(k) = eval(strrep(char(a),'x',num2str(ind(k))));

end
otherwise
error('Specify value for x as p(x)')
end
```

Once the polynomial expression has been generated by the *char* method, the *strrep* function is used to swap the passed in value for the character *x*. The *eval* function then evaluates the expression and returns the value in the output argument.

Overloading Arithmetic Operators for polynom

Several arithmetic operations are meaningful on polynomials and should be implemented for the polynom class. When overloading arithmetic operators, keep in mind what data types you want to operate on. In this section, *theplus*, *minus*, and *mtimes* methods are defined for the polynom class to handle addition, subtraction, and multiplication on polynom/polynom and polynom/double combinations of operands.

Overloading the + Operator

If either p or q is a polynom, the expression

$p + q$

generates a call to a function @polynom/plus.m, if it exists (unless p or q is an object of a higher precedence, as described in “Object Precedence” on page 22-65).

The following M-file redefines the + operator for the polynom class.

```
function r = plus(p,q)
% POLYNOM/PLUS Implement p + q for polynomials.
p = polynom(p);
q = polynom(q);
k = length(q.c) - length(p.c);
r = polynom([zeros(1,k) p.c] + [zeros(1,-k) q.c]);
```

The function first makes sure that both input arguments are polynomials. This ensures that expressions such as

$p + 1$

that involve both a polynom and a double, work correctly. The function then accesses the two coefficient vectors and, if necessary, pads one of them with zeros to make them the same length. The actual addition is simply the vector sum of the two coefficient vectors. Finally, the function calls the polynom constructor a third time to create the properly typed result.

Overloading the - Operator

You can implement the overloaded minus operator (-) using the same approach as the plus (+) operator. MATLAB calls @polynom/minus.m to compute $p - q$.

```
function r = minus(p,q)
% POLYNOM/MINUS Implement p - q for polynomials.
p = polynom(p);
q = polynom(q);
k = length(q.c) - length(p.c);
r = polynom([zeros(1,k) p.c] - [zeros(1,-k) q.c]);
```

Overloading the * Operator

MATLAB calls the method @polynom/mtimes.m to compute the product $p * q$. The letter m at the beginning of the function name comes from the fact that it is overloading MATLAB’s *matrix* multiplication. Multiplication of two polynomials is simply the convolution of their coefficient vectors.

```
function r = mtimes(p,q)
% POLYNOM/MTIMES Implement p * q for polynomials.
p = polynom(p);
q = polynom(q);
r = polynom(conv(p.c,q.c));
```

Using the Overloaded Operators

Given the polynom object

```
p = polynom([1 0 -2 -5])
```

MATLAB calls these two functions @polynom/plus.m and @polynom/mtimes.m when you issue the statements

```
q = p+1
```

```
r = p*q
```

to produce

```
q =
```

```
x^3 - 2*x - 4
```

```
r = x^6 - 4*x^4 - 9*x^3 + 4*x^2 + 18*x + 20
```

Overloading Functions for the Polynom Class

MATLAB already has several functions for working with polynomials represented by coefficient vectors. They should be overloaded to also work with the new polynom object. In many cases, the overloading methods can simply apply the original function to the coefficient field.

Overloading roots for the Polynom Class

The method @polynom/roots.m finds the roots of polynom objects.

```
function r = roots(p)
% POLYNOM/ROOTS. ROOTS(p) is a vector containing the roots of p. r = roots(p.c);
```

The statement

```
roots(p)
```

results in

```
ans =
```

```
2.0946
```

```
-1.0473 + 1.1359i
```

```
-1.0473 - 1.1359i
```

Overloading polyval for the Polynom Class

The function polyval evaluates a polynomial at a given set of points. @polynom/polyval.m uses nested multiplication, or Horner's method to reduce the number of multiplication operations used to compute the various powers of x.

```
function y = polyval(p,x)
% POLYNOM/POLYVAL POLYVAL(p,x) evaluates p at the points x. y = 0;
for a = p.c
```

```
y = y.*x + a;
```

```
end
```

Overloading plot for the Polynom Class

The overloaded plotfunction uses both root and polyval. The function selects the domain of the independent variable to be slightly larger than an interval containing all real roots. Then polyval

is used to evaluate the polynomial at a few hundred points in the domain.

```
function plot(p)
% POLYNOM/PLOT PLOT(p) plots the polynom p.
r = max(abs(roots(p)));
x = (-1.1:0.01:1.1)*r;
y = polyval(p,x);
plot(x,y);
title(char(p))
grid on
```

Overloading diff for the Polynom Class

The method @polynom/diff.m differentiates a polynomial by reducing the degree by 1 and multiplying each coefficient by its original degree.

```
function q = diff(p)
% POLYNOM/DIFF DIFF(p) is the derivative of the polynom p. c = p.c;
d = length(c) - 1; % degree
q = polynom(p.c(1:d).*(d:-1:1));
```

Listing Class Methods

The function call

```
methods('class_name')
```

or its command form

```
methods class_name
```

shows all the methods available for a particular class. For thepolynom example, the output is
methods polynom

Methods for class polynom:

```
char display minus plot diff double mtimes plus polynom roots polyval subsref
```

Plotting the two polynom objectsx andp calls most of these methods.

```
x = polynom([1 0]);
p = polynom([1 0 -2 -5]);
plot(diff(p*p + 10*p + 20*x) - 20)
```

```
6*x^5 - 16*x^3 + 8*x^8
6
4
2
0
-2
-4
-6
-8
-2 -1.5 -1 -0.5 0 0.5 1 1.5 2
```

Building on Other Classes

A MATLAB object can *inherit* properties and behavior from another MATLAB object. When one object (the child) inherits from another (the parent), the child object includes all the fields of the parent object and can call the parent's methods. The parent methods can access those fields that a child object inherited from the parent class, but not fields new to the child class.

Inheritance is a key feature of object-oriented programming. It makes it easy to reuse code by allowing child objects to take advantage of code that exists for parent objects. Inheritance enables a child object to behave exactly like a parent object, which facilitates the development of related classes that behave similarly, but are implemented differently.

There are two kinds of inheritance:

- Simple inheritance, in which a child object inherits characteristics from one parent class.
- Multiple inheritance, in which a child object inherits characteristics from more than one parent class.

This section also discusses a related topic, *aggregation*. Aggregation allows one object to contain another object as one of its fields.

Simple Inheritance

A class that inherits attributes from a single parent class, and adds new attributes of its own, uses simple inheritance. Inheritance implies that objects belonging to the child class have the same fields as the parent class, as well as additional fields. Therefore, methods associated with the parent class can operate on objects belonging to the child class. The methods associated with the child class, however, cannot operate on objects belonging to the parent class. You cannot access the parent's fields directly from the child class; you must use access methods defined for the parent.

The constructor function for a class that inherits the behavior of another has two special characteristics:

- It calls the constructor function for the parent class to create the inherited fields.
- The calling syntax for the class function is slightly different, reflecting both the child class and the parent class.

The general syntax for establishing a simple inheritance relationship using the class function is
`child_obj = class(child_obj,'child_class',parent_obj);`

Simple inheritance can span more than one generation. If a parent class is itself an inherited class, the child object will automatically inherit from the grandparent class.

Visibility of Class Properties and Methods

The parent class does not have knowledge of the child properties or methods. The child class cannot access the parent properties directly, but must use parent access methods (e.g.,`get` or `subsref` method) to access the parent properties. From the child class methods, this access is accomplished via the parent field in the child structure. For example, when a constructor creates a child object,

```
c = class(c,'child_class_name',parent_object);
```

MATLAB automatically creates a field, `c.parent_class_name`, in the object's structure that contains the parent object. You could then have a statement in the child's display method that calls the parent's display method.

```
display(c.parent_class_name)
```

See "Designing the Stock Class" on page 22-45 for examples that use simple inheritance.

Multiple Inheritance

In the multiple inheritance case, a class of objects inherits attributes from more than one parent class. The child object gets fields from all the parent classes, as well as fields of its own.

Multiple inheritance can encompass more than one generation. For example, each of the parent objects could have inherited fields from multiple grandparent objects, and so on. Multiple inheritance is implemented in the constructors by calling `class` with more than three arguments.

```
obj = class(structure,'class_name',parent1,parent2,...)
```

You can append as many parent arguments as desired to the class input list.

Multiple parent classes can have associated methods of the same name. In this case, MATLAB calls the method associated with the parent that appears first in the `class` function call in the constructor function. There is no way to access subsequent parent function of this name.

Aggregation

In addition to standard inheritance, MATLAB objects support *containment* or *aggregation*. That is, one object can contain (embed) another object as one of its fields. For example, a rational object might use two `polynom` objects, one for the numerator and one for the denominator.

You can call a method for the contained object only from within a method for the outer object. When determining which version of a function to call, MATLAB considers only the outermost containing class of the objects passed as arguments; the classes of any contained objects are ignored.

See "Example - The Portfolio Container" on page 22-54 for an example of aggregation.

Example - Assets and Asset Subclasses

As an example of simple inheritance, consider a general asset class that can be used to represent any item that has monetary value. Some examples of an asset are: stocks, bonds, savings accounts, and any other piece of property. In designing this collection of classes, the `asset` class holds the data that is common to all of the specialized asset subclasses. The individual asset subclasses, such as the `stock` class, inherit the `asset` properties and contribute additional properties. The subclasses are "kinds of" assets.

Inheritance Model for the Asset Class

An example of a simple inheritance relationship using an asset parent class is shown in this diagram. Asset Class

Structure Fields: descriptor
date
current_value

Stock Class Bond Class

Inherited Fields: Inherited Fields: descriptor descriptor date date
current_value current_value

Stock Fields: Bond Fields: num_shares interest_rate share_price asset
asset
Savings Class

Inherited Fields: descriptor
date
current_value

Savings Fields: interest_rate asset

As shown in the diagram, the stock, bond, and savings classes inherit structure fields from theasset class. In this example, theasset class is used to provide storage for data common to all subclasses and to share asset methods with these subclasses. This example shows how to implement the asset and stock classes. Thebond andsavings classes can be implemented in a way that is very similar to thestock class, as would other types of asset subclasses.

Asset Class Design

The asset class provides storage and access for information common to all asset children. It is not intended to be instantiated directly, so it does not require an extensive set of methods. To serve its purpose, the class needs to contain the following methods:

- Constructor
- get andset
- subsref andsubsasgn
- display

Other Asset Methods

The asset class provides inherited data storage for its child classes, but is not instanced directly. The set, get, anddisplay methods provide access to the stored data. It is not necessary to implement the full complement of methods for asset objects (such as converters, end, andsubsindex) since only the child classes access the data.

The Asset Constructor Method

The asset class is based on a structure array with four fields:

- descriptor— Identifier of the particular asset (e.g., stock name, savings account number, etc.)
- date— The date the object was created (calculated by thedate command)

- `type`— The type of asset (e.g., savings, bond, stock)
- `current_value`— The current value of the asset (calculated from subclass data)

This information is common to asset child objects (stock, bond, and savings), so it is handled from the parent object to avoid having to define the same fields in each child class. This is particularly helpful as the number of child classes increases.

```
function a = asset(varargin)
% ASSET Constructor function for asset object % a = asset(descriptor, current_value)
switch nargin
case 0
% if no input arguments, create a default object

a.descriptor = 'none';
a.date = date;
a.type = 'none';
a.current_value = 0;
a = class(a,'asset');

case 1
% if single argument of class asset, return it if (isa(varargin{1},'asset'))
a = varargin{1};

else
error('Wrong argument type')
end
case 2
% create object using specified values
a.descriptor = varargin{1};
a.date = date;
a.type = varargin{2};
a.current_value = varargin{3};
a = class(a,'asset');
otherwise
error('Wrong number of input arguments')
end
```

The function uses a switch statement to accommodate three possible scenarios:

- Called with no arguments, the constructor returns a default asset object.
- Called with one argument that is an asset object, the object is simply returned.
- Called with two arguments (subclass descriptor, and current value), the constructor returns a new asset object.

The asset constructor method is not intended to be called directly; it is called from the child constructors since its purpose is to provide storage for common data.

The Asset get Method

The asset class needs methods to access the data contained in asset objects. The following function implements aget method for the class. It uses capitalized property names rather than literal field names to provide an interface similar to other MATLAB objects.

```
function val = get(a,prop_name)
% GET Get asset properties from the specified object % and return the value
switch prop_name
case 'Descriptor'
    val = a.descriptor;
case 'Date'
    val = a.date;
case 'CurrentValue'
    val = a.current_value;
otherwise
    error([prop_name,' Is not a valid asset property']) end
```

This function accepts an object and a property name and uses a switch statement to determine which field to access. This method is called by the subclassget methods when accessing the data in the inherited properties. See “The Stock get Method” on page 22-48 for an example.

The Asset set Method

The asset class set method is called by subclassset methods. This method accepts an asset object and variable length argument list of property name/ property value pairs and returns the modified object.

```
function a = set(a,varargin)
% SET Set asset properties and return the updated object property_argin = varargin;
while length(property_argin) >= 2,
    prop = property_argin{1};
    val = property_argin{2};
    property_argin = property_argin(3:end);
    switch prop
    case 'Descriptor'
        a.descriptor = val;
    case 'Date'
        a.date = val;
    case 'CurrentValue'
        a.current_value = val;
    otherwise
        error('Asset properties: Descriptor, Date, CurrentValue') end
end
```

Subclass set methods call the assetset method and require the capability to return the modified object since MATLAB does not support passing arguments by reference. See “The Stock set

Method” on page 22-48 for an example.

The Asset `subsref` Method

The `subsref` method provides access to the data contained in an asset object using one-based numeric indexing and structure field name indexing. The `outerswitch` statement determines if the index is a numeric or field name syntax. The `innerswitch` statements map the index to the appropriate value.

MATLAB calls `subsref` whenever you make a subscripted reference to an object (e.g., `A(i)`, `A{i}`, or `A.fieldname`).

