

MATLAB®

Programming Fundamentals

R2012a

MATLAB®

How to Contact MathWorks



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup
www.mathworks.com/contact_TS.html Technical Support



suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB Programming Fundamentals

© COPYRIGHT 1984–2012 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

June 2004	First printing	New for MATLAB 7.0 (Release 14)
October 2004	Online only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online only	Revised for MATLAB 7.0.4 (Release 14SP2)
June 2005	Second printing	Minor revision for MATLAB 7.0.4
September 2005	Online only	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Revised for MATLAB 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online only	Revised for MATLAB 7.4 (Release 2007a)
September 2007	Online only	Revised for Version 7.5 (Release 2007b)
March 2008	Online only	Revised for Version 7.6 (Release 2008a)
October 2008	Online only	Revised for Version 7.7 (Release 2008b)
March 2009	Online only	Revised for Version 7.8 (Release 2009a)
September 2009	Online only	Revised for Version 7.9 (Release 2009b)
March 2010	Online only	Revised for Version 7.10 (Release 2010a)
September 2010	Online only	Revised for Version 7.11 (Release 2010b)
April 2011	Online only	Revised for Version 7.12 (Release 2011a)
September 2011	Online only	Revised for Version 7.13 (Release 2011b)
March 2012	Online only	Revised for Version 7.14 (Release 2012a)

Language

Syntax Basics

1

Create Variables	1-2
Create Numeric Arrays	1-3
Store Text in Character Strings	1-5
Enter Multiple Statements on One Line	1-6
Continue Long Statements on Multiple Lines	1-7
Call Functions	1-8
Ignore Function Outputs	1-9
Variable Names	1-10
Valid Names	1-10
Conflicts with Function Names	1-10
Case and Space Sensitivity	1-12
Command vs. Function Syntax	1-13
Command and Function Syntaxes	1-13
Avoid Common Syntax Mistakes	1-14
How MATLAB Recognizes Command Syntax	1-15

Operators	2-2
Arithmetic Operators	2-2
Relational Operators	2-3
Logical Operators	2-4
Operator Precedence	2-11
Special Values	2-13
Conditional Statements	2-15
Loop Control Statements	2-17
Dates and Times	2-19
Representing Dates and Times in MATLAB	2-19
Date and Time Functions	2-20
Working with Date Strings	2-21
Date String Tables	2-26
Working with Date Vectors	2-29
Working with Serial Date Numbers	2-33
Other Considerations	2-36
Function Summary	2-38
Regular Expressions	2-40
Overview	2-40
Calling Regular Expression Functions from MATLAB	2-42
Parsing Strings with Regular Expressions	2-46
Other Benefits of Using Regular Expressions	2-50
Metacharacters and Operators	2-51
Character Type Operators	2-53
Character Representation	2-57
Grouping Operators	2-58
Nonmatching Operators	2-60
Positional Operators	2-61
Lookaround Operators	2-62
Quantifiers	2-68
Tokens	2-71
Named Capture	2-76
Conditional Expressions	2-78

Dynamic Regular Expressions	2-80
String Replacement	2-89
Handling Multiple Strings	2-91
Function, Mode Options, Operator, Return Value Summaries	2-91
Comma-Separated Lists	2-100
What Is a Comma-Separated List?	2-100
Generating a Comma-Separated List	2-100
Assigning Output from a Comma-Separated List	2-102
Assigning to a Comma-Separated List	2-103
How to Use the Comma-Separated Lists	2-104
Fast Fourier Transform Example	2-106
String Evaluation	2-108
Evaluate Expressions in Text Strings	2-108
Alternatives to the eval Function	2-108
Shell Escape Functions	2-113
Symbol Reference	2-114
Asterisk — *	2-115
At — @	2-115
Colon — :	2-116
Comma — ,	2-117
Curly Braces — {}	2-118
Dot —	2-118
Dot-Dot —	2-119
Dot-Dot-Dot (Ellipsis) —	2-119
Dot-Parentheses — .()	2-121
Exclamation Point — !	2-121
Parentheses — ()	2-121
Percent — %	2-122
Percent-Brace — %{ %}	2-123
Plus — +	2-123
Semicolon — ;	2-123
Single Quotes — ' '	2-124
Space Character	2-125
Slash and Backslash — / \	2-125
Square Brackets — []	2-126
Tilde — ~	2-126

Classes (Data Types)

Overview of MATLAB Classes

3

Fundamental MATLAB Classes	3-2
How to Use the Different Classes	3-4

Numeric Classes

4

Overview of Numeric Classes	4-2
Integers	4-3
Integer Classes	4-3
Creating Integer Data	4-4
Arithmetic Operations on Integer Classes	4-5
Largest and Smallest Values for Integer Classes	4-6
Integer Functions	4-6
Floating-Point Numbers	4-7
Double-Precision Floating Point	4-7
Single-Precision Floating Point	4-8
Creating Floating-Point Data	4-8
Arithmetic Operations on Floating-Point Numbers	4-10
Largest and Smallest Values for Floating-Point Classes ..	4-11
Accuracy of Floating-Point Data	4-12
Avoiding Common Problems with Floating-Point	
Arithmetic	4-14
Floating-Point Functions	4-16
References	4-16
Complex Numbers	4-18
Creating Complex Numbers	4-18
Complex Number Functions	4-19

Infinity and NaN	4-20
Infinity	4-20
NaN	4-20
Infinity and NaN Functions	4-21
Identifying Numeric Classes	4-22
Display Format for Numeric Values	4-23
Default Display	4-23
Display Format Examples	4-23
Setting Numeric Format in a Program	4-24
Function Summary	4-26

The Logical Class

5

Overview of the Logical Class	5-2
Identifying Logical Arrays	5-4
Function Summary	5-4
Examples of Identifying Logical Arrays	5-4
Functions that Return a Logical Result	5-6
Overview	5-6
Examples of Functions that Return a Logical Result	5-6
Using Logical Arrays in Conditional Statements	5-9
Using Logical Arrays in Indexing	5-10

Creating Character Arrays	6-2
Creating a Character String	6-2
Creating a Rectangular Character Array	6-3
Identifying Characters in a String	6-4
Working with Space Characters	6-5
Expanding Character Arrays	6-6
Cell Arrays of Strings	6-7
Converting to a Cell Array of Strings	6-7
Functions for Cell Arrays of Strings	6-8
Formatting Strings	6-10
Functions that Use Format Strings	6-10
The Format String	6-11
Input Value Arguments	6-12
The Formatting Operator	6-13
Constructing the Formatting Operator	6-14
Setting Field Width and Precision	6-20
Restrictions for Using Identifiers	6-23
String Comparisons	6-25
Comparing Strings for Equality	6-25
Comparing for Equality Using Operators	6-26
Categorizing Characters Within a String	6-27
Searching and Replacing	6-28
Converting from Numeric to String	6-30
Function Summary	6-30
Converting to a Character Equivalent	6-31
Converting to a String of Numbers	6-31
Converting to a Specific Radix	6-31
Converting from String to Numeric	6-32
Function Summary	6-32
Converting from a Character Equivalent	6-33
Converting from a Numeric String	6-33

Converting from a Specific Radix	6-34
Function Summary	6-35

Structures

7

Create a Structure Array	7-2
Access Data in a Structure Array	7-6
Concatenate Structures	7-9
Generate Field Names from Variables	7-11
Access Data in Nested Structures	7-12
Access Multiple Elements of a Nonscalar Struct Array	7-14
Ways to Organize Data in Structure Arrays	7-16
Plane Organization	7-16
Element-by-Element Organization	7-18
Memory Requirements for a Structure Array	7-20

Cell Arrays

8

What Is a Cell Array?	8-2
Create a Cell Array	8-3

Access Data in a Cell Array	8-5
Add Cells to a Cell Array	8-8
Delete Data from a Cell Array	8-9
Combine Cell Arrays	8-10
Pass Contents of Cell Arrays to Functions	8-11
Preallocate Memory for a Cell Array	8-14
Cell vs. Struct Arrays	8-15
Multilevel Indexing to Access Parts of Cells	8-17

Function Handles

9

What Is a Function Handle?	9-2
Creating a Function Handle	9-3
Maximum Length of a Function Name	9-4
The Role of Scope, Precedence, and Overloading When Creating a Function Handle	9-4
Obtaining Permissions from Class Methods	9-5
Using Function Handles for Anonymous Functions	9-6
Arrays of Function Handles	9-6
Calling a Function By Means of Its Handle	9-7
Calling Syntax	9-7
Calling a Function with Multiple Outputs	9-8
Returning a Handle for Use Outside of a Function File ...	9-8
Example — Using Function Handles in Optimization	9-9
Preserving Data from the Workspace	9-10

Preserving Data with Anonymous Functions	9-10
Preserving Data with Nested Functions	9-11
Applications of Function Handles	9-13
Example of Passing a Function Handle	9-13
Pass a Function to Another Function	9-13
Capture Data Values For Later Use By a Function	9-15
Call Functions Outside of Their Normal Scope	9-18
Save the Handle in a MAT-File for Use in a Later MATLAB Session	9-18
Saving and Loading Function Handles	9-19
Invalid or Obsolete Function Handles	9-19
Advanced Operations on Function Handles	9-20
Examining a Function Handle	9-20
Converting to and from a String	9-21
Comparing Function Handles	9-23
Functions That Operate on Function Handles	9-27

Map Containers

10

Overview of the Map Data Structure	10-2
Description of the Map Class	10-4
Properties of the Map Class	10-4
Methods of the Map Class	10-5
Creating a Map Object	10-6
Constructing an Empty Map Object	10-6
Constructing An Initialized Map Object	10-7
Combining Map Objects	10-8
Examining the Contents of the Map	10-9

Reading and Writing Using a Key Index	10-11
Reading From the Map	10-11
Adding Key/Value Pairs	10-12
Building a Map with Concatenation	10-13
Modifying Keys and Values in the Map	10-15
Removing Keys and Values from the Map	10-15
Modifying Values	10-15
Modifying Keys	10-16
Modifying a Copy of the Map	10-16
Mapping to Different Value Types	10-18
Mapping to a Structure Array	10-18
Mapping to a Cell Array	10-19

Combining Unlike Classes

11

Valid Combinations of Unlike Classes	11-2
Combining Unlike Integer Types	11-3
Overview	11-3
Example of Combining Unlike Integer Sizes	11-4
Example of Combining Signed with Unsigned	11-4
Combining Integer and Noninteger Data	11-6
Combining Cell Arrays with Non-Cell Arrays	11-7
Empty Matrices	11-8
Concatenation Examples	11-9
Combining Single and Double Types	11-9
Combining Integer and Double Types	11-9
Combining Character and Double Types	11-10
Combining Logical and Double Types	11-10

MATLAB Objects	12-2
Getting Oriented	12-2
Getting Comfortable with Objects	12-2
What Are Objects and Why Use Them?	12-2
Accessing Objects	12-3
Objects In the MATLAB Language	12-3
Other Kinds of Objects Used by MATLAB	12-4
General Purpose Vs. Specialized Arrays	12-5
How They Differ	12-5
Using General-Purpose Data Structures	12-5
Using Specialized Objects	12-6
Key Object Concepts	12-8
Basic Concepts	12-8
Classes Describe How to Create Objects	12-8
Properties Contain Data	12-9
Methods Implement Operations	12-9
Events are Notices Broadcast to Listening Objects	12-10
Creating Objects	12-11
Class Constructor	12-11
When to Use Package Names	12-11
Accessing Object Data	12-14
Listing Public Properties	12-14
Getting Property Values	12-14
Setting Property Values	12-15
Calling Object Methods	12-16
What Operations Can You Perform	12-16
Method Syntax	12-16
Class of Objects Returned by Methods	12-18
Desktop Tools Are Object Aware	12-19
Tab Completion Works with Objects	12-19
Editing Objects with the Variable Editor	12-19