```
function b = subsref(a,index)
%SUBSREF Define field name indexing for asset objects switch index.type
case '()'

switch index.subs{:}
case 1
b = a.descriptor;

case 2
b = a.date;
case 3
b = a.current_value;
otherwise
error('Index out of range')
end
case '.'
switch index.subs
case 'descriptor'
b = a.descriptor;
case 'date'
b = a.date;
case 'current_value'
b = a.current_value;
otherwise
error('Invalid field name')
end
case '{}'
error('Cell array indexing not supported by asset objects') end
```

See the “The Stock `subsref` Method” on page 22-49 for an example of how the child `subsref` method calls the `parentsubsref` method.

The Asset `subsasgn` Method

The `subsasgn` method is the assignment equivalent of the `subsref` method. This version enables you to change the data contained in an object using one-based numeric indexing and structure field name indexing. The outer switch statement determines if the index is a numeric or field

name syntax. The `innerswitch` statements map the index value to the appropriate value in the stock structure.

MATLAB calls `subsasgn` whenever you execute an assignment statement (e.g., `A(i) = val`, `A{i} = val`, or `A.fieldname = val`).

```
function a = subsasgn(a,index,val)
% SUBSASGN Define index assignment for asset objects switch index.type
case '()'%
switch index.subs{:}
case 1
a.descriptor = val;
case 2
a.date = val;
case 3
a.current_value = val;
otherwise
error('Index out of range') end
case ':'
switch index.subs
case 'descriptor'
a.descriptor = val;
case 'date'
a.date = val;
case 'current_value'
a.current_value = val; otherwise
error('Invalid field name') end
end
```

The `subsasgn` method enables you to assign values to the asset object data structure using two techniques. For example, suppose you have a child stock objects.

```
s = stock('XYZ',100,25);
```

Within stock class methods, you could change `thedescriptor` field with either of the following statements

```
s.asset(1) = 'ABC';
or
s.asset.descriptor = 'ABC';
```

See the “The Stock `subsasgn` Method” on page 22-51 for an example of how the child `subsasgn` method calls the `parentsubsasgn` method.

The Asset display Method

The asset display method is designed to be called from child-classdisplay methods. Its purpose is to display the data it stores for the child object. The method simply formats the data for display in a way that is consistent with the formatting of the child’s display method.

```

function display(a)

% DISPLAY(a) Display an asset object
stg = sprintf...
'Descriptor: %s\nDate: %s\nType: %s\nCurrent Value:%9.2f',...
a.descriptor,a.date,a.type,a.current_value);

disp(stg)

```

The stock class display method can now call this method to display the data stored in the parent class. This approach isolates the stock display method from changes to the asset class. See “The Stock display Method” on page 22-52 for an example of how this method is called.

The Asset fieldcount Method

The asset fieldcount method returns the number of fields in the asset object data structure. fieldcount enables asset child methods to determine the number of fields in the asset object during execution, rather than requiring the child methods to have knowledge of the asset class. This allows you to make changes to the number of fields in the asset class data structure without having to change child-class methods.

```

function num_fields = fieldcount(asset_obj)
% Determines the number of fields in an asset object % Used by asset child class methods
num_fields = length(fieldnames(asset_obj));

```

The struct function converts an object to its equivalent data structure, enabling access to the structure’s contents.

Designing the Stock Class

A stock object is designed to represent one particular asset in a person’s investment portfolio. This object contains two properties of its own and inherits three properties from its parent asset object.

Stock properties:

- NumberShares— The number of shares for the particular stock object.
- SharePrice— The value of each share.

Asset properties:

- Descriptor— The identifier of the particular asset (e.g., stock name, savings account number, etc.).
- Date— The date the object was created (calculated by the date command).
- CurrentValue— The current value of the asset.

Note that the property names are not actually the same as the field names of the structure array used internally by stock and asset objects. The property name interface is controlled by the stock and assetset and get methods and is designed to resemble the interface of other MATLAB object properties.

The asset field in the stock object structure contains the parent asset object and is used to access the inherited fields in the parent structure.

Stock Class Methods

The stock class implements the following methods:

- Constructor
- get andset
- subsref and subsasgn
- display

The Stock Constructor Method

The stock constructor creates a stock object from three input arguments:

- The stock name
- The number of shares
- The share price

The constructor must create an asset object from within the stock constructor to be able to specify it as a parent to the stock object. The stock constructor must, therefore, call the asset constructor. The class function, which is called to create the stock object, defines the asset object as the parent.

Keep in mind that the asset object is created in the temporary workspace of the stock constructor function and is stored as a field (.asset) in the stock structure. The stock object inherits the asset fields, but the asset object is not returned to the base workspace.

```
function s = stock(varargin)
% STOCK Stock class constructor.
% s = stock(descriptor, num_shares, share_price)
switch nargin
case 0
    % if no input arguments, create a default object

    s.num_shares = 0;
    s.share_price = 0;
    a = asset('none',0);
    s = class(s, 'stock', a);

case 1
    % if single argument of class stock, return it if (isa(varargin{1}, 'stock'))
    s = varargin{1};

else
    error('Input argument is not a stock object')
end
case 3
    % create object using specified values
    s.num_shares = varargin{2};
```

```

s.share_price = varargin{3};
a = asset(varargin{1}, 'stock', varargin{2} * varargin{3});
s = class(s, 'stock', a);
otherwise
error('Wrong number of input arguments')
end

```

Constructor Calling Syntax

The stock constructor method can be called in one of three ways:

- No Input Argument – If called with no arguments, the constructor returns a default object with empty fields.
- Input Argument is a Stock Object – If called with a single input argument that is a stock object, the constructor returns the input argument. A single argument that is not a stock object generates an error.
- Three Input Arguments – If there are three input arguments, the constructor uses them to define the stock object.

Otherwise, if none of the above three conditions are met, return an error. For example, this statement creates a stock object to record the ownership of 100 shares of XYZ corporation stocks with a price per share of 25 dollars. XYZ_stock = stock('XYZ', 100, 25);

The Stock get Method

The get method provides a way to access the data in the stock object using a “property name” style interface, similar to Handle Graphics. While in this example the property names are similar to the structure field name, they can be quite different. You could also choose to exclude certain fields from access via theget method or return the data from the same field for a variety of property names, if such behavior suits your design.

```

function val = get(s, prop_name)
% GET Get stock property from the specified object
% and return the value. Property names are: NumberShares % SharePrice, Descriptor, Date,
CurrentValue
switch prop_name
case 'NumberShares'

val = s.num_shares;
case 'SharePrice'
val = s.share_price;
case 'Descriptor'
val = get(s.asset, 'Descriptor'); % call asset get method
case 'Date'
val = get(s.asset, 'Date');
case 'CurrentValue'
val = get(s.asset, 'CurrentValue');
otherwise
error([prop_name, ', Is not a valid stock property']) end

```

Note that the asset object is accessed via the stock object's asset field (s.asset). MATLAB automatically creates this field when the class function is called with the parent argument.

The Stock set Method

The setmethod provides a “property name” interface like the get method. It is designed to update the number of shares, the share value, and the descriptor. The current value and the date are automatically updated.

```
function s = set(s,varargin)
% SET Set stock properties to the specified values
% and return the updated object
property_argin = varargin;
while length(property_argin) >= 2,
    prop = property_argin{1};
    val = property_argin{2};
    property_argin = property_argin(3:end);
    switch prop
        case 'NumberShares'
            s.num_shares = val;
        case 'SharePrice'
            s.share_price = val;
        case 'Descriptor'
            s.asset = set(s.asset,'Descriptor',val); otherwise
                error('Invalid property')
            end
        end
    s.asset = set(s.asset,'CurrentValue',...
        s.num_shares * s.share_price,'Date',date);
```

Note that this function creates and returns a new stock object with the new values, which you then copy over the old value. For example, given the stock object,

```
s = stock('XYZ',100,25);
the following set command updates the share price.
s = set(s,'SharePrice',36);
```

It is necessary to copy over the original stock object (i.e., assign the output to s) because MATLAB does not support passing arguments by reference. Hence the set method actually operates on a copy of the object.

The Stock subsref Method

The subsref method defines subscripted indexing for the stock class. In this example,subsref is implemented to enable numeric and structure field name indexing of stock objects.

```

function b = subsref(s,index)
% SUBSREF Define field name indexing for stock objects fc = fieldcount(s.asset);
switch index.type
case '()''

if (index.subs{:} <= fc)
b = subsref(s.asset,index);
else
switch index.subs{:} Hfc
case 1
b = s.num_shares;
case 2
b = s.share_price;
otherwise
error(['Index must be in the range 1 to ',num2str(fc + 2)]) end
end
case ':'
switch index.subs
case 'num_shares'
b = s.num_shares;
case 'share_price'
b = s.share_price;
otherwise
b = subsref(s.asset,index);
end
end

```

The outer switch statement determines if the index is a numeric or field name syntax.

The `fieldcount` asset method determines how many fields there are in the asset structure, and the `if` statement calls the `assetsubsref` method for indices 1 to `fieldcount`. See “The Asset `fieldcount` Method” on page 22-45 and “The Asset `subsref` Method” on page 22-42 for a description of these methods.

Numeric indices greater than the number returned by `fieldcount` are handled by the inner switch statement, which maps the index value to the appropriate field in the stock structure.

Field-name indexing assumes field names other than `num_shares` and `share_price` are asset fields, which eliminates the need for knowledge of asset fields by child methods. The `assetsubsref` method performs field-name error checking.

See `thesubsref` help entry for general information on implementing this method.

The Stock `subsasgn` Method

The `subsasgn` method enables you to change the data contained in a stock object using numeric indexing and structure field name indexing. MATLAB calls `subsasgn` whenever you execute an assignment statement (e.g., `A(i) = val`, `A{i} = val`, or `A.fieldname = val`).

```

function s = subsasgn(s,index,val)
% SUBSASGN Define index assignment for stock objects fc = fieldcount(s.asset);
switch index.type
case '()' 

if (index.subs{:} <= fc)
s.asset = subsasgn(s.asset,index,val);
else
switch index.subs{:}>fc
case 1
s.num_shares = val;
case 2
s.share_price = val;
otherwise
error(['Index must be in the range 1 to ',num2str(fc + 2)]) end
end
case ':'
switch index.subs
case 'num_shares'
s.num_shares = val;
case 'share_price'
s.share_price = val;
otherwise
s.asset = subsasgn(s.asset,index,val);
end
end

```

The outer switch statement determines if the index is a numeric or field name syntax. The fieldcount asset method determines how many fields there are in the asset structure and the if statement calls the assetsubsasgn method for indices1 to fieldcount. See “The Asset fieldcount Method” on page 22-45 and “The Asset subsasgn Method” on page 22-43 for a description of these methods.

Numeric indices greater than the number returned by fieldcount are handled by the inner switch statement, which maps the index value to the appropriate field in the stock structure.

Field-name indexing assumes field names other than num_shares and share_price are asset fields, which eliminates the need for knowledge of asset fields by child methods. The assetsubsasgn method performs field-name error checking.

The subsasgn method enables you to assign values to stock object data structure using two techniques. For example, suppose you have a stock object `s = stock('XYZ',100,25)`

You could change the descriptor field with either of the following statements `s(1) = 'ABC'`; or
`s.descriptor = 'ABC';`

See `thesubsasgn` help entry for general information on assignment statements in MATLAB.

The Stock display Method

When you issue the statement (without terminating with a semicolon) `XYZStock = stock('XYZ',100,25)`

MATLAB looks for a method in the `@stock` directory called `display`. The `display` method for the `stock` class produces this output.

```
Descriptor: XYZ
Date: 17-Nov-1998
Type: stock
Current Value: 2500.00
Number of shares: 100
Share price: 25.00
```

Here is the `stockdisplay` method.

```
function display(s)
% DISPLAY(s) Display a stock object
display(s.asset)
stg = sprintf('Number of shares: %g\nShare price: %3.2f\n',...
s.num_shares,s.share_price);
disp(stg)
```

First, the parent asset object is passed to the `assetdisplay` method to display its fields (MATLAB calls the `assetdisplay` method because the input argument is an asset object). The `stock` object's fields are displayed in a similar way using a formatted text string.

Note that if you did not implement a `stock` class `display` method, MATLAB would call the `assetdisplay` method. This would work, but would display only the descriptor, date, type, and current value.

Example - The Portfolio Container

Aggregation is the containment of one class by another class. The basic relationship is: each contained class “is a part of” the container class.

For example, consider a financial portfolio class as a container for a set of assets (stocks, bonds, savings, etc.). Once the individual assets are grouped, they can be analyzed, and useful information can be returned. The contained objects are not accessible directly, but only via the portfolio class methods.

See “Example - Assets and Asset Subclasses” on page 22-38 for information about the assets collected by this portfolio class.

Designing the Portfolio Class

The portfolio class is designed to contain the various assets owned by a given individual and

provide information about the status of his or her investment portfolio. This example implements a somewhat over-simplified portfolio class that:

- Contains an individual's assets
- Displays information about the portfolio contents
- Displays a 3-D pie chart showing the relative mix of asset types in the portfolio

Required Portfolio Methods

The portfolio class implements only three methods:

- `portfolio`— The portfolio constructor.
- `display`— Displays information about the portfolio contents.
- `pie3`— Overloaded version of `pie3` function designed to take a single portfolio object as an argument.

Since a portfolio object contains other objects, the portfolio class methods can use the methods of the contained objects. For example, the portfolio display method calls the stock class `display` method, and so on.

The Portfolio Constructor Method

The portfolio constructor method takes as input arguments a client's name and a variable length list of asset subclass objects (stock, bond, and savings objects in this example). The portfolio object uses a structure array with the following fields:

- `name`— The client's name.
- `ind_assets`— The array of asset subclass objects (stock, bond, savings).
- `total_value`— The total value of all assets. The constructor calculates this value from the objects passed in as arguments.
- `account_number`— The account number. This field is assigned a value only

when you save a portfolio object (see “Saving and Loading Objects” on page 22-60).

```
function p = portfolio(name,varargin)
% PORTFOLIO Create a portfolio object containing the % client's name and a list of assets
switch nargin
case 0

% if no input arguments, create a default object p.name = 'none';
p.total_value = 0;
p.ind_assets = {};
p.account_number = "";
p = class(p,'portfolio');

case 1
% if single argument of class portfolio, return it if isa(name,'portfolio')
```

```

p = name;
else
    disp(['inputname(1) is not a portfolio object'])
    return
end
otherwise
% create object using specified arguments
p.name = name;
p.total_value = 0;
for k = 1:length(varargin)
    p.ind_assets(k) = varargin{k};
    asset_value = get(p.ind_assets{k}, 'CurrentValue');
    p.total_value = p.total_value + asset_value;
end
p.account_number = "";
p = class(p, 'portfolio');
end

```

Constructor Calling Syntax

The portfolio constructor method can be called in one of three different ways:

- No input arguments – If called with no arguments, it returns an object with empty fields.
- Input argument is an object – If the input argument is already a portfolio object, MATLAB returns the input argument. Theisa function checks for this case.
- More than two input arguments – If there are more than two input arguments, the constructor assumes the first is the client's name and the rest are asset subclass objects. A more thorough implementation would perform more careful input argument checking, for example, using theisa function to determine if the arguments are the correct class of objects.

The Portfolio display Method

The portfolio display method lists the contents of each contained object by calling the object's display method. It then lists the client name and total asset value.

```

function display(p)
% DISPLAY Display a portfolio object
for k = 1:length(p.ind_assets)

    display(p.ind_assets{k})
end
stg = sprintf('\nAssets for Client: %s\nTotal Value: %9.2f\n', ... p.name, p.total_value);
disp(stg)

```

The Portfolio pie3 Method

The portfolio class overloads the MATLAB pie3 function to accept a portfolio object and display a 3-D pie chart illustrating the relative asset mix of the client's portfolio. MATLAB calls the @portfolio/pie3.m version of pie3 whenever the input argument is a single portfolio object.

```

function pie3(p)
% PIE3 Create a 3-D pie chart of a portfolio
stock_amt = 0; bond_amt = 0; savings_amt = 0;
for k = 1:length(p.ind_assets)

if isa(p.ind_assets{k},'stock')
stock_amt = stock_amt + ...
get(p.ind_assets{k},'CurrentValue');
elseif isa(p.ind_assets{k},'bond')
bond_amt = bond_amt + ...
get(p.ind_assets{k},'CurrentValue');
elseif isa(p.ind_assets{k},'savings')
savings_amt = savings_amt + ...
get(p.ind_assets{k},'CurrentValue');

end
end
i = 1;
if stock_amt ~= 0

label(i) = {'Stocks'};
pie_vector(i) = stock_amt;
i = i +1;

end
if bond_amt ~= 0
label(i) = {'Bonds'};
pie_vector(i) = bond_amt;
i = i +1;
end
if savings_amt ~= 0
label(i) = {'Savings'};
pie_vector(i) = savings_amt;
end
pie3(pie_vector,label)
set(gcf,'Renderer','zbuffer')
set(findobj(gca,'Type','Text'),'FontSize',14)
cm = gray(64);
colormap(cm(48:end,:))
stg(1) = {[['Portfolio Composition for ',p.name]};
stg(2) = {[['Total Value of Assets: ',num2str(p.total_value)]}]; title(stg,'FontSize',12)

```

There are three parts in the overloadedpie3 method.

- The first uses the asset subclassget methods to access theCurrentValue property of each contained object. The total value of each class is summed.
- The second part creates the pie chart labels and builds a vector of graph data, depending on

which objects are present.

- The third part calls the MATLABpie3 function, makes some font and colormap adjustments, and adds a title.

Creating a Portfolio

Suppose you have implemented a collection of asset subclasses in a manner similar to the stock class. You can then use a portfolio object to present the individual's financial portfolio. For example, given the following assets

```
XYZStock = stock('XYZ',200,12);
SaveAccount = savings('Acc # 1234',2000,3.2);
Bonds = bond('U.S. Treasury',1600,12);
```

Create a portfolio object.

```
p = portfolio('Gilbert Bates',XYZStock,SaveAccount,Bonds)
```

The portfoliodisplay method summarizes the portfolio contents (because this statement is not terminated by a semicolon).

Descriptor: XYZ

Date: 24-Nov-1998

Current Value: 2400.00

Type: stock

Number of shares: 200

Share price: 12.00

Descriptor: Acc # 1234

Date: 24-Nov-1998

Current Value: 2000.00

Type: savings

Interest Rate: 3.2%

Descriptor: U.S. Treasury

Date: 24-Nov-1998

Current Value: 1600.00

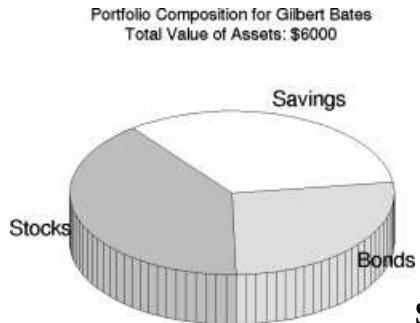
Type: bond

Interest Rate: 12%

Assets for Client: Gilbert Bates

Total Value: 6000.00

The portfoliopic3 method displays the relative mix of assets using a pie chart. pie3(p)



Saving and Loading Objects

You can use the MATLAB save and load commands to save and retrieve user-defined objects to and from .mat files, just like any other variables.

When you load objects, MATLAB calls the object's class constructor to register the object in the workspace. The constructor function for the object class you are loading must be able to be called with no input arguments and return a default object. See "Guidelines for Writing a Constructor" on page 22-10 for more information.

Modifying Objects During Save or Load

When you issue a save or load command on objects, MATLAB looks for class methods called saveobj and loadobj in the class directory. You can overload these methods to modify the object before the save or load operation. For example, you could define a saveobj method that saves related data along with the object or you could write a loadobj method that updates objects to a newer version when this type of object is loaded into the MATLAB workspace.

Example - Defining saveobj and loadobj for Portfolio

In the section "Example - The Portfolio Container" on page 22-54, portfolio objects are used to collect information about a client's investment portfolio. Now suppose you decide to add an account number to each portfolio object that is saved. You can define a portfolio saveobj method to carry out this task automatically during the save operation.

Suppose further that you have already saved a number of portfolio objects without the account number. You want to update these objects during the load operation so that they are still valid portfolio objects. You can do this by defining a loadobj method for the portfolio class.

Summary of Code Changes

To implement the account number scenario, you need to add or change the following functions:

- portfolio – The portfolio constructor method needs to be modified to create a new field, account_number, which is initialized to the empty string when an object is created.
- saveobj – A new portfolio method designed to add an account number to a portfolio object during the save operation, only if the object does not already have one.
- loadobj – A new portfolio method designed to update older versions of portfolio objects that were saved before the account number structure field was added.
- subsref – A new portfolio method that enables subscripted reference to portfolio objects outside of a portfolio method.
- getAccountNumber – a MATLAB function that returns an account number that consists of the

first three letters of the client's name.

New Portfolio Class Behavior

With the additions and changes made in this example, the portfolio class now:

- Includes a field for an account number
- Adds the account number when a portfolio object is saved for the first time
- Automatically updates the older version of portfolio objects when you load them into the MATLAB workspace

The saveobj Method

MATLAB looks for the portfolio saveobj method whenever the save command is passed a portfolio object. If @portfolio/saveobj exists, MATLAB passes the portfolio object to saveobj, which must then return the modified object as an output argument. The following implementation of saveobj determines if the object has already been assigned an account number from a previous save operation. If not, saveobj calls getAccountNumber to obtain the number and assigns it to the account_number field.

```
function b = saveobj(a)
if isempty(a.account_number)
a.account_number = getAccountNumber(a);
end
b = a;
```

The loadobj Method

MATLAB looks for the portfolio loadobj method whenever the load command detects portfolio objects in the .mat file being loaded. If loadobj exists, MATLAB passes the portfolio object to loadobj, which must then return the modified object as an output argument. The output argument is then loaded into the workspace.

If the input object does not match the current definition as specified by the constructor function, then MATLAB converts it to a structure containing the same fields and the object's structure with all the values intact (that is, you now have a structure, not an object).

The following implementation of loadobj first uses isa to determine whether the input argument is a portfolio object or a structure. If the input is an object, it is simply returned since no modifications are necessary. If the input argument has been converted to a structure by MATLAB, then the new account_number field is added to the structure and is used to create an updated portfolio object.

```
function b = loadobj(a)
% loadobj for portfolio class
if isa(a,'portfolio')
b = a;
else % a is an old version
a.account_number = getAccountNumber(a); b = class(a,'portfolio');
```

```
end
```

Changing the Portfolio Constructor

The portfolio structure array needs an additional field to accommodate the account number. To create this field, add the line

```
p.account_number = ";  
to@portfolio/portfolio.m in both the zero argument and variable argument sections.
```

The getAccountNumber Function

In this example, `getAccountNumber` is a MATLAB function that returns an account number composed of the first three letters of the client name prepended to a series of digits. To illustrate implementation techniques, `getAccountNumber` is not a portfolio method so it cannot access the portfolio object data directly. Therefore, it is necessary to define a `portfoliosubsref` method that enables access to the name field in a portfolio object's structure.

For this example, `getAccountNumber` simply generates a random number, which is formatted and concatenated with elements 1 to 3 from the portfolio name field.

```
function n = getAccountNumber(p)  
% provides a account number for object p  
n = [upper(p.name(1:3)) strcat(num2str(round(rand(1,7)*10)))'];
```

Note that the portfolio object is indexed by field name, and then by numerical subscript to extract the first three letters. The `subsref` method must be written to support this form of subscripted reference.

The Portfolio subsref Method

When MATLAB encounters a subscripted reference, such as that made in the `getAccountNumber` function

```
p.name(1:3)
```

MATLAB calls the portfolio `subsref` method to interpret the reference. If you do not define a `subsref` method, the above statement is undefined for portfolio objects (recall that here `p` is an object, not just a structure).

The `portfoliosubsref` method must support field-name and numeric indexing for the `getAccountNumber` function to access the portfolio name field.

```
function b = subsref(p,index)  
% SUBSREF Define field name indexing for portfolio objects switch index(1).type  
case '!'  
  
switch index(1).subs  
case 'name'  
if length(index)== 1  
  
b = p.name;  
else
```

```

switch index(2).type
case '()'
b = p.name(index(2).subs{:});
end
end
end
end

```

Note that the portfolio implementation of `subsref` is designed to provide access to specific elements of the `name` field; it is not a general implementation that provides access to all structure data, such as the stock class implementation of `subsref`.

See `thesubsref` help entry for more information about indexing and objects. **Object Precedence**

Object Precedence is a means to resolve the question of which of possibly many versions of an operator or function to call in a given situation. Object precedence enables you to control the behavior of expressions containing different classes of objects. For example, consider the expression

objectA + *objectB*

Ordinarily, MATLAB assumes that the objects have equal precedence and calls the method associated with the leftmost object. However, there are two exceptions:

- User-defined classes have precedence over MATLAB built-in classes.
- User-defined classes can specify their relative precedence with respect to other user-defined classes using `theinferior` and `superior` functions.

For example, in the section “Example - A Polynomial Class” on page 22-24 the `polynom` class defines `aplus` method that enables addition of `polynom` objects. Given the `polynom` object *p*