Getting Information About Objects	12-21
The Class of Workspace Variables	12-21
Information About Class Members	12-23
Logical Tests for Objects	12-23
Displaying Objects	12-24
Getting Help for MATLAB Objects	12-25
Copying Objects	12-26
Two Copy Behaviors	12-26
Value Object Copy Behavior	12-26
Handle Object Copy Behavior	12-27
Testing for Handle or Value Class	12-31
Destroying Objects	12-33
Object Lifecycle	12-33
Difference Between clear and delete	12-33

Defining Your Own Classes

13

Scripts and Functions

Program Files

14

Program Development	14-2
Overview	14-2
Creating a Program	14-2
Getting the Bugs Out	14-3
Cleaning Up the Program	14-4
Improving Performance	14-5
Checking It In	14-6
Protecting Your Source Code	14-6
Working with Functions in Files	14-9
Overview	14-9

Types of Program Files	14-9
Basic Parts of a Program File	14-10
Creating a Program File	14-15
Providing Help for Your Program	14-17
Cleaning Up When the Function Completes	14-17
Scripts and Functions	14-24
Scripts	14-24
Functions	14-25
Types of Functions	14-26
Organizing Your Functions	14-27
Identifying Dependencies	14-27
Base and Function Workspaces	14-29
Share Data Between Workspaces	14-30
Best Practice: Passing Arguments	14-30
Nested Functions	14-31
Persistent Variables	14-32
Global Variables	14-32
Evaluating in Another Workspace	14-33
Calling Functions	14-35
What Happens When You Call a Function	14-35
Function Precedence Order	14-35
Resolving Difficulties In Calling Functions	14-38
Calling External Functions	14-43
Running External Programs	14-43
Functions Provided By MATLAB	14-44
Overview	14-44
Functions	14-44
Built-In Functions	14-45
Overloaded MATLAB Functions	14-46
Internal Utility Functions	14-47

Overview of MATLAB Function Types	15-2
Anonymous Functions	15-3
Constructing an Anonymous Function	15-3
Arrays of Anonymous Functions	15-6
Outputs from Anonymous Functions	15-7
Variables Used in the Expression	15-8
Examples of Anonymous Functions	15-11
Primary Functions	15-15
Nested Functions	15-16
Writing Nested Functions	15-16
Calling Nested Functions	15-18
Variable Scope in Nested Functions	15-19
Using Function Handles with Nested Functions	15-21
Restrictions on Assigning to Variables	15-26
Examples of Nested Functions	15-27
Subfunctions	15-33
Overview	15-33
Calling Subfunctions	15-34
Accessing Help for a Subfunction	15-34
Private Functions	15-35
Overview	15-35
Private Folders	15-35
Accessing Help for a Private Function	15-36
Overloaded Functions	15-37

16

Find Number of Function Arguments	16-2
Support Variable Number of Inputs	16-4
Support Variable Number of Outputs	16-6
Validate Number of Function Arguments	16-8
Argument Checking in Nested Functions	16-11
Ignore Function Inputs	16-13
Check Function Inputs with validateattributes	16-14
Parse Function Inputs	16-17
Input Parser Validation Functions	16-22

17

Introduction	17-2
Command and Function Syntax	17-3
Syntax Help	17-3
Command and Function Syntaxes	17-3
Command Line Continuation	17-3
Completing Commands Using the Tab Key	17-4
Recalling Commands	17-4
Clearing Commands	17-5
Suppressing Output to the Screen	17-5

Help	17-6
Using the Help Browser	17-6
Help on Functions from the Help Browser	17-6
Help on Functions from the Command Window	17-7
Topical Help	17-7
Paged Output	17-8
Writing Your Own Help	17-8
Help for Subfunctions and Private Functions	17-8
Help for Methods and Overloaded Functions	17-9
Development Environment	17-10
Workspace Browser	17-10
Using the Find Utility	17-10
Commenting Out a Block of Code	17-11
Creating Functions from Command History	17-11
Editing Functions in EMACS	17-11
Functions	17-12
Function Structure	17-12
Using Lowercase for Function Names	17-12
Getting a Function's Name and Path	17-13
What Files Does a Function Use?	17-13
Dependent Functions, Built-Ins, Classes	17-14
Function Arguments	17-15
Getting the Input and Output Arguments	17-15
Variable Numbers of Arguments	17-15
String or Numeric Arguments	17-16
Passing Arguments in a Structure	17-16
Passing Arguments in a Cell Array	17-16
Program Development	17-18
Planning the Program	17-18
Using Pseudo-Code	17-18
Selecting the Right Data Structures	17-18
General Coding Practices	17-19
Naming a Function Uniquely	17-19
The Importance of Comments	17-19
Coding in Steps	17-20
Making Modifications in Steps	17-20
Functions with One Calling Function	17-20
Testing the Final Program	17-20

Debugging	17-21
The MATLAB Debug Functions	17-21
More Debug Functions	17-21
The MATLAB Graphical Debugger	17-22
A Quick Way to Examine Variables	17-22
Setting Breakpoints from the Command Line	17-22
Finding Line Numbers to Set Breakpoints	17-23
Stopping Execution on an Error or Warning	17-23
Locating an Error from the Error Message	17-23
Using Warnings to Help Debug	17-23
Making Code Execution Visible	17-24
Debugging Scripts	17-24
Variables	17-25
Rules for Variable Names	17-25
Making Sure Variable Names Are Valid	17-25
Do Not Use Function Names for Variables	17-26
Checking for Reserved Keywords	17-26
Avoid Using i and j for Variables	17-26
Avoid Overwriting Variables in Scripts	17-27
Persistent Variables	17-27
Protecting Persistent Variables	17-27
Global Variables	17-27
Strings	17-29
Creating Strings with Concatenation	17-29
Comparing Methods of Concatenation	17-29
Store Arrays of Strings in a Cell Array	17-30
Converting Between Strings and Cell Arrays	17-30
Search and Replace Using Regular Expressions	17-30
Evaluating Expressions	17-32
Find Alternatives to Using eval	17-32
Assigning to a Series of Variables	17-32
Short-Circuit Logical Operators	17-32
Changing the Counter Variable within a for Loop	17-33
MATLAB Path	17-34
Precedence Rules	17-34
Adding a Folder to the Search Path	17-35
Handles to Functions Not on the Path	17-35
Making Toolbox File Changes Visible to MATLAB	17-36

Making Nontoolbox File Changes Visible to MATLAB	17-36
Change Notification on Windows	17-37
Program Control	17-38
Using break, continue, and return	17-38
Using switch Versus if	17-39
MATLAB case Evaluates Strings	17-39
Multiple Conditions in a case Statement	17-39
Implicit Break in switch-case	17-39
Variable Scope in a switch	17-40
Catching Errors with try-catch	17-40
Nested try-catch Blocks	17-41
Forcing an Early Return from a Function	17-41
Save and Load	17-42
Saving Data from the Workspace	17-42
Loading Data into the Workspace	17-42
Viewing Variables in a MAT-File	17-43
Appending to a MAT-File	17-43
Save and Load on Startup or Quit	17-44
Saving to an ASCII File	17-44
Files and Filenames	17-45
Naming Functions	17-45
Naming Other Files	17-45
Passing Filenames as Arguments	17-46
Passing Filenames to ASCII Files	17-46
Determining Filenames at Run-Time	17-46
Returning the Size of a File	17-46
Input/Output	17-48
Common I/O Functions	17-48
Loading Mixed Format Data	17-48
Reading Files with Different Formats	17-49
Interactive Input into Your Program	17-49
Starting MATLAB	17-50
Getting MATLAB to Start Up Faster	17-50
Operating System Compatibility	17-51
Executing O/S Commands from MATLAB	17-51

Searching Text with grep	17-51
Constructing Paths and Filenames	17-51
Finding the MATLAB Root Folder	17-52
Temporary Directories and Filenames	17-52
For More Information	17-53
Current CSSM	17-53
Archived CSSM	17-53
MATLAB Technical Support	17-53
MATLAB Central	17-53
MATLAB Newsletters (Digest, News & Notes)	17-53
MATLAB Documentation	17-53
MATLAB Index of Examples	17-54

Software Development

Error Handling

18

Error Reporting in a MATLAB Application	18-2
Overview	18-2
Getting an Exception at the Command Line	18-2
Getting an Exception in Your Program Code	18-3
Generating a New Exception	18-4
Capturing Information About the Error	18-5
Overview	18-5
The MException Class	18-5
Properties of the MException Class	18-7
Methods of the MException Class	18-14
Throwing an Exception	18-16
Responding to an Exception	18-18
Overview	18-18
The try-catch Statement	18-18
Suggestions on How to Handle an Exception	18-20

Warnings	18-23
Reporting a Warning	18-23
Identifying the Cause	18-24
Warning Control	18-25
Overview	18-25
Warning Statements	18-26
Warning Control Statements	18-27
Output from Control Statements	18-30
Saving and Restoring State	18-32
Backtrace and Verbose Modes	18-33
Debugging Errors and Warnings	18-37

Program Scheduling

19

Using a MATLAB Timer Object	19-2
Overview	19-2
Example: Displaying a Message	19-3
Creating Timer Objects	19-5
Creating the Object	19-5
Naming the Object	19-6
Working with Timer Object Properties	19-7
Retrieving the Value of Timer Object Properties	19-7
Setting the Value of Timer Object Properties	19-8
Starting and Stopping Timers	19-10
Starting a Timer	19-10
Starting a Timer at a Specified Time	19-10
Stopping Timer Objects	19-11
Blocking the MATLAB Command Line	19-12
Creating and Executing Callback Functions	19-14
Associating Commands with Timer Object Events	19-14
Creating Callback Functions	19-15

Specifying the Value of Callback Function Properties	19-17
Timer Object Execution Modes	19-19
Executing a Timer Callback Function Once	19-19
Executing a Timer Callback Function Multiple Times	19-20
Handling Callback Function Queuing Conflicts	19-21
Deleting Timer Objects from Memory	19-23
Deleting One or More Timer Objects	19-23
Testing the Validity of a Timer Object	19-23
Finding Timer Objects in Memory	19-24
Finding All Timer Objects	19-24
Finding Invisible Timer Objects	19-24

Performance

20

Analyzing Your Program's Performance	20-2
Overview	20-2
The Profiler Utility	20-2
Stopwatch Timer Functions	20-2
Techniques for Improving Performance	20-4
Preallocating Arrays	20-4
Limiting Size and Complexity	20-5
Assigning to Variables	20-6
Using Appropriate Logical Operators	20-7
Overloading Built-In Functions	20-7
Functions Are Generally Faster Than Scripts	20-8
Load and Save Are Faster Than File I/O Functions	20-8
Vectorizing Loops	20-8
Avoid Large Background Processes	20-11

21

Memory Allocation 21-2
 Memory Allocation for Arrays 21-2
 Data Structures and Memory 21-6