```

p = polynom([1 0 1 2 1 5])
p =
x^3*x^5 The expression, 1 + p
ans =
x^3*x^4

```

calls the `polynom plus` method (which converts the double, 1, to a `polynom` object, and then adds it top). The user-defined `polynom` class has precedence over the MATLAB double class.

Specifying Precedence of User-Defined Classes

You can specify the relative precedence of user-defined classes by calling the `inferior` or `superior` function in the class constructor.

The `inferior` function places a class below other classes in the precedence hierarchy. The calling syntax for the `inferior` function is

`inferior('class1','class2',...)`

You can specify multiple classes in the argument list, placing the class below many other classes

in the hierarchy.

Similarly, the `superioro` function places a class above other classes in the precedence hierarchy. The calling syntax for the `superioro` function is `superioro('class1','class2',...)`

Location in the Hierarchy

If `objectA` is above `objectB` in the precedence hierarchy, then the expression `objectA + objectB` calls `@classA/plus.m`. Conversely, if `objectB` is above `objectA` in the precedence hierarchy, then MATLAB calls `@classB/plus.m`.

See “How MATLAB Determines Which Method to Call” on page 22-67 for related information.

How MATLAB Determines Which Method to Call

In MATLAB, functions exist in directories in the computer’s file system. A directory may contain many functions (M-files). Function names are unique only within a single directory (e.g., more than one directory may contain a function called `pie3`). When you type a function name on the command line, MATLAB must search all the directories it is aware of to determine which function to call. This list of directories is called the *MATLAB path*.

When looking for a function, MATLAB searches the directories in the order they are listed in the path, and calls the first function whose name matches the name of the specified function.

If you write an M-file called `pie3.m` and put it in a directory that is searched before the `specgraph` directory that contains MATLAB’s `pie3` function, then MATLAB uses your `pie3` function instead (note that this is not true for built-in functions like `plot`, which are always found first).

Object-oriented programming allows you to have many methods (MATLAB functions located in class directories) with the same name and enables MATLAB to determine which method to use based on the type or class of the variables passed to the function. For example, if `p` is a portfolio object, then

`pie3(p)`

calls `@portfolio/pie3.m` because the argument is a portfolio object.

Selecting a Method

When you call a method for which there are multiple versions with the same name, MATLAB determines the method to call by:

- Looking at the classes of the objects in the argument list to determine which argument has the highest object precedence; the class of this object controls the method selection and is called the *dispatch type*.
- Applying the *function precedence order* to determine which of possibly several implementations of a method to call. This order is determined by the location and type of function.

Determining the Dispatch Type

MATLAB first determines which argument controls the method selection. The class type of this argument then determines the class in which MATLAB searches for the method. The controlling argument is either:

- The argument with the highest precedence, or
- The leftmost of arguments having equal precedence

User-defined objects take precedence over MATLAB's built-in classes such as double or char. You can set the relative precedence of user-defined objects with the `inferior` and `superior` functions, as described in "Object Precedence" on page 22-65.

MATLAB searches for functions by name. When you call a function, MATLAB knows the name, number of arguments, and the type of each argument. MATLAB uses the dispatch type to choose among multiple functions of the same name, but does not consider the number of arguments.

Function Precedence Order

The function precedence order determines the precedence of one function over another based on the type of function and its location on the MATLAB path. From the perspective of method selection, MATLAB contains two types of functions: those built into MATLAB, and those written as M-files. MATLAB treats these types differently when determining the function precedence order.

MATLAB selects the correct function for a given context by applying the following function precedence rules, in the order given.

For built-in functions:

1 Overloaded Methods

If there is a method in the class directory of the dispatching argument that has the same name as a MATLAB built-in function, then this method is called instead of the built-in function.

2 Nonoverloaded MATLAB Functions

If there is no overloaded method, then the MATLAB built-in function is called.

MATLAB built-in functions take precedence over both subfunctions and private functions. Therefore, subfunctions or private functions with the same name as MATLAB built-in functions can never be called.

For nonbuilt-in functions:

1 Subfunctions

Subfunctions take precedence over all other M-file functions and overloaded methods that are on the path and have the same name. Even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the subfunction and ignores the overloaded method.

2 Private Functions

Private functions are called if there is no subfunction of the same name within the current scope. As with subfunctions, even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the private function and ignores the overloaded method.

3 Class Constructor Functions

Constructor functions (functions having names that are the same as the @ directory, for example @polynom/polynom.m) take precedence over other MATLAB functions. Therefore, if you create an M-file calledpolynom.m and put it on your path before the constructor@polynom/polynom.m version, MATLAB will always call the constructor version.

4 Overloaded Methods

MATLAB calls an overloaded method if it is not masked by a subfunction or private function.

5 Current Directory

A function in the current working directory is selected before one elsewhere on the path.

6 Elsewhere On Path

Finally, a function anywhere else on the path is selected.

Selecting Methods from Multiple Directories

There may be a number of directories on the path that contain methods with the same name. MATLAB stops searching when it finds the first implementation of the method on the path, regardless of the implementation type (MEX-file, P-code, M-file).

Selecting Methods from Multiple Implementation Types

There are four file precedence types. MATLAB uses file precedence to select between identically named functions in the same directory. The order of precedence for file types is:

1 MEX-files

2 MDL-file (Simulink model)

3 P-code

4 M-file

For example, if MATLAB finds a P-code and an M-file version of a method in a class directory, then the P-code version is used. It is, therefore, important to regenerate the P-code version whenever you edit the M-file.

Querying Which Method MATLAB Will Call

You can determine which method MATLAB will call using thewhichcommand. For example,

`which pie3`

`your_matlab_path/toolbox/matlab/specgraph/pie3.m`

However, ifp is a portfolio object,

`which pie3(p)`

`dir_on_your_path/@portfolio/pie3.m % portfolio method`

The whichcommand determines which version ofpie3MATLAB will call if you passed a portfolio object as the input argument. To see a list of all versions of a particular function that are on your MATLAB path, use theAll option. See the which reference page for more information on this command.

A

External Interfaces and the MATLAB API

Reference Documentation A-4

Finding the Documentation in Online Help

MATLAB provides interface capabilities that allow you to communicate between MATLAB and the following programs and devices:

- External C and Fortran programs
- Object-oriented technologies like Java and ActiveX
- Hardware devices on your computer's serial port

You can also import and export data to and from MATLAB.

These interfaces, also referred to as the MATLAB Application Program Interface (API), are documented in full in the online help. Use the following path to locate the help sections listed below.

MATLAB -> Using MATLAB -> External Interfaces/API

Calling C and Fortran Programs from MATLAB

MATLAB provides an interface to external programs written in the C and Fortran languages that enables you to interact with data and programs external to the MATLAB environment. This section explains how to call your own C or Fortran subroutines from MATLAB as if they were built-in functions.

Creating C Language MEX-Files

MATLAB callable C and Fortran programs are referred to as MEX-files. This section explains how to create and work with C MEX-files.

Creating Fortran MEX-Files

This section explains how to create and work with Fortran MEX-files.

Calling MATLAB from C and Fortran Programs

You can employ MATLAB as a computational engine that responds to calls from your C and Fortran programs. This section describes the MATLAB functions that allow you to:

- Start and end a MATLAB process
- Send commands to and exchange data with MATLAB
- Compile and link MATLAB engine programs

Calling Java from MATLAB

This section describes how to use the MATLAB interface to Java classes and objects. This MATLAB capability enables you to:

- Bring Java classes into the MATLAB environment
- Construct objects from those classes
- Work with Java arrays in MATLAB
- Call methods on Java objects, passing MATLAB or Java data types

Importing and Exporting Data

This section describes how to use MAT-files to import data to and export data from the MATLAB environment. MAT-files provide a convenient mechanism for moving your MATLAB data between different platforms in a highly portable manner. In addition, they provide a means to import and export your data to other stand-alone MATLAB applications.

ActiveX and DDE Support

MATLAB has interfaces that allow you to interact with ActiveX and Dynamic Data Exchange, (DDE). This section explains how to:

- Integrate ActiveX control components into an ActiveX control container such as a MATLAB figure window
- Use ActiveX Automation components to control or be controlled by MATLAB
- Enable access between MATLAB and other Windows applications using DDE

Serial Port I/O

This section describes the MATLAB serial port interface which provides direct access to peripheral devices such as modems, printers, and scientific instruments that you connect to your computer's serial port. This interface is established through a serial port object that allows you to:

- Configure serial port communications
- Use serial port control pins
- Write data to and read data from the device
- Execute an action when a particular event occurs
- Create a record of your serial port session

Reference Documentation

The online Help Reference section provides a detailed description of each of the MATLAB functions available in the MATLAB external interfaces. You can find this documentation in the online Help Contents by following the path shown here. Click on the page called External Interfaces/API Reference to see links to the sections listed below.

MATLAB -> Reference -> External Interfaces/API Reference

API Notes

An introduction to using the mex script, the MATLAB mxArray and other data types, and passing pointers in Fortran.

C Engine Routines

Functions that allow you to call MATLAB from your own C programs.

C MAT-File Routines

Functions that allow you to incorporate and use MATLAB data in your own C programs.

C MEX-Functions

Functions that you use in your C files to perform operations in the MATLAB environment.

C MX-Functions

Array access and creation functions that you use in your C files to manipulate MATLAB arrays.

Fortran Engine Routines

Functions that allow you to call MATLAB from your own Fortran programs.

Fortran MAT-File Routines

Functions that allow you to incorporate and use MATLAB data in your own Fortran programs.

Fortran MEX-Functions

Functions that you use in your Fortran files to perform operations in the MATLAB environment.
Reference Documentation

Fortran MX-Functions

Array access and creation functions that you use in your Fortran files to manipulate MATLAB arrays.

J ava Interface Functions

Functions that enable you to create and interact with Java classes and objects from MATLAB.

ActiveX Functions

Functions that create ActiveX objects and manipulate their interfaces.

DDE Functions

Dynamic Data Exchange functions that enable MATLAB to access other Windows applications and vice versa.

Serial P ort I/O Functions

Functions that enable you to interact with devices connected to your computer's serial port.

Symbols

! function 3-12 %

for comment lines 17-11

for H1 and help text lines 17-10 & 17-29

... in functions 3-5

../ref/#performance 8-2

; after functions 3-9

>> prompt in Command Window 3-2

@ 21-1

| 17-29

~ 17-29

~= 17-28

A

absolute accuracy

BVP 15-78

ODE 15-19

accelerators, keyboard 2-18
access modes
HDF files 6-34
Access program Web page 2-20 accuracy of calculations 17-21 ActiveX
entries for MATLAB 1-6
startup options for MATLAB 1-6 adding
cells to cell arrays 20-21
fields to structure arrays 20-10 additional parameters
BVP example 15-68, 15-72
ODE example 15-13
addpath 5-17
adjacency matrix 16-17
and graphing 16-17
Bucky ball 16-18
defined 16-17
distance between nodes 16-22
node 16-17
numbering nodes 16-19
aggregation 22-37
airflow modeling 16-23
amp1dae demo 15-5
analytical partial derivatives (BVP) 15-79 AND operator, rules for evaluating 17-30 ans 17-21
answer, assigned toans 17-21
arguments
checking number of 17-14
defined for function 17-9
order
varargin, varargout 17-16, 17-18
passing 17-14
passing by reference 17-14