Memory Management Functions 21-12
 The whos Function 21-13

Strategies for Efficient Use of Memory 21-15
 Ways to Reduce the Amount of Memory Required 21-15
 Using Appropriate Data Storage 21-17
 How to Avoid Fragmenting Memory 21-20
 Reclaiming Used Memory 21-21

Resolving “Out of Memory” Errors 21-23
 General Suggestions for Reclaiming Memory 21-23
 Setting the Process Limit 21-24
 Disabling Java VM on Startup 21-25
 Increasing System Swap Space 21-26
 Using the 3GB Switch on Windows Systems 21-26
 Freeing Up System Resources on Windows Systems 21-27

Create Help and Demos

22

Types of Help You Can Create 22-2

Add Help for Your Program Files 22-4
 Help Within a Program File 22-4
 Help Summary for Your Program Files (Contents.m) 22-6
 Help for Classes You Create 22-7

Add Documentation to the Help Browser 22-12
 Types of Documentation You Can Provide 22-12
 Learning to Add Help from Examples 22-13

Summary of Creating and Installing HTML Help Files ...	22-14
Organizing Your Documentation	22-15
Creating Function Reference Pages	22-28
Creating Function and Block Category Listings	22-33
Making Your HTML Help Files Searchable	22-40
Summary of Workflow for Providing HTML Help Files ...	22-42
Add Demos to the Help Browser	22-48
About Creating Demos	22-48
Providing Demos to Others	22-56
Address Validation Errors for info.xml Files	22-57
About XML File Validation	22-57
Entities Missing or Out of Order in info.xml	22-57
Unrelated info.xml File	22-58
Invalid Constructs in info.xml File	22-58
Outdated info.xml File for a MathWorks Product	22-58

Index



Language

- Chapter 1, “Syntax Basics”
- Chapter 2, “Program Components”

Syntax Basics

- “Create Variables” on page 1-2
- “Create Numeric Arrays” on page 1-3
- “Store Text in Character Strings” on page 1-5
- “Enter Multiple Statements on One Line” on page 1-6
- “Continue Long Statements on Multiple Lines” on page 1-7
- “Call Functions” on page 1-8
- “Ignore Function Outputs” on page 1-9
- “Variable Names” on page 1-10
- “Case and Space Sensitivity” on page 1-12
- “Command vs. Function Syntax” on page 1-13

Create Variables

This example shows several ways to assign a value to a variable.

```
x = 5.71;  
A = [1 2 3; 4 5 6; 7 8 9];  
I = besseli(x,A);
```

You do not have to declare variables before assigning values.

If you do not end an assignment statement with a semicolon (;), MATLAB® displays the result in the Command Window. For example,

```
x = 5.71
```

displays

```
x =  
    5.7100
```

If you do not explicitly assign the output of a command to a variable, MATLAB generally assigns the result to the reserved word `ans`. For example,

```
5.71
```

returns

```
ans =  
    5.7100
```

The value of `ans` changes with every command that returns an output value that is not assigned to a variable.

Create Numeric Arrays

This example shows how to create a numeric variable. In the MATLAB computing environment, all variables are arrays, and by default, numeric variables are of type `double` (that is, double-precision values). For example, create a scalar value.

```
A = 100;
```

Because scalar values are single element, 1-by-1 arrays,

```
whos A
```

returns

Name	Size	Bytes	Class	Attributes
A	1x1	8	double	

To create a *matrix* (a two-dimensional, rectangular array of numbers), you can use the `[]` operator.

```
B = [12, 62, 93, -8, 22; 16, 2, 87, 43, 91; -4, 17, -72, 95, 6]
```

When using this operator, separate columns with a comma or space, and separate rows with a semicolon. All rows must have the same number of elements. In this example, B is a 3-by-5 matrix (that is, B has three rows and five columns).

```
B =
    12    62    93    -8    22
    16     2    87    43    91
    -4    17   -72    95     6
```

A matrix with only one row or column (that is, a 1-by-n or n-by-1 array) is a *vector*, such as

```
C = [1, 2, 3]
```

or

```
D = [10; 20; 30]
```

For more information, see:

- “Multidimensional Arrays”
- “Matrix Indexing”

Store Text in Character Strings

This example shows how to create a character string.

```
myString = 'Hello, world';
```

If the text contains a single quotation mark, include two quotation marks within the string definition:

```
otherString = 'You''re right';
```

In the MATLAB computing environment, all variables are arrays, and strings are of type `char` (character arrays). For example,

```
whos myString
```

returns

Name	Size	Bytes	Class	Attributes
myString	1x12	24	char	

Enter Multiple Statements on One Line

This example shows how to enter more than one command on the same line.

```
A = magic(5), B = ones(5) * 4.7; C = A./B
```

To distinguish between commands, end each one with a comma or semicolon. Commands that end with a comma display their results, while commands that end with a semicolon do not.

A =

```
17    24     1     8    15
23     5     7    14    16
 4     6    13    20    22
10    12    19    21     3
11    18    25     2     9
```

C =

```
3.6170    5.1064    0.2128    1.7021    3.1915
4.8936    1.0638    1.4894    2.9787    3.4043
0.8511    1.2766    2.7660    4.2553    4.6809
2.1277    2.5532    4.0426    4.4681    0.6383
2.3404    3.8298    5.3191    0.4255    1.9149
```

Continue Long Statements on Multiple Lines

This example shows how to continue a statement to the next line using ellipses (...).

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

Build a long character string by concatenating shorter strings together:

```
mystring = ['Accelerating the pace of ' ...  
           'engineering and science'];
```

The start and end quotation marks for a string must appear on the same line. For example, this code returns an error, because each line contains only one quotation mark:

```
mystring = 'Accelerating the pace of ...  
           engineering and science'
```

An ellipses outside a quoted string is equivalent to a space. For example,

```
x = [1.23...  
     4.56];
```

is the same as

```
x = [1.23 4.56];
```

Call Functions

These examples show how to call a MATLAB function. To run the examples, you must first create numeric arrays A and B, such as:

```
A = [1 3 5];  
B = [10 6 4];
```

Enclose inputs to functions in parentheses:

```
max(A)
```

Separate multiple inputs with commas:

```
max(A,B)
```

Store output from a function by assigning it to a variable:

```
maxA = max(A)
```

Enclose multiple outputs in square brackets:

```
[maxA, location] = max(A)
```

Call a function that does not require any inputs, and does not return any outputs, by typing only the function name:

```
clc
```

Enclose text string inputs in single quotation marks:

```
disp('hello world')
```

Ignore Function Outputs

This example shows how to request specific outputs from a function.

Request all three possible outputs from the `fileparts` function.

```
helpFile = which('help');  
[helpPath,name,ext] = fileparts(helpFile);
```

The current workspace now contains three variables from `fileparts`: `helpPath`, `name`, and `ext`. In this case, the variables are small. However, some functions return results that use much more memory. If you do not need those variables, they waste space on your system.

Request only the first output, ignoring the second and third.

```
helpPath = fileparts(helpFile);
```

For any function, you can request only the first N outputs (where N is less than or equal to the number of possible outputs) and ignore any remaining outputs. If you request more than one output, enclose the variable names in square brackets, `[]`.

Ignore the first output using a tilde (`~`).

```
[~,name,ext] = fileparts(helpFile);
```

You can ignore any number of function outputs, in any position in the argument list. Separate consecutive tildes with a comma, such as

```
[~,-,ext] = fileparts(helpFile);
```

Variable Names

In this section...

“Valid Names” on page 1-10

“Conflicts with Function Names” on page 1-10

Valid Names

A valid variable name starts with a letter, followed by letters, digits, or underscores. MATLAB is case sensitive, so *A* and *a* are *not* the same variable. The maximum length of a variable name is the value that the `namelengthmax` command returns.

You cannot define variables with the same names as MATLAB keywords, such as `if` or `end`. For a complete list, run the `iskeyword` command.

Examples of valid names:

`x6`

`lastValue`

`n_factorial`

Invalid names:

`6x`

`end`

`n!`

Conflicts with Function Names

Avoid creating variables with the same name as a function (such as `i`, `j`, `mode`, `char`, `size`, and `path`). In general, variable names take precedence over function names. If you create a variable that uses the name of a function, you sometimes get unexpected results.

Check whether a proposed name is already in use with the `exist` or `which` function. `exist` returns 0 if there are no existing variables, functions, or other artifacts with the proposed name. For example:

```
exist checkname
```

```
ans =  
    0
```


If you inadvertently create a variable with a name conflict, remove the variable from memory with the `clear` function.

Another potential source of name conflicts occurs when you define a function that calls `load` or `eval` (or similar functions) to add variables to the workspace. In some cases, `load` or `eval` add variables that have the same names as functions. Unless these variables are in the function workspace before the call to `load` or `eval`, the MATLAB parser interprets the variable names as function names. For more information, see:

- “Troubleshooting: Loading Variables within a Function”
- “Alternatives to the `eval` Function” on page 2-108

See Also

`clear` | `exist` | `iskeyword` | `namelengthmax` | `which`

Case and Space Sensitivity

MATLAB code is sensitive to casing, but insensitive to blank spaces.

Upper and Lowercase

In MATLAB code, use an exact match with regard to case for variables, files, and functions. For example, if you have a variable `a`, you cannot refer to that variable as `A`. It is a best practice to use lowercase only when naming functions. This is especially useful when you use both Microsoft® Windows® and UNIX¹ platforms because their file systems behave differently with regard to case.

When you use the `help` function, the help displays some function names in all uppercase, for example, `PLOT`, solely to distinguish a function name from the rest of the text. Some functions for interfacing to Sun Microsystems™ Java™ software do use mixed case and the command-line help and the documentation accurately reflect that.

Spaces

Blank spaces around operators such as `-`, `:`, and `()`, are optional, but they can improve readability. For example, MATLAB interprets the following statements the same way.

```
y = sin (3 * pi) / 2
y=sin(3*pi)/2
```

1. UNIX is a registered trademark of The Open Group in the United States and other countries.

Command vs. Function Syntax

In this section...

“Command and Function Syntaxes” on page 1-13

“Avoid Common Syntax Mistakes” on page 1-14

“How MATLAB Recognizes Command Syntax” on page 1-15

Command and Function Syntaxes

In MATLAB, these statements are equivalent:

```
load durer.mat           % Command syntax
load('durer.mat')       % Function syntax
```

This equivalence is sometimes referred to as *command-function duality*.

All functions support this standard *function syntax*:

```
[output1, ..., outputM] = functionName(input1, ..., inputN)
```

If you do not require any outputs from the function, and all of the inputs are literal strings (that is, text enclosed in single quotation marks), you can use this simpler *command syntax*:

```
functionName input1 ... inputN
```

With command syntax, you separate inputs with spaces rather than commas, and do not enclose input arguments in parentheses. Because all inputs are literal strings, single quotation marks are optional, unless the input string contains spaces. For example:

```
disp 'hello world'
```

When a function input is a variable, you must use function syntax to pass the value to the function. Command syntax always passes inputs as literal text and cannot pass variable values. For example, create a variable and call the `disp` function with function syntax to pass the value of the variable:

```
A = 123;
disp(A)
```

This code returns the expected result,

123

You cannot use command syntax to pass the value of A, because this call

`disp A`

is equivalent to

`disp('A')`

and returns

A

Avoid Common Syntax Mistakes

Suppose that your workspace contains these variables:

```
filename = 'accounts.txt';
A = int8(1:8);
B = A;
```

The following table illustrates common misapplications of command syntax.

This Command...	Is Equivalent to...	Correct Syntax for Passing Value
<code>open filename</code>	<code>open('filename')</code>	<code>open(filename)</code>
<code>isequal A B</code>	<code>isequal('A','B')</code>	<code>isequal(A,B)</code>
<code>strcmp class(A) int8</code>	<code>strcmp('class(A)','int8')</code>	<code>strcmp(class(A),'int8')</code>
<code>cd matlabroot</code>	<code>cd('matlabroot')</code>	<code>cd(matlabroot)</code>
<code>isnumeric 500</code>	<code>isnumeric('500')</code>	<code>isnumeric(500)</code>
<code>round 3.499</code>	<code>round('3.499')</code> , same as <code>round([51 46 52 57 57])</code>	<code>round(3.499)</code>

Passing Variable Names

Some functions expect literal strings for variable names, such as `save`, `load`, `clear`, and `whos`. For example,

```
whos -file durer.mat X
```

requests information about variable `X` in the demo file `durer.mat`. This command is equivalent to

```
whos('-file','durer.mat','X')
```

How MATLAB Recognizes Command Syntax

Consider the potentially ambiguous statement

```
ls ./d
```

This could be a call to the `ls` function with the folder `./d` as its argument. It also could request elementwise division on the array `ls`, using the variable `d` as the divisor.

If you issue such a statement at the command line, MATLAB can access the current workspace and path to determine whether `ls` and `d` are functions or variables. However, some components, such as the Code Analyzer and the Editor/Debugger, operate without reference to the path or workspace. In those cases, MATLAB uses syntactic rules to determine whether an expression is a function call using command syntax.

In general, when MATLAB recognizes an identifier (which might name a function or a variable), it analyzes the characters that follow the identifier to determine the type of expression, as follows:

- An equal sign (=) implies assignment. For example:

```
ls =d
```

- An open parenthesis after an identifier implies a function call. For example:

```
ls('./d')
```

- Space after an identifier, but not after a potential operator, implies a function call using command syntax. For example:

```
ls ./d
```

- Spaces on both sides of a potential operator, or no spaces on either side of the operator, imply an operation on variables. For example, these statements are equivalent:

```
ls ./ d
```

```
ls./d
```

Therefore, the potentially ambiguous statement `ls ./d` is a call to the `ls` function using command syntax.

The best practice is to avoid defining variable names that conflict with common functions, to prevent any ambiguity.

Program Components

- “Operators” on page 2-2
- “Special Values” on page 2-13
- “Conditional Statements” on page 2-15
- “Loop Control Statements” on page 2-17
- “Dates and Times” on page 2-19
- “Regular Expressions” on page 2-40
- “Comma-Separated Lists” on page 2-100
- “String Evaluation” on page 2-108
- “Shell Escape Functions” on page 2-113
- “Symbol Reference” on page 2-114

Operators

In this section...
“Arithmetic Operators” on page 2-2
“Relational Operators” on page 2-3
“Logical Operators” on page 2-4
“Operator Precedence” on page 2-11

Arithmetic Operators

Arithmetic operators perform numeric computations, for example, adding two numbers or raising the elements of an array to a given power. The following table provides a summary. For more information, see the arithmetic operators reference page.

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
\	Matrix left division
^	Matrix power

Arithmetic Operators and Arrays

Except for some matrix operators, MATLAB 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

Relational operators compare operands quantitatively, using operators like “less than” and “not equal to.” The following table provides a summary. For more information, see the relational operators reference page.

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

The MATLAB 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 logical 1. Locations where the relation is false receive logical 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. This behavior is consistent with that of all other binary operators, such as +, -, >, <, &, |, etc.

To test for empty arrays, use the function

```
isempty(A)
```

Logical Operators

MATLAB offers three types of logical operators and functions:

- Element-wise — operate on corresponding elements of logical arrays.

- Bit-wise — operate on corresponding bits of integer values or arrays.
- Short-circuit — operate on scalar, logical expressions.

The values returned by MATLAB logical operators and functions, with the exception of bit-wise functions, are of type `logical` and are suitable for use with logical indexing.

Element-Wise Operators and Functions

The following logical operators and functions perform elementwise logical operations on their inputs to produce a like-sized output array.

The examples shown in the following table use vector inputs A and B, where

```
A = [0 1 1 0 1];
B = [1 1 0 0 1];
```

Operator	Description	Example
&	Returns 1 for every element location that is true (nonzero) in both arrays, and 0 for all other elements.	A & B = 01001
	Returns 1 for every element location that is true (nonzero) in either one or the other, or both arrays, and 0 for all other elements.	A B = 11101
~	Complements each element of the input array, A.	~A = 10010
xor	Returns 1 for every element location that is true (nonzero) in only one array, and 0 for all other elements.	xor(A,B) = 10100

For operators and functions that take two array operands, (&, |, and xor), both arrays must have equal dimensions, with each dimension being the same size. The one exception to this is where one operand is a scalar and the other is not. In this case, MATLAB tests the scalar against every element of the other operand.

Note MATLAB converts any finite nonzero, numeric values used as inputs to logical expressions to logical 1, or `true`.

Operator Overloading. You can overload the `&`, `|`, and `~` operators to make their behavior dependent upon the class on which they are being used. Each of these operators has a representative function that is called whenever that operator is used. These are shown in the table below.

Logical Operation	Equivalent Function
<code>A & B</code>	<code>and(A, B)</code>
<code>A B</code>	<code>or(A, B)</code>
<code>~A</code>	<code>not(A)</code>

Other Array Functions. Two other MATLAB functions that operate logically on arrays, but not in an elementwise fashion, are `any` and `all`. These functions show whether *any* or *all* elements of a vector, or a vector within a matrix or an array, are nonzero.

When used on a matrix, `any` and `all` operate on the columns of the matrix. When used on an N-dimensional array, they operate on the first nonsingleton dimension of the array. Or, you can specify an additional dimension input to operate on a specific dimension of the array.

The examples shown in the following table use array input A, where

```
A = [0  1  2;  
     0 -3  8;  
     0  5  0];
```

Function	Description	Example
any(A)	Returns 1 for a vector where <i>any</i> element of the vector is true (nonzero), and 0 if no elements are true.	any(A) ans = 0 1 1
all(A)	Returns 1 for a vector where <i>all</i> elements of the vector are true (nonzero), and 0 if all elements are not true.	all(A) ans = 0 1 0

Note The all and any functions ignore any NaN values in the input arrays.

Short-Circuiting in Elementwise Operators. When used in the context of an if or while expression, and only in this context, the elementwise | and & operators use short-circuiting in evaluating their expressions. That is, A|B and A&B ignore the second operand, B, if the first operand, A, is sufficient to determine the result.

So, although the statement 1|[] evaluates to false, the same statement evaluates to true when used in either an if or while expression:

```
A = 1;   B = [];
if(A|B) disp 'The statement is true', end;
    The statement is true
```

while the reverse logical expression, which does not short-circuit, evaluates to false

```
if(B|A) disp 'The statement is true', end;
```

Another example of short-circuiting with elementwise operators shows that a logical expression such as the following, which under most circumstances is invalid due to a size mismatch between A and B,

```
A = [1 1];   B = [2 0 1];
A|B          % This generates an error.
```

works within the context of an if or while expression:

```
if (A|B) disp 'The statement is true', end;
```

The statement is true

Logical Expressions Using the find Function. The `find` function determines the indices of array elements that meet a given logical condition. The function is 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
```

Note An alternative to using `find` in this context is to index into the matrix using the logical expression itself. See the example below.

The last two statements of the previous example can be replaced with this one statement:

```
A(A > 8) = 100;
```

You can also use `find` to obtain both the row and column indices of a rectangular matrix for the array values that meet the logical condition:

```
A = magic(4)
A =
    16     2     3    13
```

```

5  11  10  8
9  7   6  12
4  14  15  1

[row, col] = find(A > 12)
row =
     1
     4
     4
     1
col =
     1
     2
     3
     4

```

Bit-Wise Functions

The following functions perform bit-wise logical operations on nonnegative integer inputs. Inputs may be scalar or in arrays. If in arrays, these functions produce a like-sized output array.

The examples shown in the following table use scalar inputs A and B, where

```

A = 28;           % binary 11100
B = 21;           % binary 10101

```

Function	Description	Example
bitand	Returns the bit-wise AND of two nonnegative integer arguments.	bitand(A,B) = 20 (binary 10100)
bitor	Returns the bit-wise OR of two nonnegative integer arguments.	bitor(A,B) = 29 (binary 11101)

Function	Description	Example
<code>bitcmp</code>	Returns the bit-wise complement as an n-bit number, where n is the second input argument to <code>bitcmp</code> .	<code>bitcmp(A,5) = 3</code> (binary 00011)
<code>bitxor</code>	Returns the bit-wise exclusive OR of two nonnegative integer arguments.	<code>bitxor(A,B) = 9</code> (binary 01001)

Short-Circuit Operators

The following operators perform AND and OR operations on logical expressions containing scalar values. They are *short-circuit* operators in that they evaluate their second operand only when the result is not fully determined by the first operand.

Operator	Description
<code>&&</code>	Returns logical 1 (<i>true</i>) if both inputs evaluate to <i>true</i> , and logical 0 (<i>false</i>) if they do not.
<code> </code>	Returns logical 1 (<i>true</i>) if either input, or both, evaluate to <i>true</i> , and logical 0 (<i>false</i>) if they do not.

The statement shown here performs an AND of two logical terms, A and B:

```
A && B
```

If A equals zero, then the entire expression will evaluate to logical 0 (*false*), regardless of the value of B. Under these circumstances, there is no need to evaluate B because the result is already known. In this case, MATLAB short-circuits the statement by evaluating only the first term.

A similar case is when you OR two terms and the first term is *true*. Again, regardless of the value of B, the statement will evaluate to *true*. There is no need to evaluate the second term, and MATLAB does not do so.

Advantage of Short-Circuiting. You can use the short-circuit operators to evaluate an expression only when certain conditions are satisfied. For example, you want to execute a function only if the function file resides on the current MATLAB path.

Short-circuiting keeps the following code from generating an error when the file, `myfun.m`, cannot be found:

```
comp = (exist('myfun.m') == 2) && (myfun(x) >= y)
```

Similarly, this statement avoids attempting to divide by zero:

```
x = (b ~= 0) && (a/b > 18.5)
```

You can also use the `&&` and `||` operators in `if` and `while` statements to take advantage of their short-circuiting behavior:

```
if (nargin >= 3) && (ischar(varargin{3}))
```

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 Element-wise AND (&)
- 9 Element-wise OR (|)
- 10 Short-circuit AND (&&)
- 11 Short-circuit OR (||)

Precedence of AND and OR Operators

MATLAB always gives the & operator precedence over the | operator. Although MATLAB typically evaluates expressions from left to right, the expression `a|b&c` is evaluated as `a|(b&c)`. It is a good idea to use parentheses to explicitly specify the intended precedence of statements containing combinations of & and |.

The same precedence rule holds true for the && and || operators.

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
```

Special Values

Several functions return important special values that you can use in your own program files.