passing by value 17-14
passing variable number 17-16 arithmetic expressions 17-26
arithmetic operators
overloading 22-29
using 17-26
Array Editor 5-10
preferences 5-12
arrays
cell array of strings 18-8
character 18-5
dimensions
inverse permutation 19-13
editing 5-10
indexing 17-51
multidimensional 19-3
numeric

converting to cell array 20-30
of strings 18-6
storage 17-51
workspace 5-3
arrow keys for editing commands 3-8
ASCII data
definition of 6-2
exporting 6-16, 6-18
exporting delimited data 6-17
formats 6-4
importing 6-4
importing delimited files 6-12
importing mixed alphabetic and numeric data 6-14
importing space-delimited data 6-11 saving 6-17
table of format samples 6-10
using the Import Wizard with 6-4
with text headers 6-10, 6-13
ASCII files
reading formatted text 6-59
specifying delimiter 6-12
viewing contents of 5-27
writing 6-61
assignment statements
building structure arrays with 20-5
local and global variables 17-19
attributes
HDF files 6-35, 6-47
Audio/Video Interleave format
saving graphs 6-27
autoinit cells
converting input cells to 10-26
converting to input cell 10-27
defining 10-10
AutoInit style
definition of 10-19
automatic scalar expansion 17-27
automation startup option (automation server) 1-6
AVI. *See* Audio/Video Interleave format

B

ballode demo 15-41
bandwidth of sparse matrix, reducing 16-29 bang (!) function 3-12
base (numeric), converting 18-15
base date 17-63
base number conversion 18-4
base workspace 5-8
Basic Fitting interface 13-31

batch mode for starting MATLAB 1-6 batonode demo 15-5
bicubic interpolation 12-13
bilinear interpolation 12-13
binary data

exporting 6-25
importing 6-20
using the Import Wizard 6-20

binary files
controlling data type of values read 6-54 writing to 6-55

binary from decimal conversion 18-15 blank spaces in MATLAB commands 3-4 blanks

finding in string arrays 18-12
removing from strings 18-7
bookmarks in Help browser 4-15
boundary conditions
BVP 15-59
BVP example 15-65
PDE 15-84
PDE example 15-89
Boundary Value Problems. *See* BVP
break long lines 3-5
breakpoints
clearing 7-30
description 7-18
setting 7-22
Bring MATLAB to Front command 10-26 Brusselator system (ODE example) 15-38 brussode
demo 15-38
Buckminster Fuller dome 16-18
Bucky ball 16-18
bugs, reporting to The MathWorks 4-32 burgersode demo 15-5
BVP 15-57
defined 15-59
rewriting problem as first-order system 15-64 BVP solver 15-60
additional known parameters 15-63
basic syntax 15-61
evaluate solution at specific points 15-68 examples
boundary condition at infinity (shockbvp) 15-72
Mathieu's Equation (mat4bvp) 15-63 rapid solution changes (shockbvp) 15-68 initial guess 15-68
performance 15-75
representing problems 15-63
unknown parameters 15-67
BVP solver properties 15-76
analytical partial derivatives 15-79

bvpset function 15-76
error tolerance 15-77
mesh 15-80
modifying property structure 15-77
querying property structure 15-77
solution statistics 15-81
vectorization 15-78

C

C++ and MATLAB OOP 22-8
caching

MATLAB directory 5-14, 17-6
MATLAB/toolbox directory 17-6
M-files 5-14
toolbox path 1-10

calc zones
defining 10-10
ensuring workspace consistency in M-books

10-7
evaluating 10-14
output from 10-15

calling context 17-14
calling MATLAB functions
compiling for later use 17-13
how MATLAB searches for functions 17-12
storing as pseudocode 17-13
canonical class 22-9
capitalization in MATLAB 3-4
case 17-37
case conversion 18-3
case sensitivity in MATLAB 3-4
cat 16-26, 19-2, 19-7, 19-18
catch
for error handling 17-59
cell
indexing 20-20
accessing a subset of cells 20-24
cell arrays 20-19
accessing a subset of cells 20-24
accessing data 20-23
applying functions to 20-27
cell indexing 20-20
concatenating 20-22
content indexing 20-21

converting to numeric array 20-30 creating 20-20
using assignments 20-20
`withcells` function 20-23
with curly braces 20-22
deleting cells 20-24
deleting dimensions 20-24
displaying 20-21
expanding 20-21
flat 20-29
indexing 20-21
multidimensional 19-18
nested 20-29
building with `thecells` function 20-29
indexing 20-30
of strings 18-8
comparing strings 18-11
of structures 20-31
organizing data 20-27
overview 20-19
preallocating 17-73, 20-23
replacing comma-separated list with 20-25 reshaping 20-25
visualizing 20-21
cell data type 17-24
cell groups
converting to input cells 10-32
creating 10-9
definition of 10-8, 10-28
evaluating 10-13
output from 10-13
cell markers
defined 10-8
hiding 10-31
printing 10-18

`celldisp` 20-21
`cellplot` 20-21
cells
building nested arrays with 20-29
preallocating empty arrays with 20-20, 20-23 char data type 6-55, 17-24
character arrays 18-5
categorizing characters of 18-12
comparing 18-10
comparing values on cell arrays 18-11 concatenating 17-54
conversion 18-3, 18-15
converting to cell arrays 18-8
creating 18-5
delimiting character 18-13

evaluating 17-54, 18-16
finding a substring 18-13
functions that test 18-2
in cell arrays 18-8
operations 18-3
padding for equal row length 18-6
removing trailing blanks 18-7
representation 18-5
scalar 18-11
searching and replacing 18-13
token 18-13
two-dimensional 18-6
using relational operators on 18-11
characteristic polynomial of matrix 12-5 characteristic roots of matrix 12-5
characters
corresponding ASCII values 18-7
finding in string 18-12
characters per line, maximum 3-5
checkin 9-8
checking in files 9-7
checking out files 9-9
undoing 9-10
checkout 9-10
chol 16-26, 16-33
Cholesky factorization 11-25
for sparse matrices 16-33
class directories 22-7
class function 22-11
classes 17-22
clearing definition 22-7
constructor method 22-10
debugging 22-6
designing 22-9
Java 17-24
methods required by MATLAB 22-9 object-oriented methods 22-3
overview 22-3
clc 3-9
clear 5-8, 17-13, 17-74
ClearCase view 9-6
clearing
Command Window 3-9
clicking on multiple items 2-19
clipboard 2-20
importing binary data 6-20
closest point searches
Delaunay triangulation 12-25

closing
desktop tools 2-9
MATLAB 1-15
closing files 6-52, 6-62
cmopts 9-5
colamd 16-29
Collatz problem 7-18
colmmd 16-29, 16-31
colon operator 17-26
for multidimensional array subscripting 19-9 indexing a page with 19-15
scalar expansion with 19-5
to access subsets of cells 20-24
color
general preferences 2-24
color modes
for printing M-books 10-23
color printing
of an M-book 10-18
colors
Command Window preferences 3-16 indicators for syntax 3-5
colperm 16-28
column vector
indexing as 17-51
comma to separate function arguments 17-9 command flags 1-5
Command History
copying entries from 3-19
deleting entries in 3-18
description 3-17
running functions from 3-18
command line editing 3-6
Command Window
bringing to the front in Notebook 10-26 clearing 3-9
description 3-2
editing in 3-6
help 4-3
paging of output in 3-10
preferences for 3-13
printing contents of 3-11
prompt 3-2
commands
on multiple lines 3-5
running 3-2
to operating system 3-12
See also functions
comma-separated list
using cell arrays 20-25

comments
color indicators 2-24
comments in code 17-11
comparing
sparse and full matrix storage 16-7 strings 18-10
complex conjugate transpose operator 17-26 complex values in sparse matrix 16-7 computational functions
applying to cell arrays 20-27
applying to multidimensional arrays 19-14 applying to sparse matrices 16-25 applying to structure fields 20-10
in M-file 17-4
computational geometry
multidimensional 12-27
two-dimensional 12-19
computer 17-21
computer type 17-21
concatenating
cell arrays 20-22
matrices 17-50
strings 17-54
condest 16-4
conditional statements 17-14
configuration management. *See* source control system
configuration, desktop 2-7
configuring Notebook 10-24
console mode 3-15
constructor methods 22-10
guidelines 22-10
usingclass in 22-11
containment 22-37
content indexing 20-21
to access cell contents 20-23
content of M-files, searching 5-28
contents of sparse matrix 16-13
Contents tab in Help browser
description 4-9
synchronizing preference 4-23
synchronizing with display 4-10 Contents.m file 17-5
context menus 2-17
continue 17-41
continue long statements 3-5
control keys for editing commands 3-8 conv 20-26
converter methods 22-20, 22-25
converting
base numbers 18-4
cases of strings 18-3

dates 17-62
numbers 18-15
strings 18-3, 18-15
converting Word document to M-book 10-5 convex hulls
multidimensional 12-28
two-dimensional 12-21
convolution 12-6
correlation coefficients 13-11
cos 17-7
covariance 13-11
creating
cell array 20-20
multidimensional array 19-4
sparse matrix 16-8
string array 18-6
strings 18-5
structure array 20-5
cropping graphics
in M-books 10-22
cross 19-14
cubic interpolation
multidimensional 12-18
one-dimensional 12-12
cubic spline interpolation 12-12 curly braces
for cell array indexing 20-20, 20-22 to build cell arrays 20-22
to nest cell arrays 20-29
current directory
at startup for MATLAB 1-3
changing 5-22
contents of 5-22
field in toolbar 5-20
relevance to MATLAB 5-20
tool 5-20
Current Directory browser 5-20 preferences 5-30
curve fitting 13-22
Basic Fitting interface 13-31 error bounds 13-30
exponential 13-27
polynomial 12-7, 13-22
curves
computing length 14-19
Cuthill-McKee, reverse ordering 16-29

D

DAE
solution of 15-3
data

type for input 6-54
data analysis
column-oriented 13-8
data class hierarchy 22-4
data consistency
evaluating M-books 10-7
in an M-book 10-7
using calc zones in M-books 10-7
data filtering. *See* filtering
data fitting. *See* curve fitting
data gridding
multidimensional 12-18
data organization
cell arrays 20-27
multidimensional arrays 19-16 structure arrays 20-12
data sets
See HDF data sets
data types 17-22
cell 17-24
char 17-24
classes 22-4
double 17-24
precision 6-55
double 6-55
reading files 6-54
single 17-23
sparse 17-24
sparse matrices 17-22
struct 17-24
uint 17-24
user-defined 22-4
UserObject 17-24
data. *See also*
multivariate data
statistical data
univariate data
datatip 7-27
date 17-67
datenum 17-64, 17-65
dates
base 17-63
conversions 17-64
formats 17-62
handling and converting 17-62 number 17-63
string, vector of input 17-65
datestr 17-64, 17-65

datevec 17-64
dbclear 7-30
dbstack 7-24
dbstop 7-23
deblank 18-7
debugger
option for UNIX 1-7
debugging
ending 7-30
example 7-18
M-files 7-2, 7-17
option for using functions 7-36 prompt 7-24
techniques 7-17
tool 7-18
debugging class methods 22-6
decimal places in output 3-10
decimal representation
to binary 18-15
to hexadecimal 18-15
decomposition
eigenvalue 11-35
Schur 11-37
singular value 11-39
deconvolution 12-6
default function 21-15
defaults
preferences for MATLAB 2-21
setting in startup file for MATLAB 1-5
Define Autoinit Cell command 10-26
Define Calc Zone command 10-27 Define Input Cell command 10-27
Delaunay multidimensional tessellations 12-30
Delaunay triangulation 12-21
closest point searches 12-25
delay-differential equations
with ODE solvers 15-55
delete 5-25
deleting
cells from cell array 20-24
fields from structure arrays 20-10
files 5-25
matrix rows and columns 17-49
deletion operator 17-49
delimiter in string 18-13
delimiters,defined 6-9
demos
accessing from Launch Pad 2-5

density of sparse matrix 16-7
derivatives
polynomial 12-6
desktop
configuration 2-7
saving 2-7
description 2-2
font preferences for 2-23
layout, predefined options 2-16
starting without 1-8
tools
closing 2-9
opening 2-4
windows
closing 2-9
docking 2-10
grouping together 2-14
moving 2-10
opening 2-4
sizing 2-9
undocking 2-12
determinant of matrix 11-21
development environment for MATLAB 2-2 diag 16-26
diagonal
creating sparse matrix from 16-10
diary 3-12, 6-18
difference equations 13-40
differential equations 15-1
boundary value problems for ODEs 15-57 initial value problems for ODEs and DAEs 15-3
partial differential equations 15-82
differential-algebraic equations. *See* DAE
dim argument for cat 19-7
dimensions
deleting 20-24
permuting 19-12
removing singleton 19-11
dir 5-22
direct methods for systems of equations 16-36
directories
adding to path 22-7
caching for MATLAB 5-14
class 22-7
Contents.m file 17-5
creating 5-24
deleting 5-25
help for 17-5

MATLAB
caching 17-6
private 17-45, 22-6
renaming 5-25
searching contents of 5-20
temporary 6-53
See also current directory, search path discrete Fourier transform. *See* Fourier transforms
disp 20-17
dispatch type 22-68
dispatching priority 17-12
display method 22-12
examples 22-13
display pane in Help browser 4-17
displaying
cell arrays 20-21
error and warning messages 17-60 field names for structure array 20-6 sparse matrices 16-15
displaying output 3-9
distance between nodes 16-22
docking windows in desktop 2-10
documentation
location preference for Help browser 4-22 printing 4-26
problems, reporting 4-32
viewing 4-17
dot product 11-8
double data type 17-24
double precision 6-55
double-precision matrix 17-22
dragging in the desktop 2-20

E

Earth Observing System (EOS)

See EOS (Earth Observing System) echo
preferences setting 3-15
edit 7-4
editing
in Command Window 3-6
M-files 7-3
\$nopage>editor
See also Editor/Debugger 3-6
editor
accessing 17-5
for creating M-files 17-3, 17-5
setting as default 7-36
editor external to MATLAB 7-36
Editor, stand-alone (Windows) 7-6
Editor/Debugger

description 7-3
example 7-18
indenting 7-3
preferences 7-35
See also editor
eig 19-15
eigenvalues 11-35
of sparse matrix 16-39
eigenvectors 11-35
element-by-element organization for structures 20-15
ellipsis (...) in functions 3-5
else,elseif 17-35, 17-36
Embed Figures in M-book check box 10-21 embedding graphics
in M-book 10-21
empty arrays
andif statement 17-37
and relational operators 17-29
and while loops 17-39
empty matrices 17-57
end method 22-18
end of file 6-56
ending MATLAB 1-15
environment settings at startup 1-5
EOS (Earth Observing System)
sources of information 6-29
eps 17-21
epsilon 17-21
equal to operator 17-28
error 17-8
error bounds
curve fitting 13-30
error messages
in Command Window 3-12
error style
definition of 10-19
error tolerance
BVP problems 15-77
effects of too large (ODE) 15-55
machine precision 15-53
ODE problems 15-18
errors 17-59