Function	Return Value
<code>ans</code>	Most recent answer (variable). If you do not assign an output variable to an expression, MATLAB automatically stores the result in <code>ans</code> .
<code>eps</code>	Floating-point relative accuracy. This is the tolerance the MATLAB software uses in its calculations.
<code>intmax</code>	Largest 8-, 16-, 32-, or 64-bit integer your computer can represent.
<code>intmin</code>	Smallest 8-, 16-, 32-, or 64-bit integer your computer can represent.
<code>realmax</code>	Largest floating-point number your computer can represent.
<code>realmin</code>	Smallest positive floating-point number your computer can represent.
<code>pi</code>	3.1415926535897...
<code>i, j</code>	Imaginary unit.
<code>inf</code>	Infinity. Calculations like $n/0$, where n is any nonzero real value, result in <code>inf</code> .
<code>NaN</code>	Not a Number, an invalid numeric value. Expressions like $0/0$ and inf/inf result in a NaN, as do arithmetic operations involving a NaN. Also, if n is complex with a zero real part, then $n/0$ returns a value with a NaN real part.
<code>computer</code>	Computer type.
<code>version</code>	MATLAB version string.

Here are some examples that use these values in MATLAB expressions.

```
x = 2 * pi
x =
    6.2832
```

```
A = [3+2i 7-8i]
A =
    3.0000 + 2.0000i    7.0000 - 8.0000i
```

```
tol = 3 * eps
tol =
    6.6613e-016
```

```
intmax('uint64')
ans =
    18446744073709551615
```

Conditional Statements

Conditional statements enable you to select at run time which block of code to execute. The simplest conditional statement is an `if` statement. For example:

```
% Generate a random number
a = randi(100, 1);

% If it is even, divide by 2
if rem(a, 2) == 0
    disp('a is even')
    b = a/2;
end
```

`if` statements can include alternate choices, using the optional keywords `elseif` or `else`. For example:

```
a = randi(100, 1);

if a < 30
    disp('small')
elseif a < 80
    disp('medium')
else
    disp('large')
end
```

Alternatively, when you want to test for equality against a set of known values, use a `switch` statement. For example:

```
[dayNum, dayString] = weekday(date, 'long', 'en_US');

switch dayString
    case 'Monday'
        disp('Start of the work week')
    case 'Tuesday'
        disp('Day 2')
    case 'Wednesday'
        disp('Day 3')
    case 'Thursday'
        disp('Day 4')
```

```
        case 'Friday'
            disp('Last day of the work week')
        otherwise
            disp('Weekend!')
    end
```

For both `if` and `switch`, MATLAB executes the code corresponding to the first true condition, and then exits the code block. Each conditional statement requires the `end` keyword.

In general, when you have many possible discrete, known values, `switch` statements are easier to read than `if` statements. However, you cannot test for inequality between `switch` and `case` values. For example, you cannot implement this type of condition with a `switch`:

```
yourNumber = input('Enter a number: ');

if yourNumber < 0
    disp('Negative')
elseif yourNumber > 0
    disp('Positive')
else
    disp('Zero')
end
```

Loop Control Statements

With loop control statements, you can repeatedly execute a block of code. There are two types of loops:

- **for** statements loop a specific number of times, and keep track of each iteration with an incrementing index variable.

For example, preallocate a 10-element vector, and calculate five values:

```
x = ones(1,10);
for n = 2:6
    x(n) = 2 * x(n - 1);
end
```

- **while** statements loop as long as a condition remains true.

For example, find the first integer n for which `factorial(n)` is a 100-digit number:

```
n = 1;
nFactorial = 1;
while nFactorial < 1e100
    n = n + 1;
    nFactorial = nFactorial * n;
end
```

Each loop requires the `end` keyword.

It is a good idea to indent the loops for readability, especially when they are nested (that is, when one loop contains another loop):

```
A = zeros(5,100);
for m = 1:5
    for n = 1:100
        A(m, n) = 1/(m + n - 1);
    end
end
```

You can programmatically exit a loop using a `break` statement, or skip to the next iteration of a loop using a `continue` statement. For example, count

the number of lines in the help for the `magic` function (that is, all comment lines until a blank line):

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line)
        break
    elseif ~strncmp(line, '%',1)
        continue
    end
    count = count + 1;
end
fprintf('%d lines in MAGIC help\n',count);
fclose(fid);
```

Tip If you inadvertently create an infinite loop (a loop that never ends on its own), stop execution of the loop by pressing **Ctrl+C**.

Dates and Times

In this section...

- “Representing Dates and Times in MATLAB” on page 2-19
- “Date and Time Functions” on page 2-20
- “Working with Date Strings” on page 2-21
- “Date String Tables” on page 2-26
- “Working with Date Vectors” on page 2-29
- “Working with Serial Date Numbers” on page 2-33
- “Other Considerations” on page 2-36
- “Function Summary” on page 2-38

Representing Dates and Times in MATLAB

The MATLAB software represents date and time information in any of three formats:

- **Date String** — A character string for which you select which fields you want to include, and how you want these fields to appear in the string.

Example: `Wednesday, August 23, 2010 10:35:42.946 AM`
- **Date Vector** — A 1-by-6 numeric vector containing the year, month, day, hour, minute, and second.

Example: `[2010 5 25 9 45 44.9]`
- **Serial Date Number** — A single number equal to the number of days since a fixed, preset date (January 0, 0000).

Example: `7.3428e+005`

You have the choice of using any of these formats. If you work with more than one date and time format, MATLAB provides functions to help you easily convert from one format to another.

Dates and Dates with Time

You can work either with dates alone, or with dates and times combined. This table shows both options in the default date number, vector, and string formats.

	Date	Date and Time
Date Number	Days since Jan 1, 0000 n=734455 (n is whole)	Days since Jan 1, 0000 n = 734455.36 (n is real)
Date Vector	[year month day 0 0 0] [2010 11 13 0 0 0]	[year month day hour min sec] [2010 11 13 8 35 24]
Date String	day-month-year '13-Nov-2010'	day-month-year hour:min:sec '13-Nov-2010 08:35:24'

These formats also support *elapsed time*. See “Using Serial Date Numbers For Elapsed Time” on page 2-35 for more information.

Date and Time Functions

This table shows what information is available in MATLAB with respect to dates and times and which function provides this information. The sections that follow the table provide more information on how to use the different date and time formats.

Date and Time Information	Output Format	Function to Use
Current date and time	Date number	now
	Date vector	clock
	Date string	datestr(now)
Current date	Date number	datenum(date)
	Date vector	datevec(date)
	Date string	date
Day of week for given date	Full day name, abbreviated name, or day number in week (1-7)	weekday

Date and Time Information	Output Format	Function to Use
Last day of given month(s)	Vector of one or more days	eomday
Date with modified field	Date number	addtodate
Calendar for given month	6-by-7 matrix of days	calendar

For examples showing how to use these functions, see the function reference documentation.

Working with Date Strings

There are a number of ways to represent dates and times in character string format. For example, all of the following are date strings for August 23, 2010 at 04:35:42 PM:

```
'23-Aug-2010 04:35:06 PM'
'Wednesday, August 23'
'08/23/10 16:35'
'Aug 23 16:35:42.946'
```

Creating Date Strings In MATLAB

A date string is a character string composed of fields related to a specific date and/or time. In its default form, a date string has six fields containing values for a specific day, month, year, hour, minute, and second, in that order:

```
'23-Aug-2010 16:35:10'
```

A default date string that contains just a date consists of three fields: day, month, and year. The date function returns this type of date string:

```
date
ans =
    23-Aug-2010
```

To create a date string, you simply enter it as a MATLAB character string. Include any characters you might need to separate the fields, such as the hyphen, space, and colon used here:

```
d = '23-Aug-2010 16:35:42'
```

Minimum Requirements for Date Strings. A date string must contain at least a month and day field, or an hour and minute field. When entering month and day fields, MATLAB expects the month to precede the day. (There are ways to reverse this order, but that is more advanced usage and is documented under “How to Use the Field Specifier” on page 2-23.) The following date string is August 12, not December 8:

```
d = '08/12'
```

When entering the hour and minute fields, put the hour first and separate the two fields with a colon. The following date strings are 10:35 in the morning and 3:20 in the afternoon in both 24-hour and 12-hour notation:

```
t = '10:35'      t = '10:35 AM'  
t = '15:20'      t = '3:20 PM'
```

If the date includes the year, then the year value must immediately precede or immediately follow the month and day. The following are both valid date strings that represent the same day, August 12 in the year 2010:

```
d = '08/12/2010'  
d = '2010/08/12'
```

When unspecified, the year defaults to the current year, month and day default to January 1, hour and minute default to 00:00, and second and millisecond default to 00.000.

Specifying the Fields of a Date String

The MATLAB software provides the `datevec`, `datenum`, and `datestr` functions for converting from one date format to another. When you pass a date string to one of these functions, MATLAB does not necessarily know which fields have been included in the string, or the order in which they are positioned. Also, if you expect a date string to be returned by one of the conversion functions, you might need to indicate how that string is to be composed. For these reasons, unless you are passing a date string that uses the default field format, MathWorks recommends that you pass an additional argument called a *field specifier* in the call.

How to Use the Field Specifier. The reference page syntax for date conversion functions identifies the field specifier arguments as *FieldSpecIn* and *FieldSpecOut*. The *datevec* and *datenum* functions use the *FieldSpecIn* argument to indicate how MATLAB is to interpret the *input* date string:

```
DateVector = datevec(DateString, FieldSpecIn)
```

The *datestr* function uses the *FieldSpecOut* argument to indicate how MATLAB is to display the *output* date string:

```
DateString = datestr(DateVector, FieldSpecOut)
```

If you need to use a field specifier for both input and output strings, nest a call to *datenum* inside a call to *datestr* or *datevec*. This example changes date string 'August 11' to '11 August'. Because neither string is in the default format, you need a field specifier for both the input and output date strings:

```
datestr(datenum('August 11', 'mmm dd'), 'dd mmm')
ans =
    11 August
```

Note To convert a nonstandard date form into a MATLAB date form, first convert the nonstandard date form to a date number.

Character-Based Field Identifiers. There are two types of identifiers with which you can indicate the layout of fields in a date string. The more general of the two employs character-based symbols, such as those shown in this table, to designate the fields of a date string. (See “Symbolic Identifiers for Individual Fields” on page 2-26 for the full table).

Field	Example	Identifier
Year (in 4 digits)	2010	yyyy
Month (in 3 characters)	Aug	mmm
Day Number	23	dd
Hour	16	HH

Field	Example	Identifier
Minute	35	MM
Second	42	SS

The goal of the following example is to convert the date vector [2010 08 23 16 35 00] to a date string that has the nondefault format:

```
2010-08-23 16:35:42
```

Look up each of the six date fields in the table and use the symbols you find to compose a field specifier string that tells MATLAB how you want the date string output to look:

```
InputVector = [2010 08 23 16 35 42];

datestr(InputVector, 'yyyy-mm-dd HH:MM:SS')
ans =
    2010-08-23 16:35:42
```

Numeric Field Identifiers. There is an additional method for indicating the type of format you want applied to a date string. The MATLAB software associates 31 commonly used field sequences with a numeric identifier for each. Using these predefined formats can simplify the creation of your date strings. The table below shows several of these identifiers and the type of date string associated with them. (See “Numeric Identifiers for Predefined Formats” on page 2-28 for the full table).

Date String Formats	Example of Output	Identifier to Use
'dd-mmm-yyyy'	01-Mar-2000	1
'HH:MM:SS PM'	3:45:17 PM	14
'mm/dd/yyyy'	03/01/2000	23
'dd/mm/yyyy'	01/03/2000	24
'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17	31

Repeat the example from the previous section, this time using a numeric field specifier:

```
InputVector = [2010 08 23 16 35 42];
```

```
datestr(InputVector, 31)
ans =
    2010-08-23 16:35:42
```

Compare the commands used to achieve the same result. The advantage of the character-based field specifiers is that they are more versatile and easier to remember. The advantage of the numeric field specifiers is that frequently used commands are easier to enter:

```
datestr(InputVector, 'yyyy-mm-dd HH:MM:SS')    % Character
datestr(InputVector, 31)                       % Numeric
```

Adding Field Separation Characters. It is customary to separate certain fields of a date string with some form of punctuation, such as commas or space characters. Insert these characters into a date string by including them in the field specifier string. You cannot use any characters that could conflict with those symbolic characters reserved for use in the `FieldSpecIn` or `fieldSpecOut` strings (for example, `y`, `m`, `H`, `M`, and so on).

Add separation characters to make a date string easier to read:

```
datestr(now, 'dddd, mmmm dd, yyyy HH:MM')
ans =
    Wednesday, August 23, 2010 16:35
```

Creating Multiple Date Strings

Calling `datestr` with more than one date string input returns a character array of converted date strings. Pass the multiple date strings in a cell array. All input date strings must use the same format. For example, the following command passes three dates that all use the `mm/dd/yyyy` format:

```
datestr(datenum({'09/16/2007'; '05/14/1996'; '11/29/2010'}, ...
    'mm/dd/yyyy'))
ans =
    16-Sep-2007
    14-May-1996
    29-Nov-2010
```

Time Display in Date Strings

In MATLAB, you can represent time in a date string using either a 12-hour or 24-hour system in MATLAB. The following table shows how to create a 12-hour time string in the first column, and how to convert that time to its 24-hour equivalent.

12-hour time	Command to convert to 24-hour format	Equivalent 24-hour time
05:32 AM	<code>datestr('05:32 AM', 'HH:MM')</code>	5:32
05:32 PM	<code>datestr('05:32 PM', 'HH:MM')</code>	17:32

The next table shows how to create a 24-hour time string in the first column, and how to convert that time to its 12-hour equivalent.

24-hour time	Command to convert to 12-hour format	Equivalent 12-hour time
05:32	<code>datestr('05:32', 'HH:MM PM')</code>	5:32 AM
17:32	<code>datestr('17:32', 'HH:MM PM')</code>	5:32 PM

Warning The terms AM and PM, when used in the field specifier string, can be misleading. These terms do not influence which characters actually become part of the date string; they only determine whether or not to include them in the date string. MATLAB selects AM versus PM based on the time entered.

A few other things to remember when specifying time in MATLAB are:

- When you use AM or PM, the HH field is also required .
- When you do not use AM and PM, single-digit hours display a leading zero .

Date String Tables

Symbolic Identifiers for Individual Fields

The following table shows:

- The nine fields of a date string (left column)
- Ways to format each field
- Example output for each field
- Assigned character-based field identifiers

Field	String Format	Example of Output	Field Identifier
Quarter year	Letter Q and 1 digit	Q4	'QQ'
Year	4 digits	2007	'yyyy'
	2 digits	07	'yy'
Month	Full name	December	'mmmm'
	First 3 letters	Dec	'mmm'
	2 digits	12	'mm'
	First letter	D	'm'
Day	Full name	Tuesday	'dddd'
	First 3 letters	Tue	'ddd'
	2 digits	20	'dd'
	First letter	T	'd'
Hour	2 digits	16	'HH'
Minute	2 digits	02	'MM'
Second	2 digits	54	'SS'
Millisecond	Decimal point and 3 digits	.057	'FFF'
12-hour period	AM or PM	PM	'AM' or 'PM'

Notes Concerning Usage. Here are a few points to remember when using the symbolic identifiers:

MATLAB interprets the field specifiers in this table according to your computer's language setting and the current MATLAB language setting.

In a field specifier string, you cannot specify any field more than once. For example, you cannot use 'yy-mmm-dd-m' because it has two month identifiers. The one exception to this is that you can combine one instance of dd with one instance of any of the other day identifiers:

```
ds = datestr(now, 'dddd mmm dd yyyy')
ds =
    Wednesday Jun 30 2010
```

You only can use QQ (quarter of the year) alone or with a year specifier.

Numeric Identifiers for Predefined Formats

The following table shows numeric identifiers you can use to include certain field and format combinations in a date string.

Date String Formats	Example of Output	Identifier to Use
'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17	0
'dd-mmm-yyyy'	01-Mar-2000	1
'mm/dd/yy'	03/01/00	2
'mmm'	Mar	3
'm'	M	4
'mm'	03	5
'mm/dd'	03/01	6
'dd'	01	7
'ddd'	Wed	8
'd'	W	9
'yyyy'	2000	10
'yy'	00	11
'mmyy'	Mar00	12

Date String Formats	Example of Output	Identifier to Use
'HH:MM:SS'	15:45:17	13
'HH:MM:SS PM'	3:45:17 PM	14
'HH:MM'	15:45	15
'HH:MM PM'	3:45 PM	16
'QQ-YY'	Q1-01	17
'QQ'	Q1	18
'dd/mm'	01/03	19
'dd/mm/yy'	01/03/00	20
'mmm.dd,yyyy HH:MM:SS'	Mar.01,2000 15:45:17	21
'mmm.dd,yyyy'	Mar.01,2000	22
'mm/dd/yyyy'	03/01/2000	23
'dd/mm/yyyy'	01/03/2000	24
'yy/mm/dd'	00/03/01	25
'yyyy/mm/dd'	2000/03/01	26
'QQ-YYYY'	Q1-2001	27
'mmmyyyy'	Mar2000	28
'yyyy-mm-dd' (ISO 8601)	2000-03-01	29
'yyyymmddTHHMSS' (ISO 8601)	20000301T154517	30
'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17	31

Working with Date Vectors

Date vectors are an internal format for some MATLAB functions. You do not typically use date vectors in calculations, although you can use them to perform some simple computations such as the one shown in the example in this section.

A date vector is a 1-by-6 matrix of double-precision numbers arranged in the following order.

```
year month day hour minute second
```

The following date vector represents 10:45:07 AM on October 24, 2009.

```
[2009 10 24 10 45 07]
```

The fields of a date vector must follow these guidelines:

- All six elements, or *fields*, of the vector are required. If you are interested only in the date, and not the time, you can set the last three digits of the vector to zero.
- Time values are expressed in 24-hour notation. There is no AM or PM setting.
- The values for any of the six fields must be within an approximate range of 300 greater than or 550 less than the current value for that field.

Creating a Date Vector

As with any vector, you can create a date vector just as shown here. Be sure to put the fields in the correct order:

```
dv = [2010 8 23 16 35 42];
```

You can also create a date vector by converting a date string or serial date number. The `datevec` function converts from a date string or serial date number to a date vector.

Converting from Date String to Date Vector

The first argument to `datevec` can be a date string. If this string is in the default format for a date string, then you need only the one input argument:

```
dv = datevec('23-Aug 2010 16:35')
dv =
    2010         8        23        16        35         0
```

If the date string is in a nondefault date string format such as the one shown below, it is recommended that you also pass a field specifier argument. This argument tells MATLAB how the input string has been formatted:

```
dv = datevec('August 23, 2010 16:35', 'mmm dd, yyyy HH:MM')
dv =
    2010     8    23    16    35     0
```

The tables “Symbolic Identifiers for Individual Fields” on page 2-26 and “Numeric Identifiers for Predefined Formats” on page 2-28 list all of the format specifiers for date strings. When converting from a date string to a date vector, you can use any string from these tables.

Converting from Serial Date Number to Date Vector

To convert from a serial date number to a date vector, pass only the date number argument:

```
dv = datevec(7.343736909722223e+005)
dv =
    2010     8    23    16    35     0
```

This argument can be the output of a function that yields a serial date number such as the `now` function:

```
dv = datevec(now)
dv =
    1.0e+003 *
    2.0100    0.0060    0.0290    0.0100    0.0350    0.0460
```

Creating Multiple Date Vectors

Calling `datevec` with more than one date string input returns a character array of converted date strings. Pass the multiple date strings in a cell array. All input date strings must use the same format. For example, the following command passes three dates that all use the `mm/dd/yyyy` format:

```
datevec({'09/16/2007'; '05/14/1996'; '11/29/2010'})
ans =
    2007     9    16     0     0     0
    1996     5    14     0     0     0
```

```
2010    11    29    0    0    0
```

Milliseconds in Serial Date Numbers

Date vectors have no separate field in which to specify milliseconds. However, the `seconds` field has a fractional part and accurately keeps the milliseconds field. This example converts a date string with 647 milliseconds into a vector, and then converts it back into a string. Note that MATLAB fully restores the milliseconds count:

```
datenum([2010    8    23    16    35    0])
ans =
    7.3437e+005
dvec = datevec('11:21:02.647', 'HH:MM:SS.FFF')
dvec =
    1.0e+003 *
    2.0100    0.0010    0.0010    0.0110    0.0210    0.0026

datestr(dvec, 'HH:MM:SS.FFF')
ans =
    11:21:02.647
```

Examples of Using Date Vectors

Distribute the three time values in this vector to separate output variables:

```
[~, ~, ~, hour, min, sec] = datevec(now)
hour =
    11
min =
    47
sec =
    19.4660
```

Construction of the Eiffel Tower was begun on January 26 in 1887 and completed on March 31, 1889. This example finds the difference between the year, month, and day elements of the starting and ending date vectors to compute the time the tower was under construction:

```
startDate = [1887  1  26  0  0  0];    % January 26, 1887
finishDate = [1889  3  31  0  0  0];    % March 31, 1889
```

```
t = finishDate - startDate
t =
     2     2     5     0     0     0

[fprintf('\nThe Eiffel Tower took %d years, ', t(1)) ...
 fprintf('%d months, and %d days to construct.\n', t(1), t(2))];
```

The Eiffel Tower took 2 years, 2 months, and 2 days to construct.

Working with Serial Date Numbers

A serial date number 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 '31-Oct-2003, 6:00 PM' in MATLAB is date number 731885.75.

MATLAB works internally with serial date numbers. If you are using functions that handle large numbers of dates or doing extensive calculations with dates, you get better performance if you use date numbers.

You can create a serial date number from a date string or date vector using the following commands:

Converting from Date String to Serial Date Number

The first argument to `datenum` can be a date string. If this string is in the default format for a date string, then you need only the one input argument:

```
dv = datenum('23-Aug 2010 16:35')
dn =
    7.3437e+005
```

If the date string is in a nondefault date string format such as the one shown below, it is recommended that you also pass a field specifier argument. The field specifier tells MATLAB how the input string has been formatted:

```
dn = datenum('August 23, 2010 16:35', 'mmm dd, yyyy HH:MM')
```

```
dn =  
    7.3437e+005
```

The tables “Symbolic Identifiers for Individual Fields” on page 2-26 and “Numeric Identifiers for Predefined Formats” on page 2-28 list all of the format specifiers for date strings. When converting from a date string to a serial date number, you can use any string from these tables.

Certain formats may not contain enough information to compute a date number. In these cases, hours, minutes, seconds, and milliseconds default to 0, the month defaults to January, the day to 1, and the year to the current year.

Converting from Date Vector to Serial Date Number

To convert from a date vector to a serial date number, pass just the date vector argument:

```
datenum([2010 8 23 16 35 42])  
ans =  
    7.3437e+005
```

This argument can be the output of a function that yields a date vector such as the `clock` function:

```
dv = clock  
dv =  
    1.0e+003 *  
    2.0100    0.0060    0.0290    0.0110    0.0130    0.0057  
  
datenum(dv)  
ans =  
    7.3432e+005
```

Creating Multiple Serial Date Numbers

Calling `datenum` with more than one date string input returns a character array of converted date strings. Pass the multiple date strings in a cell array. All input date strings must use the same format. For example, the following command passes three dates that all use the `mm/dd/yyyy` format:


```

datenum({'09/16/2007'; '05/14/1996'; '11/29/2010'})
ans =
    733301
    729159
    734471

```

Using Serial Date Numbers For Elapsed Time

To find the time elapsed between two events, subtract the starting time from the ending time using serial date number format.

Warning Do not use date vectors to represent elapsed time. Also, be careful not to confuse the time of day format (7:30) with that of elapsed time (7 hours, 30 seconds).

This example computes the time elapsed between 8:15 AM and 3:45 PM on different days. Create two strings containing the starting and ending time:

```
s1 = '20-Apr 8:15';    s2 = '23-Apr 15:45';
```

Convert date strings to serial date numbers:

```

n1 = datenum(s1, 'dd-mmm HH:MM');
n2 = datenum(s2, 'dd-mmm HH:MM');

```

Subtract the starting time from the ending time:

```

n = n2 - n1
n =
    3.3125

```

Convert the answer to a string:

```

days = floor(n);
hrs = datestr(n, 'HH');
mins = datestr(n, 'MM');

fprintf('\n  %d days, %s hours, %s minutes\n', ...
        days, hrs, mins);

    3 days, 07 hours, 30 minutes

```

Examples of Using Serial Date Numbers

Add 50 days to the serial date number returned by the `now` function:

```
fprintf('\n In 50 days it will be %s.\n', ...
datestr(now+50, 'dddd, mmmm dd'))
```

```
In 50 days it will be Saturday, July 10.
```

Using the `addtodate` function, add 20 days to the date 20-Jan-2002. This date is first converted to a date number by a nested call to `datenum`:

```
addtodate(datenum('20.01.2002', 'dd.mm.yyyy'), 20, 'day')
R =
    731256
```

Other Considerations

- “Carrying to the Next Field” on page 2-36
- “Specifying a Pivot Year” on page 2-37
- “Date Vectors vs. Vectors of Date Numbers” on page 2-37

Carrying to the Next Field

Where reasonable, MATLAB automatically carries values outside the normal range of each unit to the next field. For example, specifying a date vector with a month value of 14 affects the output by setting the month to February and incrementing the year value by 1. The carrying forward of values applies only to date vectors and to time and day values in date strings, carries the 10 extra months from the input date string 22/03/2009 into October of the following year:

```
datestr([2009 22 03 00 00 00])
ans =
    03-Oct-2010
```

All units can wrap and have negative values. Note that specifying a negative day value, `D`, sets the output to the last day of the previous month minus

[D]. This example takes the input month (07, or July), finds the last day of the previous month (June 30), and subtracts the number of days in the field specifier (5 days) from that date to yield a return date of June 25, 2010:

```
datestr([2010 07 -05 00 00 00])
ans =
    25-Jun-2010
```

Specifying a Pivot Year

Use the pivot year to interpret date strings that specify the year using two characters. The pivot year is the starting year of the 100-year range in which a two-character date string year resides. The default pivot year is the current year minus 50 years.

The syntax for entering a pivot year is:

```
newString = datestr(oldString, fieldSpec, pivotYear);
```

This example changes the pivot year. Note the effect on the output:

```
datestr('4/16/55', 1, 1900)
ans =
    16-Apr-1955
```

```
datestr('4/16/55', 1, 2000)
ans =
    16-Apr-2055
```

Date Vectors vs. Vectors of Date Numbers

A six-element vector could represent either a single date vector, or a vector of six individual serial date numbers. For example, the vector [2010 12 15 11 45 03] could represent either 11:45:03 on December 15, 2010 or a vector of serial date numbers 2010, 12, 15, and so on.

If you are working with a date vector that the date conversion functions interpret by default as a vector of serial date numbers, you might need to explicitly convert your vector to a date number first, and then convert the value returned by `datenum` to the desired format.

In this example, the year 3000 is beyond the range of years for which MATLAB defaults to date vector format. Because of this, the intended format of the input vector is unclear to the `datestr` function, and the input is considered to be a vector of date numbers:

```
datestr([3000 11 05 10 32 56])
ans =
    18-Mar-0008
    11-Jan-0000
    05-Jan-0000
    10-Jan-0000
    01-Feb-0000
    25-Feb-0000
```

If it is your intention that the input is a date vector instead, convert it to a date number first, and then to a date string:

```
dn = datenum([3000 11 05 10 32 56]);
ds = datestr(dn)
ds =
    05-Nov-3000 10:32:56
```

Function Summary

MATLAB provides the following functions for time and date handling.

Current Date and Time Functions

Function	Description
<code>clock</code>	Return the current date and time as a date vector
<code>date</code>	Return the current date as date string
<code>now</code>	Return the current date and time as serial date number

Conversion Functions

Function	Description
<code>datenum</code>	Convert to a serial date number
<code>datestr</code>	Convert to a string representation of the date
<code>datevec</code>	Convert to a date vector

Utility Functions

Function	Description
<code>addtodate</code>	Modify a date number by field
<code>calendar</code>	Return a matrix representing a calendar
<code>datetick</code>	Label axis tick lines with dates
<code>eomday</code>	Return the last day of a year and month
<code>weekday</code>	Return the current day of the week

Timing Measurement Functions

Function	Description
<code>cputime</code>	Return the total CPU time used by MATLAB since it was started
<code>etime</code>	Return the time elapsed between two date vectors
<code>tic, toc</code>	Measure the time elapsed between invoking <code>tic</code> and <code>toc</code>

Regular Expressions

In this section...
“Overview” on page 2-40
“Calling Regular Expression Functions from MATLAB” on page 2-42
“Parsing Strings with Regular Expressions” on page 2-46
“Other Benefits of Using Regular Expressions” on page 2-50
“Metacharacters and Operators” on page 2-51
“Character Type Operators” on page 2-53
“Character Representation” on page 2-57
“Grouping Operators” on page 2-58
“Nonmatching Operators” on page 2-60
“Positional Operators” on page 2-61
“Lookaround Operators” on page 2-62
“Quantifiers” on page 2-68
“Tokens” on page 2-71
“Named Capture” on page 2-76
“Conditional Expressions” on page 2-78
“Dynamic Regular Expressions” on page 2-80
“String Replacement” on page 2-89
“Handling Multiple Strings” on page 2-91
“Function, Mode Options, Operator, Return Value Summaries” on page 2-91

Overview

A regular expression is a string of characters that defines a certain pattern. You normally use a regular expression to search text for a group of words that matches the pattern. For example, while parsing program input or while processing a block of text.

The string `'Joh?n\w*'` is an example of a regular expression. It defines a pattern that starts with the letters `Jo`, is optionally followed by the letter `h` (indicated by `'h?'`), is then followed by the letter `n`, and ends with any number of *word characters*² (indicated by `'\w*'`). This pattern matches any of the following:

Jon, John, Jonathan, Johnny

Regular expressions provide a unique way to search a volume of text for a particular subset of characters within that text. Instead of looking for an exact character match as you would do with a function like `strfind`, regular expressions give you the ability to look for a particular *pattern* of characters.

For example, several ways of expressing a metric rate of speed are:

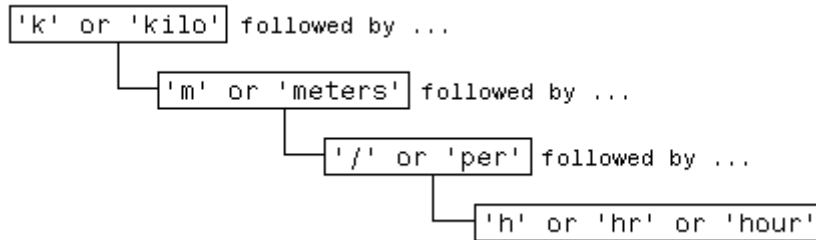
km/h
km/hr
km/hour
kilometers/hour
kilometers per hour

You could locate any of the above terms in your text by issuing five separate search commands:

```
strfind(text, 'km/h');  
strfind(text, 'km/hour');  
- etc. -
```

2. The term “word characters” in this text refers to characters that are alphabetic, numeric, or underscore.

To be more efficient, however, you can build a single phrase that applies to all of these search strings:



Translate this phrase into a regular expression (to be explained later in this section) and you have:

```
pattern = 'k(ilo)?m(eters)?(/|\sper\s)h(r|our)?';
```

Now locate one or more of the strings using just a single command:

```
text = ['The high-speed train traveled at 250 ', ...  
       'kilometers per hour alongside the automobile ', ...  
       'travelling at 120 km/h.'];  
  
regexp(text, pattern, 'match')  
ans =  
       'kilometers per hour'       'km/h'
```

Calling Regular Expression Functions from MATLAB

This section covers the following topics:

- “MATLAB Regular Expression Functions” on page 2-43
- “Returning the Desired Information” on page 2-43
- “Modifying Parameters of the Search (Modes)” on page 2-44

Note The examples in this and some of the later sections of this documentation use expressions that can be difficult to decipher for anyone not previously exposed to them. The purpose of these initial examples is to introduce the basic use of regular expressions in MATLAB. Learning how to translate the expressions begins in the “Metacharacters and Operators” on page 2-51 section.

MATLAB Regular Expression Functions

There are four MATLAB functions that support searching and replacing characters using regular expressions. The first three are similar in the input values they accept and the output values they return. For details, click the links in the table to see the corresponding function reference pages in the MATLAB Help.

Function	Description
regexp	Match regular expression.
regexpi	Match regular expression, ignoring case.
regexprep	Replace string using regular expression.
regexptranslate	Translate string into regular expression.

When calling any of the first three functions, pass the string to be parsed and the regular expression in the first two input arguments. When calling `regexprep`, pass an additional input that is an expression that specifies a pattern for the replacement string.

Returning the Desired Information

The `regexp` and `regexpi` functions return from 1 to 7 output values, providing the following information:

- The content or array indices of all matching strings
- The content of all nonmatching strings
- The content, names, or array indices of all tokens that were found

Unless you specify otherwise, MATLAB returns as many output values as you have output variables for. These are returned in the order shown by the “Return Value Summary” on page 2-99 table.

The following call to `regexp` returns all 7 outputs:

```
[matchStart, matchEnd, tokenIndices, matchStrings, ...  
    tokenStrings, tokenName, splitStrings] = regexp(str, expr);
```

To specify fewer values to return, include an identifying keyword in the input argument list when you call `regexp` or `regexp`. For example, the following statement uses two of these keywords, `match` and `start`:

```
[matchStrings, matchStart] = regexp(str, expr, 'match', 'start')
```

When you execute this statement, MATLAB assigns a cell array of all strings that match the pattern to variable `matchStrings`, and assigns an array of doubles containing the starting index of each match to variable `matchStart`.

For information on these output values and selecting which outputs to return, see the `regexp` function reference page.

Modifying Parameters of the Search (Modes)

You can fine-tune your regular expression parsing using the optional mode inputs: Case Sensitivity, Empty Match, Dot Matching, Anchor Type, and Spacing. These modes tell MATLAB whether or not to:

- Consider letter case when matching an expression to a string (Case Sensitivity mode).
- Allow successful matches of length zero (Empty Match mode).
- Include the newline (`\n`) character when matching the dot (`.`) metacharacter in a regular expression (Dot Matching mode).
- Consider the `^` and `$` metacharacters to represent the beginning and end of a string or the beginning and end of a line (Anchor Type mode).
- Ignore space characters and comments in the expression or to interpret them literally (Spacing mode).

See the reference page for the `regexp` function for more information on regular expression modes.

Applying Modes. You can apply any of these modes in either of two ways. (This is with the exception of Empty Match mode that applies only to all of the regular expression):

- Apply the mode to *all* of a regular expression by passing the mode specifier in the argument list of the call. See Example 1, below.
- Apply the mode to *specific parts* of your expression by specifying the mode symbolically within the regular expression itself. See Example 2, below.

Example 1 – Applying Case Sensitivity Mode to the Entire String.

Create two slightly different strings, `s1` and `s2`. Then write an expression `expr` that you can use to match both of these strings, but only when ignoring case. (The expression operators `.` and `+` match any consecutive series of any character between the `MAT` or `mat` phrases.)