color indicators 2-24
displaying 17-60
finding 7-17
handling 17-59
run-time 7-17

syntax 7-17
eval
for error handling 17-59
Evaluate Calc Zone command 10-28
Evaluate Cell command 10-28
Evaluate Loop command 10-29
Evaluate Loop dialog box 10-15
Evaluate M-Book command 10-30
evaluating
string containing function name 17-54
string containing MATLAB code 18-16
string containing MATLAB expression 17-54 evaluating a selection
Command History 3-18
Command Window 3-3
evaluating M-books
ensuring data consistency 10-7
event location (ODE) 15-29
advanced example 15-44
simple example 15-41
examples
adjacency matrix (sparse) 16-17
airflow modeling 16-23
Bucky ball 16-18
checking number of function arguments 17-14
container class 22-54
for 17-40
function 17-8
if 17-36
in documentation, index of 4-10
inheritance 22-38
M-file for structure array 20-12
polynomial class 22-24
running from Help browser 4-19
script 17-7
second difference operator 16-9
sparse matrix 16-9, 16-17
switch 17-38
theoretical graph (sparse) 16-17
vectorization 17-71
while 17-39
examples (BVP)
boundary condition at infinity (fsbvp) 15-72
Mathieu's Equation (mat4bvp) 15-63
rapid solution changes (shockbvp) 15-68
two solutions (twobvp) 15-58
examples (DAE)

electrical circuit (`amp1dae`) 15-5
Robertson problem (`hb1dae`) 15-47
examples (MATLAB PDE) 15-82
electrodynamics problem (`pdex4`) 15-94
simple PDE (`pdex1`) 15-88
examples (ODE) 15-31
advanced event location (`orbitode`) 15-44 Brusselator system (`brussode`) 15-38
constant mass matrix (`fem2ode`) 15-5
finite element discretization (`fem1ode`) 15-35 nonstiff 15-11
rigid body (`rigidode`) 15-32
Robertson problem (`hb1ode`) 15-5
simple event location (`ballode`) 15-41
stiff 15-14
strongly state-dependent mass matrix (`burgersode`) 15-5
time-, state-dependent mass matrix (`batonode`) 15-5
van der Pol (`vdpode`) 15-33
exclusive OR operator 17-31
execution,pausing 17-69
exiting MATLAB 1-15
expanding
cell arrays 20-21
structure arrays 20-5
exponential curve fitting 13-27
exporting
ASCII data 6-16, 6-18
binary data formats 6-25
in HDF format 6-41
exporting data,overview 6-2
expressions
arithmetic 17-26
involving empty arrays 17-29
logical 17-29
most recent answer 17-21
overloading 22-21
relational 17-28
scalar expansion with 17-27
ext startup option 1-7
external program, running from MATLAB 17-70 eye 16-25
derivation of the name 11-11

F

factorization 16-30
Cholesky 11-25
for sparse matrices 16-30
Cholesky 16-33
LU 16-30
triangular 16-30

Hermitian positive definite 11-26
incomplete 16-35
LU 11-26
partial pivoting 11-27
positive definite 11-25
QR 11-28
fast Fourier transform. *See* Fourier transforms favorites in Help browser 4-15
fclose 6-52, 6-62
feedback to The MathWorks 4-32
fem1ode demo 15-35
fem2ode demo 15-5
feof 6-56
feval
using on function handles 21-3, 21-9
using on function name strings 21-28 fid
See file identifiers
fieldnames 20-6
fields 20-4, 20-5
accessing data within 20-7
accessing withgetfield 20-9
adding to structure array 20-10
applying functions to 20-10
all like-named fields 20-11
assigning data to 20-5
assigning withsetfield 20-9
deleting from structures 20-10
indexing within 20-8
names 20-6
size 20-10
writing M-files for 20-11
fields 20-6
file identifiers
clearing 6-62
file identifiers (fid) 6-52
file management system. *See* source control system
filebrowser 5-20
files
ASCII
reading formatted text 6-59
beginning of 6-56
binary
controlling data type values read 6-54
closing 6-52, 6-62
contents, viewing 5-27
copying 5-25
creating in the Current Directory browser 5-24

current position 6-56
deleting 5-25
editing M-files 7-3
end of 6-56
failing to open 6-52
identifiers 6-52
log 1-6
MATLAB related, listing 5-22
naming 5-14
opening 5-26, 6-52
operations in MATLAB 5-20
permissions 6-52
position 6-56
renaming 5-25
running 5-27
source control 9-2
temporary 6-53
viewing contents of 5-27
files, ASCII
reading 6-58
specifying delimiter 6-12
writing 6-61
files, binary
data types 6-54
reading 6-54
writing to 6-55
fill-in of sparse matrix 16-22
filtering
difference equations 13-40
Find & Replace feature in Current Directory browser 5-28
find function
and sparse matrices 16-16
and subscripting 17-32
finding
M-file content 7-12
string in M-files 5-28
substring within a string 18-13
text in page of Help browser 4-19 finish.m 1-15
finishdlg.m 1-15
finishsav.m 1-15
finite differences 13-12
finite element discretization (ODE example) 15-35
first-order differential equations
representation for BVP solver 15-64
representation for ODE solvers 15-11 flags for startup 1-5
float 6-55

floating-point number
largest 17-21
smallest 17-21
floating-point precision 6-55
floating-point relative accuracy 17-21 flow control 17-35
catch 17-42
continue 17-41
else 17-35
elseif 17-35
for 17-40
if 17-35
return 17-42
switch 17-37
try 17-42
while 17-39
folders. *Seedirectories*
font
Help browser 4-24
Help browser display pane 4-25
Help Navigator 4-24
in Command Window 3-15
preferences in MATLAB 2-23
fopen 6-52
failing 6-52
for 17-40, 20-30
example 17-40
indexing 17-40
nested 17-40
syntax 17-40
format
controlling numeric format in M-book 10-20
date 17-62
in Array Editor 5-12
preferences 3-14
format 3-10
in M-book 10-20
Fourier analysis 13-43
concepts 13-44
Fourier transforms 13-43
calculating sunspot periodicity 13-45
FFT-based interpolation 12-13
length vs. speed 13-50
phase and magnitude of transformed data 13-49
fragmentation, reducing 17-74
fread 6-54
rewind 6-56

fsbvp demo 15-72
fseek 6-56
ftell 6-56
full 16-26, 16-29
full text searching in Help browser 4-13 func2str
description 21-21
example 21-21
function call history report 8-11
function definition line 17-4, 17-9
for subfunction 17-43
function details report 8-10
function functions 14-1
function handles 17-24, 21-1
benefits of 21-3
constructing 21-7
converting from function name string 21-22
converting to function name string 21-21
effect on performance 21-5
error conditions 21-26
evaluating a nonscalar function handle 21-27
including path in the constructor 21-26 nonexistent function 21-26
evaluating 21-9
finding the binding functions 21-13
for subfunctions, private functions 21-4
maximum name length 21-7
operations on 21-21
overloading 21-3, 21-9, 21-18
overview of 21-2
passing 21-3
saving and loading 21-25
testing for data type 21-23
testing for equality 21-23
types
constructor 21-18
overloaded 21-18
private 21-20
simple 21-17
subfunction 21-19
function name string
converting from function handle 21-21 converting to function handle 21-22
function workspace 5-8, 17-14
functions 17-8
applying
to multidimensional structure arrays 19-20
to structure contents 20-10
applying to cell arrays 20-27

arguments
passing variable number of 17-16
body 17-4, 17-10
calling 17-13
calling context 17-14
characteristics 17-3
class 22-11
clearing from memory 17-13
color indicators 2-24
computational, applying to structure fields 20-10
contents 17-9
converters 22-25
creating arrays with 19-6
default 21-15
dispatching priority 17-12
example 17-8
executing function name string 17-54 help for 4-28
reference page 4-3
searching 4-13
inferior 22-66
isa 22-11
logical 17-31
long (on multiple lines) 3-5
mathematical. See mathematical functions maximum number of characters in 3-5 multiple in one line 3-4
multiple output values 17-9
naming 5-14, 17-11
resolution 17-12
nested 17-75
optimization 17-71
optimizing 14-9
order of arguments
varargin, varargout 17-16, 17-18 overloaded 4-29
overloaded methods 21-16
overloading 22-21, 22-23, 22-31
primary 17-43
private 17-45
running 3-2
sequence report 8-11
sharing same name 4-29
storing as pseudocode 17-13
subassign 22-17
subfunction 17-43
subsref 22-14
superior 22-66
time used report 8-10

which 22-70
functions command
description 21-13
for constructor function handles 21-18 for overloaded function handles 21-18 for private function handles 21-20 for simple function handles 21-17 for subfunction handles 21-19
return fields 21-14
fwrite 6-55

G

Gaussian elimination 11-26

geodesic dome 16-18

geometric analysis

multidimensional 12-27

two-dimensional 12-19

get method 22-13

getfield 20-9

global attributes

HDF files 6-35

global minimum 14-15

global variables 17-19

rules for use 17-20

gplot 16-18

graph

characteristics 16-21

defined 16-17

theoretical 16-17

graphics

controlling output in M-book 10-21 embedding in M-book 10-21

in M-books 10-20

graphing

variables from the Workspace browser 5-9

greater than operator 17-28

greater than or equal to operator 17-28

Group Cells command 10-30

H

H1 line 17-4, 17-10

and help command 17-4

andlookfor command 17-4

hb1dae demo 15-47

hb1ode demo 15-5

HDF (Hierarchical Data Format) calling conventions 6-31

exporting data 6-41

importing into MATLAB 6-33 MATLAB support 6-30

MATLAB utility API 6-49 NCSA documentation 6-29 output arguments 6-32

overview 6-29

programming model 6-33
symbolic constants 6-32
HDF data sets
accessing 6-36
associating attributes with 6-47 attributes 6-38
closing access 6-46
creating 6-43
getting information about 6-37 reading 6-39
using predefined attributes 6-48 writing attributes 6-48
HDF files
access modes 6-34
associating attributes with 6-47 closing 6-46
closing all open identifiers 6-50 creating 6-42
getting information about 6-34 listing open identifiers 6-49 opening 6-34
reading global attributes 6-35 writing attributes 6-48
writing data 6-44
HDF-EOS
Earth Observing System 6-29
headers, printing with M-files 7-43
help
from Editor/Debugger 7-5
functions 4-28
in Command Window 4-29
MATLAB 4-2
M-file 4-3
subset for specified products 4-7
help 4-29, 17-10
and H1 line 17-4
Help browser
contents listing 4-9
copying information from 4-19
description 4-4
display pane 4-17
favorites (bookmarks) 4-15
font preferences 4-24
index 4-11
navigating 4-18
preferences 4-21
printing help 4-26
running examples from 4-19
searching 4-13
Web pages, viewing 4-20
Help Navigator 4-5
help text 17-4
helpbrowser 4-4
Hermitian positive definite matrix 11-26

hexadecimal, converting from decimal 18-15

Hide Cell Markers command 10-31

Hierarchical Data Format (HDF).

See HDF

hierarchy of data classes 22-4

history of functions called 8-11

history, automatic log file 1-6

home 3-9

HTML

source, viewing in Help browser 4-20 viewer 4-4

I

identity matrix 11-11

if 17-35

and empty arrays 17-37

example 17-36

nested 17-36

imaginary unit 17-21

Import Data option 6-20

import functions

comparison of features 6-11

Import Wizard

importing binary data 6-20 with ASCII data 6-4

importing

ASCII data 6-4

binary data 6-20

HDF data 6-33

MAT-file data 6-23

sparse matrix 16-12

spreadsheet data 6-23

importing data

Import Wizard 6-4

overview 6-2

incomplete factorization 16-35 indenting 7-9, 7-40

indenting for syntax 3-5

index

examples in documentation 4-10

Help browser 4-11

results 4-12

tips 4-12

indexed reference 22-14
indexing 17-46
advanced 17-51
cell array 20-20
content 20-21
for loops 17-40
multidimensional arrays 19-9
nested cell arrays 20-30
nested structure arrays 20-18
structures within cell arrays 20-32
within structure fields 20-8
indices, how MATLAB calculates 17-53 Inf 17-21
infeasible optimization problems 14-15 inferior function 22-66
infinity (represented in MATLAB) 17-21 inheritance
example class 22-38
multiple 22-36
simple 22-35
initial conditions
ODE 15-6
ODE example 15-12
PDE 15-84
PDE example 15-89
initial guess (BVP)
example 15-65
quality of 15-68
initial value problems 15-3
defined 15-6
initial-boundary value problems 15-82 initiation (init) file for MATLAB 1-5 inline objects
representing mathematical functions 14-4 inner product 11-7
input
from keyboard 17-69
obtaining from M-file 17-69
input cells
controlling evaluation of 10-15
controlling graphic output 10-21
converting autoinit cell to 10-27
converting text to 10-27
converting to autoinit cell 10-26
converting to cell groups 10-32
converting to text 10-11
defining in M-books 10-8
evaluating 10-12
evaluating cell groups 10-13
evaluating in a loop 10-15
maintaining consistency 10-6

timing out during evaluation 10-29
use of Word Normal style 10-11 input for MATLAB 3-2
Input style
definition of 10-19
integer data type 6-60
integers, changing to strings 18-15 integration
double 14-20
numerical 14-19
See also differential equations integration interval (ODE solvers) 15-9 interactive user input 17-69
interpolation 12-10
comparing methods graphically 12-14
FFT-based 12-13
multidimensional 12-17
scattered data 12-35
one-dimensional 12-11
speed, memory, smoothness 12-12
three-dimensional 12-17
two-dimensional 12-13
interrupting a running program 3-11
inverse of matrix 11-21
inverse permutation of array dimensions 19-13 ipermute 19-2, 19-13
isa 22-11
using with function handles 21-23
isempty 17-29
isequal
using with function handles 21-23
isinf 17-32
isnan 17-32
iterative methods
for sparse matrices 16-38
for systems of equations 16-36

J

Jacobian matrix (ODE) 15-23
generating sparse numerically 15-24 specifying 15-24
vectorizing ODE function 15-25
Java and MATLAB OOP 22-8
Java VM
starting without 1-9

K

K>> prompt in Command Window 3-3 key bindings 7-40
keyboard 7-18
keyboard shortcuts and accelerators 2-18 keys for editing in Command Window 3-8 keywords
in documentation 4-11

keywords, color indicators 2-24
Kronecker tensor matrix product 11-11

L

lasterr 17-59
and error handling 17-59
Launch Pad 2-5
least squares 16-34
length of curve, computing 14-19
less than operator 17-28
less than or equal to operator 17-28
license information 4-32
line numbers 7-11
line numbers, printing with M-files 7-43
line wrapping 3-15
linear algebra 11-5
linear equations
minimal norm solution 11-23
overdetermined systems 11-15
rectangular systems 11-22
solving systems of 11-13
underdetermined systems 11-18
linear interpolation
multidimensional 12-18
one-dimensional 12-11
linear systems of equations
direct methods 16-36
iterative methods 16-36
sparse 16-36
linear transformation 11-5
lines, maximum number of characters in 3-5
load 5-7, 16-12, 17-74
function handles 21-25
loading data, overview 6-2
loading objects 22-60
loadobj example 22-62
Lobatto IIIa BVP solver 15-60
local variables 17-19
locking files 9-9
log
automatic, for MATLAB 1-6
file 1-6
functions 3-17
session 3-12
logfile startup option 1-6
logical expressions 17-29

and subscripting 17-32
logical functions 17-31
logical operators 17-29
rules for evaluation 17-30
long 6-55
long integer 6-55
lookfor 5-30, 17-4, 17-10
and H1 line 17-4
looping
to evaluate input cells 10-15
loops
for 17-40
while 17-39
Lotus 123
importing 6-23
lowercase usage in MATLAB 3-4
lu 16-30, 16-32
LU factorization 11-26
for sparse matrices and reordering 16-31

M

mass matrix (ODE) 15-27 initial slope 15-29
singular 15-29
sparsity pattern 15-28
specifying 15-28
state dependence 15-28 mat4bvp demo 15-58
mat4bvp demo 15-63
MAT-files
creating 5-5
defined 5-5
importing data from 6-23
loading 5-7
mathematical functions
as function input arguments 14-2
finding zeros 14-12
minimizing 14-9
numerical integration 14-19
plotting 14-6
representing in MATLAB 14-4
mathematical operations on sparse matrices 16-25
Mathieu's equation (BVP example) 15-63 MATLAB
data type classes 22-4
files, listing 5-22
help for 4-2
path 5-14
programming
functions 17-8

M-files 17-3
quick start 17-3
scripts 17-7
prompt 3-2
quitting 1-15
structures 22-8
version 17-21
window 2-2

MATLAB commands
executing in a Word document 10-12 matlab.mat 5-6
matlabrc.m 1-5
matlabroot 1-3
matrices 11-5
accessing multiple elements 17-47
accessing single elements 17-46
advanced indexing 17-51
as linear transformation 11-5 characteristic polynomial 12-5 characteristic roots 12-5
concatenating 17-50
creation 11-5
deleting rows and columns 17-49 determinant 11-21
editing 5-10
empty 17-57
expanding 17-48
full to sparse conversion 16-3, 16-7 identity 11-11
inverse 11-21
orthogonal 11-28
pseudoinverse 11-22
rank deficiency 11-17
sparse 17-22
symmetric 11-8
triangular 11-25
matrix
as index forfor loops 17-40
double-precision 17-22
iterative methods 16-38
power operator 17-27
single-precision 17-22
See also matrices
matrix operations
addition and subtraction 11-7
division 11-13
exponentials 11-33
multiplication 11-9
powers 11-32
transpose 11-8
matrix products

Kronecker tensor 11-11
max 16-26
M-books
creating 10-3
cropping graphics 10-22
ensuring data consistency 10-7
entering text and commands 10-6
evaluating all input cells in 10-15
modifying style template 10-18
opening 10-5
printing 10-18
protecting data integrity 10-6
sizing graphic output 10-22
mean 19-14
measuring performance of M-files 8-2 meditor 7-6
memory
allocating for variables 17-76
function workspace 17-14
management 17-74
Out of Memory message 17-78
reducing fragmentation 17-74
mesh size (BVP) 15-80
methods 22-3
converters 22-20
determining which is called 22-70
display 22-12
end 22-18
get 22-13
invoking on objects 22-5
listing 22-33
overloaded 21-16
precedence 22-67
required by MATLAB 22-9
set 22-13
subsasgn 22-14
subsref 22-14
M-file help 4-3
viewing in Current Directory browser 5-27 M-files
changes recognized 5-14
comments 17-11
content, viewing 5-27
contents 17-4
corresponding to functions 22-22
creating
from Command History 3-19 in MATLAB directory 17-6 in MATLAB directory tree 17-6
overview 7-2

quick start 17-3
warning 5-14
creating with text editor 17-5
debugging 7-2, 7-17
dispatching priority 17-12
editing 7-2, 7-3
file association (Windows) 7-6
input
keyboard 17-69
obtaining interactively 17-69
kinds 17-3
location of 5-14
naming 5-14, 17-2
opening 7-5
operating on structures 20-11
optimization 17-71
overview 17-4
pausing 7-18
pausing during execution 17-69
performance of 8-2
primary function 17-43
printing 7-16
printing, preferences 7-42
profiling 8-2
replacing content 7-12
replacing string in 5-28
representing mathematical functions 14-4 running
Command Window 3-11
from Current Directory browser 5-27 running at startup 1-6
saving 7-15
search path 5-14
searching contents of 5-28
source control 9-2
subfunction 17-43
superseding existing names 17-44
viewing description 5-31
Microsoft Excel
importing 6-23
Microsoft Windows
MATLAB use of system resources 17-77 Microsoft Word
converting document to M-book 10-5 specifying version and location 10-24 minimize startup
option 1-6
minimizing mathematical functions
of one variable 14-9
of several variables 14-10
options 14-11

minimum degree ordering 16-29
mislocked 17-20
mkdir 5-24
mlock 17-20
model files
source control 9-2
models
interfacing with source control systems 9-2 Moore-Penrose pseudoinverse 11-22
more 3-10
mouse, right-clicking 2-17
movies
saving in AVI format 6-27
multidimensional arrays 19-2
applying functions 19-14
element-by-element functions 19-14 matrix functions 19-15
vector functions 19-14
cell arrays 19-18
computations on 19-14
creating 19-4
at the command line 19-4
with functions 19-6
with thecat function 19-7
extending 19-5
format 19-8
indexing 19-9
avoiding ambiguity 19-9
with the colon operator 19-9
number of dimensions 19-8
organizing data 19-16
permuting dimensions 19-12
removing singleton dimensions 19-11
reshaping 19-10
size of 19-8
storage 19-8
structure arrays 19-19
applying functions 19-20
subscripts 19-3
multidimensional data gridding 12-18 multidimensional interpolation 12-17
scattered data 12-27
multiple conditions forswitch 17-38 multiple inheritance 22-36
multiple item selection 2-19
multiple lines for functions 3-5
multiple output values 17-9
multiprocessing 3-3
multistep solver (ODE) 15-7
multivariate data

matrix representation 13-4
vehicle traffic sample data 13-4 munlock 17-20

N

names
function 17-11
structure fields 20-6
superseding 17-44
variable 17-19
naming functions and variables 5-14 NaN 17-21
NaNs
propagation 13-14
removing from data 13-15
nargin 17-14
nargout 17-14
ndgrid 19-2
ndims 19-2, 19-8
nearest neighbor interpolation
multidimensional 12-18
one-dimensional 12-11
three-dimensional 12-17
two-dimensional 12-13
nesting
cell arrays 20-29
for loops 17-40
functions 17-75
if statements 17-36
structures 20-17
newlines in string arrays 18-12 nnz 16-13, 16-15
nodes 16-17
distance between 16-22
numbering 16-19
nodesktop startup option 1-8 nojvm startup option 1-9
nonstiff ODE examples
rigid body (rigidode) 15-32
nonzero elements
number of 16-13
nonzero elements of sparse matrix 16-13
maximum number in sparse matrix 16-9
storage 16-6, 16-13
values 16-13
visualizing with spy plot 16-21
nonzeros 16-13
Normal style (Microsoft Word)
default style in M-book 10-18
defaults 10-19
used in undefined input cells 10-11 normal.dot 10-25

normalizing data 13-23
norms
vector and matrix 11-12
nosplash startup option 1-9
not equal to operator 17-28
NOT operator
rules for evaluating 17-30
Not-a-Number 17-21
Notebook
configuring 10-24
overview 10-2
notebook command
set up 10-24
using 10-3
Notebook menu
in Word menu bar 10-3
Notebook Options command 10-31
now 17-68
number of arguments 17-14
numbering lines 7-11
numbers
changing to strings 18-15 date 17-63
time 17-63
numeric format
controlling in M-book 10-20
output 3-10
preferences 3-14
numerical integration 14-19
computing length of curve 14-19
double 14-20
nzmax 16-13, 16-15

O

objective function 14-2
return values 14-15
object-oriented programming 22-2 converter functions 22-25 features of 22-3
inheritance
multiple 22-36
simple 22-35
overloading 22-21, 22-23 subscripting 22-14
overview 22-3
See also classes and objects objects
accessing data in 22-13
as indices into objects 22-19 creating 22-4
invoking methods on 22-5 loading 22-60
overview 22-3
precedence 22-65

saving 22-60
ODE
coding in MATLAB 15-11 defined 15-6
overspecified systems 15-51
rewriting problem as first-order system 15-11 solution of 15-3
ODE solver
evaluate solution at specific points 15-16 ODE solver properties 15-16
error tolerance 15-18
event location 15-29
fixed step sizes 15-52
Jacobian matrix 15-23
mass matrix 15-27
modifying property structure 15-17
ode15s 15-30
odeset function 15-17
querying property structure 15-18
solver output 15-20
step size 15-25
ODE solvers 15-6
algorithms
Adams-Basforth-Moulton PECE 15-7 Bogacki-Shampine 15-7
Dormand-Prince 15-7
modified Rosenbrock formula 15-8 numerical differentiation formulas 15-8 backward
differentiation formulas 15-30 backwards in time 15-55
basic example
nonstiff problem 15-11
stiff problem 15-14
basic syntax 15-8
calling 15-12
examples 15-31
minimizing output storage 15-51
minimizing startup cost 15-52
multistep solver 15-7
nonstiff problems 15-7
numerical differentiation formulas 15-30 obtaining solutions at specific times 15-9 one-step
solver 15-7
overview 15-6
passing additional parameters 15-10 performance 15-16
problem size 15-51
representing problems 15-10
sampled data 15-54
stiff problems 15-7, 15-14
troubleshooting 15-50
variable order solver 15-30
odephas2 output function 15-21
odephas3 output function 15-21

odeplot output function 15-21
odeprint output function 15-21
offsets for indexing 17-53
one-dimensional interpolation 12-11 ones 16-25, 17-48, 19-6
one-step solver (ODE) 15-7
online help 17-10
open 5-26, 7-5
opening
files
Current Directory browser 5-26 opening files 6-52
failing 6-52
HDF 6-34
permissions 6-52
openvar 5-11
operating system commands 3-12
operator
second difference 16-9
operator precedence 17-32
overriding 17-33
operators 17-26
& 17-29
| 17-29
~ 17-29
applying to cell arrays 20-27
applying to structure fields 20-10
arithmetic 17-26
colon 17-26, 19-5, 19-9, 19-15, 20-24
complex conjugate 17-26
deletion 17-49
equal to 17-28
greater than 17-28
greater than or equal to 17-28
less than 17-28
less than or equal to 17-28
logical 17-29
matrix power 17-27
not equal to 17-28
overloading 22-3, 22-21
power 17-26
relational 17-28
subtraction 17-26
table of 22-22
unary minus 17-26
optimization 14-9, 17-71
calling sequence changes (Version 5) 14-17 helpful hints 14-15
options parameters 14-11

preallocation, array 17-73
troubleshooting 14-15
vectorization 17-71
See also minimizing mathematical functions optimization code
updating to MATLAB Version 5 syntax 14-16 optimizing performance of M-files 8-2
options
shutdown 1-15
orbitode demo 15-44
Ordinary Differential Equations. *See* ODE organizing data
cell arrays 20-27
multidimensional arrays 19-16
structure arrays 20-12
orthogonal matrix 11-28
Out of Memory message 17-78
outer product 11-7
outliers
removing from statistical data 13-15 output
Command Window 3-10
display format 3-10
MATLAB 3-2
not displaying 3-9
spaces per tab 3-14
spacing of 3-14
output cells
converting to text 10-16
purging 10-17
output points (ODE)
increasing number of 15-22
output properties (ODE) 15-20
increasing number of output points 15-22 output storage
minimizing for ODE problems 15-51 Output style
definition of 10-19
overdetermined
rectangular matrices 11-15
overloaded functions 4-29
overloading 22-14
arithmetic operators 22-29
functions 22-21, 22-23, 22-31
loadobj 22-61
operators 22-3
pie3 22-57
saveobj 22-61
overspecified ODE systems 15-51

P

pack 17-74
page subscripts 19-3

paging in the Command Window 3-10
parentheses

for input arguments 17-9
overriding operator precedence with 17-33 Partial Differential Equations. *See PDE*
partial fraction expansion 12-8
passing arguments
by reference 17-14
by value 17-14
Paste Special option 6-20
path
adding directories to 5-23, 22-7
changing 5-17
description 5-14
order of directories 5-14
saving changes 5-18
viewing 5-17
pathdef.m 5-14, 5-19
pathtool 5-16
pausing during M-file execution 17-69
pausing execution of M-file 7-22
PCODE 17-13
PCs and MATLAB use of system resources 17-77
PDE 15-82
defined 15-83
discretized 15-54
PDE solver (MATLAB) 15-84
additional known parameters 15-87
basic syntax 15-85
evaluate solution at specific points 15-93 examples
electrodynamics problem 15-94
simple PDE 15-88
performance 15-93
representing problems 15-88
PDE solver (MATLAB) properties 15-94 pdex1 demo 15-88
pdex2 demo 15-83
pdex3 demo 15-83
pdex4 demo 15-94
pdex5 demo 15-83
PDF
printing documentation files 4-26
reader, preference for Help browser 4-23 percent sign (comments) 17-11
performance
de-emphasizing an ODE solution component 15-53
improving for BVP solver 15-75
improving for M-files 8-2
improving for ODE solvers 15-16

improving for PDE solver 15-93
improving for startup 1-10
permission string 6-52
permutations 16-26
permute 19-2, 19-12
permuting array dimensions 19-12
inverse 19-13
persistent variables 17-20
pi 17-21
pie3 function overloaded 22-57
plane organization for structures 20-14 plotting
mathematical functions 14-6
plotting results 8-12
polar 17-7
polynomial
curve fitting 13-22
regression 13-18
polynomial interpolation 12-11
polynomials
basic operations 12-3
calculating coefficients from roots 12-4 calculating roots 12-4
curve fitting 12-7
derivatives 12-6
evaluating 12-5
example class 22-24
multiplying and dividing 12-6
partial fraction expansion 12-8 representing as vectors 12-4
pop-up menus 2-17
power operator 17-26
preallocation 17-73
cell array 17-73, 20-23
structure array 17-74
precedence
object 22-65
operator 17-32
overriding 17-33
precision
char 6-55
data types 6-55
double 6-55
float 6-55
long 6-55
output display 3-10
short 6-55
single 6-55
uchar 6-55

preconditioner for sparse matrix 16-35
preferences
Command Window 3-13
general, MATLAB 2-23
MATLAB 2-21
setting 2-21
toolbox caching 1-10
viewing 2-21
primary function 17-43
printing
Command Window contents 3-11 documentation 4-26
help 4-26
M-files 7-16
printing an M-book
cell markers 10-18
color 10-18
color modes 10-23
defaults 10-18
private directories 17-45
private directory
in dispatching priority 17-12
private functions 17-45
function handles to 21-4, 21-20
precedence of 22-69
private methods 22-6
problems, reporting to The MathWorks 4-32
product filter in Help browser 4-7
preference 4-23
profile 8-4
example 8-6
profiling 8-2
reports 8-7
programming, object-oriented 22-2
programs
running external 17-70
running from MATLAB 3-12
stopping while running 3-11
prompt
debugging type 7-24
prompt, in Command Window 3-2
property structure (BVP)
creating 15-76
modifying 15-77
querying 15-77
property structure (ODE)
creating 15-17

modifying 15-17
querying 15-18
pseudocode 17-13
pseudoinverse
of matrix 11-22
Purge Output Cells command 10-31
purging output cells 10-17

Q

QR factorization 11-28, 16-33
quad,quadl

differ from ODE solvers 15-51 quadrature. *See* numerical integration quit 17-74
quitting MATLAB 1-15

R

rand 16-25
randn 19-6
rank 16-4
deficiency 16-34
rank deficiency
detecting 11-30
rectangular matrices 11-17 reading
HDF data 6-33
reading data from a disk file overview 6-2
realmax 17-21
realmin 17-21
recall, smart 3-7
rectangular matrices
identity 11-11
overdetermined systems 11-15 pseudoinverse 11-22
QR factorization 11-28
rank deficient 11-17
singular value decomposition 11-39 underdetermined systems 11-18 redo in MATLAB 2-17
reducing memory fragmentation 17-74 reference pages 4-3
reference, subscripted 22-14
regression
linear-in-the-parameters 13-19 multiple 13-21
polynomial 13-18
regserver startup option 1-6
relational operators
empty arrays 17-29
strings 18-11
relative accuracy
BVP 15-78
norm of ODE solution 15-19
ODE 15-19

removing
cells from cell array 20-24
fields from structure arrays 20-10 singleton dimensions 19-11
reorderings 16-26
and LU factorization 16-31
for sparser factorizations 16-28 minimum degree ordering 16-29 to reduce bandwidth 16-29
replacing M-file content 5-28
replacing substring within string 18-13 repmap 19-6
reports
function call history 8-11
function details 8-10
saving 8-13
summary 8-7
representing
mathematical functions 14-4
requirements 1-3
reshape 19-10, 20-25
reshaping
cell arrays 20-25
multidimensional arrays 19-10 residuals
analyzing 13-24
for exponential data fit 13-29
resizing windows in the desktop 2-9 results in MATLAB, displaying 5-11 return 17-42
revision control system. *See* source control system
rigid body (ODE example) 15-32
rigidode demo 15-32
rmfield 20-10
roadmap for documentation 4-10
Robertson problem
DAE example 15-47
ODE example 15-5
root directory for MATLAB 1-3
roots
polynomial 12-4
running
M-files 5-27
run-time errors 7-17

S

sampled data with ODE solvers 15-54

save 5-6, 16-12, 17-74
function handles 21-25
saveobj example 22-62
saving
M-files 7-15
objects 22-60

scalar
and relational operators 18-11 as a matrix 11-6
expansion 17-27
string 18-11
scalar product 11-8
scattered data
multidimensional interpolation 12-35 multidimensional tessellation 12-27 triangulation and
interpolation 12-19
Schur decomposition 11-37
Scientific Data API
programming model 6-33
scripts 17-3, 17-7
characteristics 17-3
example 17-7
executing 17-7
scrolling in Command Window 3-10
SCS. *See* source control system
seamount data set 12-20
search path 5-14, 17-12
M-files on 17-43
searching
Help browser 4-13
results 4-14
text in page 4-19
tips 4-14
type 4-13
M-file content 5-28, 7-12
technical support Online Knowledge Base 4-13
second difference operator, example 16-9
section breaks
in calc zones 10-27
selecting multiple items 2-19
semicolon (;)
after functions 3-9
between functions 3-4
separator in functions 3-5
sequence of functions 8-11
session
automatic log file 1-6
session log
Command History 3-17
diary 3-12
set method 22-13
setfield 20-9
setting breakpoints 7-22
shadowed functions 5-15

shell escape 3-12
shell escape functions 17-70
shiftdim 19-2
shockbvp demo 15-68
short 6-55
short integer 6-55
shortcut
MATLAB 1-3
shortcuts 7-40
shortcuts, keyboard 2-18
Show Cell Markers command 10-31
shut down
MATLAB 1-15
options 1-15
simple inheritance 22-35
Simulink
interfacing files with source control systems 9-2
sin 17-7, 19-14
single data type 17-23
single precision 6-55
single process 3-3
single-precision matrix 17-22
singular value matrix decomposition 11-39
size
structure arrays 20-10
structure fields 20-10
size 16-25, 19-8, 20-10
sizing windows in the desktop 2-9
smallest value system can represent 17-21
smart recall 3-7
solution changes, rapid (BVP example) 15-68, 15-72
solution statistics (BVP) 15-81
solving linear systems of equations
sparse 16-36
sort 16-29
source control system
current system 9-4
interfacing with 9-2
preferences 9-4
specifying 9-4
supported systems 9-2
SourceSafe project hierarchy 9-6
spaces in MATLAB commands 3-4
spacing
output in Command Window 3-14 tabs in Command Window 3-14
sparse 16-7, 16-25

sparse data type 17-24
sparse matrix 17-22
advantages 16-6
and complex values 16-7
Cholesky factorization 16-33
computational considerations 16-25 contents 16-13
conversion from full 16-3, 16-7
creating 16-7
directly 16-8
from diagonal elements 16-10
defined 16-2
density 16-7
distance between nodes 16-22
eigenvalues 16-39
elementary 16-3
example 16-9
fill-in 16-22
importing 16-12
linear algebra 16-4
linear equations 16-5
linear systems of equations 16-36
LU factorization 16-30
and reordering 16-31
mathematical operations 16-25
nonzero elements 16-13
maximum number 16-9
specifying when creating matrix 16-8 storage 16-6, 16-13
values 16-13
nonzero elements of sparse matrix number of 16-13
operations 16-25
permutation 16-26
preconditioner 16-35
propagation through computations 16-25
QR factorization 16-33
reordering 16-4, 16-26
storage 16-6
for various permutations 16-28 viewing 16-13
theoretical graph 16-17
triangular factorization 16-30
viewing contents graphically 16-15 viewing storage 16-13
visualizing 16-21
working with 16-3
sparse ODE examples
Brusselator system (brussode) 15-38
spconvert 16-12
spdiags 16-10

special values 17-21
speye 16-25, 16-28, 16-32
splash screen
UNIX startup option 1-9
Windows startup option 1-6
spones 16-28
spparms 16-37
sprand 16-25
spreadsheet data
importing 6-23
sprintf 6-62
spy 16-15
spy plot 16-21
square brackets
for output arguments 17-9
squeeze 19-2, 19-11, 19-15
sscanf 6-60
stack
viewing 5-8
starting MATLAB
DOS 1-3
UNIX 1-3
Windows 1-3
startup
directory for MATLAB 1-3
changing 1-4
files for MATLAB 1-5
M-files open 7-37
options for MATLAB 1-5
UNIX 1-7
Windows 1-5
startup cost
minimizing for ODE solvers 15-52
startup.m 1-5
Stateflow files
source control 9-2
statements
conditional 17-14
long (on multiple lines) 3-5
statistical data
missing values 13-14
normalizing 13-23
outliers 13-15
preprocessing 13-14
removingNaNs 13-15
See also multivariate data

See also univariate data
statistics
descriptive 13-8
step size (ODE) 15-25
initial step size 15-26
upper bound 15-26
stepping through M-file 7-25
stiff ODE examples
Brusselator system (brussode) 15-38 differential-algebraic problem (hb1dae) 15-47 finite element dicretization (fem1ode) 15-35 van der Pol (vdpode) 15-33
stiffness (ODE), defined 15-14
stopping a running program 3-11
storage
array 17-51
for various permutations of sparse matrix 16-28
of sparse matrix 16-6
sparse and full, comparison 16-7
viewing for sparse matrix 16-13

str2func
description 21-22
example 21-23

strcmp 18-10
strings
across multiple lines 3-5
color indicators 2-24
See also character arrays
struct data type 17-24
structs 20-5, 20-6, 20-17
structure arrays 20-4
accessing data 20-7, 20-9
adding fields 20-10
applying functions to 20-10
building 20-5
usingstructs 20-6
data organization 20-12
deleting fields 20-10
element-by-element organization 20-15
expanding 20-5, 20-6
fields 20-4
assigning data to 20-5
assigning usingsetfield 20-9
indexing
nested structures 20-18
within fields 20-8
multidimensional 19-19

applying functions 19-20
nesting 20-17
obtaining field names 20-6
organizing data 20-12
example 20-16
overview 20-4
plane organization 20-14
preallocation 17-74
size 20-10
subarrays, accessing 20-8
subscripting 20-5
used with classes 22-8
within cell arrays 20-31
writing M-files for 20-11
example 20-12
structures
See also structure arrays
styles in M-book
Normal 10-18
subassign 22-17
subfunction, going to in M-file 7-11
subfunctions 17-43
accessing 17-43
creating 17-43
debugging 17-44
definition line 17-43
function handles to 21-4, 21-19
in dispatching priority 17-12
precedence of 22-69
subsasgn 22-14
subscripted assignment 22-17
subscripting 17-46
how MATLAB calculates indices 17-53
multidimensional arrays 19-3
overloading 22-14
page 19-3
structure arrays 20-5
with logical expression 17-32
with thefind function 17-32
subsref 22-14
subsref method 22-14
substring within a string 18-13
subtraction operator 17-26
suggestions to The MathWorks 4-32
sum 16-26, 16-29, 19-14
sunspot periodicity
calculating using Fourier transforms 13-45
superior function 22-66

superseding existing M-files names 17-44 suppressing output 3-9
switch 17-37
case groupings 17-37
example 17-38
multiple conditions 17-38
symamd 16-29
symmetric
matrix 11-8
symmmd 16-29, 16-31
symrcm 16-29, 16-31
syntax
color indicators 2-24
errors 7-17
highlighting 7-8
syntax coloring and indenting 3-5
system requirements 1-3
systems of equations. *See* linear systems of equations

T

tab
completion of line 3-8
spacing in Command Window 3-14 tabbing desktop windows together 2-14 table of contents for help 4-9
tabs 7-9, 7-40
tabs in string arrays 18-12
Technical Support
contacting 4-32
searching Online Knowledge Base 4-13 technical support Web page 2-20
tempdir 6-53
templates
M-book 10-18
specifying 10-25
tempname 6-53
temporary files
creating 6-53
terminating a running program 3-11
tessellations, multidimensional
Delaunay 12-30
Voronoi diagrams 12-32
text
converting to input cells 10-27
finding in page in Help browser 4-19
preferences in MATLAB 2-23
styles in M-book 10-18
text editor, setting as default 7-36
text files
importing 6-4

reading 6-58
theoretical graph 16-17
example 16-18
node 16-17
three-dimensional interpolation 12-17 time measured for M-files 8-2
numbers 17-63
time-out message
while evaluating multiple input cells in an M-book 10-29
Toggle Graph Output for Cell command 10-31 token in string 18-13
tolerance 17-21
toolbar, desktop 2-17
toolbox path cache 1-10
enabling 1-12
generating 1-12
preferences 1-10
updating 1-13
tools in desktop
description 2-2
running from Launch Pad 2-5
See also windows in desktop
 tooltips 2-17
preference in MATLAB 2-23
transfer functions
using partial fraction expansion 12-8 transpose
complex conjugate 11-8
unconjugated complex 11-8
transpose 19-13
triangular factorization
for sparse matrices 16-30
triangular matrix 11-25
triangulation
closest point searches 12-25
Delaunay 12-21
scattered data 12-19
Voronoi diagrams 12-26
See also tessellation
tricubic interpolation 12-17
trigonometric functions 17-7, 19-14 trilinear interpolation 12-17
troubleshooting (ODE) 15-50
twobvp demo 15-58
two-dimensional interpolation 12-13
comparing methods graphically 12-14 type ahead feature 3-7

U

uchar data type 6-55
uint data type 17-24

unary minus operator 17-26 Undefine Cells command 10-32 underdetermined

rectangular matrices 11-18 undo in MATLAB 2-17

undocheckout 9-11

undocking windows from desktop 2-12

Ungroup Cells command 10-32

unitary matrices

QR factorization 11-28

univariate data 13-4

unknown parameters (BVP) 15-67

example 15-63

unregserver startup option 1-6

updates to products 2-20

updating optimization code to MATLAB Version 5

syntax 14-16

uppercase usage in MATLAB 3-4

Use 16-Color Figures check box 10-23

user classes, designing 22-9

user input

obtaining interactively 17-69

UserObject data type 17-24

utilities, running from MATLAB 3-12

V

value

largest system can represent 17-21

values

data type 6-54

examining 7-26

van der Pol example 15-33

simple, nonstiff 15-11

simple, stiff 15-14

varargin 17-17, 20-27

in argument list 17-18

unpacking contents 17-17

varargout 17-17

in argument list 17-18

packing contents 17-17

variable-order solver (ODE) 15-30

variables

clearing 5-8
deleting
and memory use 17-75

dispatching priority 17-12
displaying values of 5-11
editing values for 5-10
global 17-19

rules for use 17-20
graphing from the Workspace browser 5-9 local 17-19
memory usage 17-76
naming 5-14, 17-19
persistent 17-20
replacing list with a cell array 20-25 saving 5-5
storage in memory 17-75
viewing during execution 7-17
workspace 5-3

vdpode demo 15-33
vector
of dates 17-65
preallocation 17-73
vector products
dot or scalar 11-8
outer and inner 11-7
vectorization 17-71
example 17-71
replacing for
vectorization 17-35
vectorizing ODE function (BVP) 15-79 vectors
column and row 11-6
multiplication 11-7
vehicle traffic sample data 13-4
version 17-21
version control system. See source control system version information 4-32
version startup option for UNIX 1-7
version, obtaining 17-21
viewing desktop tools 2-7
Visible figure property
embedding graphics in M-book 10-21 visualizing
cell array 20-21
sparse matrix
spy plot 16-21
visualizing solver results
BVP 15-66
ODE 15-12

PDE 15-91

Voronoi diagrams

multidimensional 12-32

two-dimensional 12-26

W

warnings 17-59

displaying 17-60

Web

accessing from MATLAB 2-20

Web browser in MATLAB 4-4

Web site for The MathWorks 2-20

what 5-22

which used with methods 22-70

while 17-39

empty arrays 17-39

example 17-39

syntax 17-39

white space

finding in string 18-12

who 5-5

whos 5-5, 16-7, 19-8

interpreting memory use 17-75

Windows

MATLAB use of system resources 17-77

windows in desktop

about 2-2

arrangement 2-7

closing 2-9

docking 2-10

moving 2-10

opening 2-7

sizing 2-9

undocking 2-12

winword.exe 10-24

Word documents

converting to M-book 10-5

work directory 1-3

working directory 5-20

workspace

base 5-8

clearing 5-8

context 17-14

defined 5-3

functions 5-8

initializing in an M-book 10-10

loading 5-7
M-book contamination 10-6
of individual functions 17-14
opening 5-7
protecting integrity 10-6
saving 5-5
tool 5-3
viewing 5-4
viewing during execution 7-17
Workspace browser
description 5-3
preferences 5-9
wrap lines 3-15
writing
ASCII data 6-16 HDF data 6-44
in HDF format 6-41
writing data to disk file overview 6-2

X

xor 17-31
Xserver option for UNIX 1-7

Z

zeros
of mathematical functions 14-12 zeros 16-25, 19-6