```
s1 = 'Save your MATLAB data to a .mat file in C:\work\matlab';
s2 = 'Save your MATLAB data to a .MAT file in C:\work\matlab';
expr = '.+MAT.+mat.+mat.+';
```

Run `regexp` on both strings at the same time in `ignorecase` mode and examine the output in cell array `c`:

```
c = regexp({s1, s2}, expr, 'match', 'ignorecase');
c{:}
ans =
    'Save your MATLAB data to a .mat file in C:\work\matlab'
ans =
    'Save your MATLAB data to a .MAT file in C:\work\matlab'
```

Because of the `ignorecase` mode, there is a match for both strings. When you use `matchcase` mode instead, only the exact case match is accepted:

```
c = regexp({s1, s2}, expr, 'match', 'matchcase');
c{:}
ans =
    'Save your MATLAB data to a .mat file in C:\work\matlab'
```

```
ans =  
    {}
```

Example 2 – Applying Case Sensitivity Mode Selectively. This example uses symbolic mode designators within the expression itself. The (?i) symbol tells `regexp` to ignore case for that part of the expression that immediately follows it. Similarly, the (?-i) symbol requires case to match for the part of the expression following it.

Here are three strings that vary slightly in case. Following that is the expression `expr` that employs the two states of the Case Sensitivity mode. Note that each of the (?-i) or (?i) symbols used in this expression applies only to the letters `MAT` or `mat` that immediately follow it:

```
s1 = 'Save your MATLAB data to a .mat file in C:\work\matlab';  
s2 = 'Save your MATLAB data to a .MAT file in C:\work\MATLAB';  
s3 = 'Save your MATLAB data to a .MAT file in C:\work\matlab';  
expr = '.*(?-i)MAT.*(?i)mat.*(?-i)mat';
```

Run `regexp` on the three strings. According to the expression `expr`, the first and third instances of the letters 'mat' must be in upper and lower case, respectively. Case is ignored for the second instance. Only strings `s1` and `s3` satisfy this condition:

```
c = regexp({s1,s2,s3}, expr, 'match');  
c{:}  
ans =  
    'Save your MATLAB data to a .mat file in C:\work\mat'  
ans =  
    {}  
ans =  
    'Save your MATLAB data to a .MAT file in C:\work\mat'
```

Parsing Strings with Regular Expressions

MATLAB parses a string from left to right, “consuming” the string as it goes. If matching characters are found, `regexp` records the location and resumes parsing the string, starting just after the end of the most recent match. There is no overlapping of characters in this process. See Examples 2a and 2b under “Using the Lookahead Operator” if you need to match overlapping character groups.

There are three steps involved in using regular expressions to search text for a particular string:

1 Identify unique patterns in the string

This entails breaking up the string you want to search for into groups of like character types. These character types could be a series of lowercase letters, a dollar sign followed by three numbers and then a decimal point, etc.

2 Express each pattern as a regular expression

Use the *metacharacters* and operators described in this documentation to express each segment of your search string as a regular expression. Then combine these expression segments into the single expression to use in the search.

3 Call the appropriate search function

Pass the string you want to parse to one of the search functions, such as `regexp` or `regexp_i`, or to the string replacement function, `regxprep`.

The example shown in this section searches a record containing contact information belonging to a group of five friends. This information includes each person's name, telephone number, place of residence, and email address. The goal is to extract specific information from one or more of the strings.

```
contacts = { ...
'Harry 287-625-7315 Columbus, OH hparker@hmail.com'; ...
'Janice 529-882-1759 Fresno, CA jan_stephens@horizon.net'; ...
'Mike 793-136-0975 Richmond, VA sue_and_mike@hmail.net'; ...
'Nadine 648-427-9947 Tampa, FL nadine_berry@horizon.net'; ...
'Jason 697-336-7728 Montrose, CO jason_blake@mymail.com'};
```

The first part of the example builds a regular expression that represents the format of a standard email address. Using that expression, the example then searches the information for the email address of one of the group of friends. Contact information for Janice is in row 2 of the `contacts` cell array:

```
contacts{2}
ans =
    Janice 793-882-1759 Fresno, CA jan_stephens@horizon.net
```

Step 1 – Identify Unique Patterns in the String

A typical email address is made up of standard components: the user's account name, followed by an @ sign, the name of the user's internet service provider (ISP), a dot (period), and the domain to which the ISP belongs. The table below lists these components in the left column, and generalizes the format of each component in the right column.

Unique patterns of an email address	General description of each pattern
Start with the account name jan_stephens . . .	One or more lowercase letters and underscores
Add '@' jan_stephens@ . . .	@ sign
Add the ISP jan_stephens@horizon . . .	One or more lowercase letters, no underscores
Add a dot (period) jan_stephens@horizon. . . .	Dot (period) character
Finish with the domain jan_stephens@horizon.net	com or net

Step 2 – Express Each Pattern as a Regular Expression

In this step, you translate the general formats derived in Step 1 into segments of a regular expression. You then add these segments together to form the entire expression.

The table below shows the generalized format descriptions of each character pattern in the left-most column. (This was carried forward from the right column of the table in Step 1.) The second column links to tables in this documentation that show the appropriate expressions to use in translating this description into a regular expression. The third column shows the operators or metacharacters chosen from those tables to represent the character pattern.

Description of each segment	Tables referenced	Related metacharacters
One or more lowercase letters and underscores	See Character Types on page 2-93, Quantifiers on page 2-96.	[a-z_]+
@ sign	See Character Representation on page 2-94.	@
One or more lowercase letters, no underscores	See Character Types on page 2-93, Quantifiers on page 2-96.	[a-z]+
Dot (period) character	See Character Representation on page 2-94.	\.
com or net	See Grouping Operators on page 2-94.	(com net)

Assembling these metacharacters into one string gives you the complete expression:

```
email = '[a-z_]+@[a-z]+\.(com|net)';
```

Step 3 – Call the Appropriate Search Function

In this step, you use the regular expression derived in Step 2 to match an email address for one of the friends in the group. Use the `regexp` function to perform the search.

Here is the list of contact information shown earlier in this section. Each person's record occupies a row of the `contacts` cell array:

```
contacts = { ...
'Harry 287-625-7315 Columbus, OH hparker@hmail.com'; ...
'Janice 529-882-1759 Fresno, CA jan_stephens@horizon.net'; ...
'Mike 793-136-0975 Richmond, VA sue_and_mike@hmail.net'; ...
'Nadine 648-427-9947 Tampa, FL nadine_berry@horizon.net'; ...
'Jason 697-336-7728 Montrose, CO jason_blake@mymail.com'};
```

This is the regular expression that represents an email address, as derived in Step 2:

```
email = '[a-z_]+@[a-z]+\.(com|net)';
```

Call the `regexp` function, passing row 2 of the `contacts` cell array and the `email` regular expression. This returns the email address for Janice.

```
regexp(contacts{2}, email, 'match')
ans =
    'jan_stephens@horizon.net'
```

Note The last input passed to `regexp` in this command is the keyword `'match'`. This keyword causes `regexp` to return the output as a string instead of as indices into the cell array.

Make the same call, but this time for the fifth person in the list:

```
regexp(contacts{5}, email, 'match')
ans =
    'jason_blake@mymail.com'
```

You can also search for the email address of everyone in the list by using the entire cell array for the input string argument:

```
regexp(contacts, email, 'match');
```

Other Benefits of Using Regular Expressions

In addition to parsing single strings, you can also use the MATLAB regular expression functions for any of the following tasks:

- “Parsing or Replacing with Multiple Expressions and Strings” on page 2-50
- “Replacing Parts of a String” on page 2-51
- “Matching with Tokens Taken from the String” on page 2-51
- “Matching and Replacing Strings Dynamically” on page 2-51

Parsing or Replacing with Multiple Expressions and Strings

The MATLAB regular expression functions also work on multiple strings contained in a cell array. You can use multiple strings as the strings to be parsed, as regular expressions to match against the parse string(s), as replacement strings, or most combinations of these.

Replacing Parts of a String

String replacement with regular expressions requires the `regexprep` function. This function accepts two regular expressions in its input argument list. Each expression specifies a character pattern to match in the string to be parsed. The function then replaces occurrences of the first pattern with occurrences of the second.

Matching with Tokens Taken from the String

A token is one or more characters selected from within the string being parsed that you can use to match other characters in the same string. The characters representing a token are not constants; they depend upon the contents of the parse string that match a part of the expression. You define a token by enclosing part of a regular expression in parentheses. You search for that token using the metacharacters `\1`, `\2`, etc. You can also use tokens in specifying a replacement string for the `regexprep` function. In this case, you refer to specific tokens using the metacharacters `$1`, `$2`, etc.

Matching and Replacing Strings Dynamically

With dynamic expressions, you can:

- Execute a MATLAB command within your expression parsing command.
- Execute a MATLAB command, and include the returned string in the match expression.
- Parse a regular expression and include the resulting string in the match expression.

Metacharacters and Operators

Much of the remainder of this section on regular expressions documents the various metacharacters and operators that you need to compose your expressions.

Category	Metacharacters and Operators
“Character Type Operators” on page 2-53	One of a certain group of characters (e.g., a character in a predefined set or range, a whitespace character, an alphabetic, numeric, or underscore character, or a character that is not in one of these groups).
“Character Representation” on page 2-57	Metacharacters that represent a special character (e.g., backslash, new line, tab, hexadecimal values, any untranslated literal character, etc).
“Grouping Operators” on page 2-58	A grouping of letters or metacharacters to apply a regular expression operator to.
“Nonmatching Operators” on page 2-60	Text included in an expression for the purpose of adding a comment statement, but not to be used as a pattern to find a match for.
“Positional Operators” on page 2-61	Location in the string where the characters or pattern must be positioned for there to be a match (e.g., start or end of the string, start or end of a word, an entire word).
“Lookaround Operators” on page 2-62	Characters or patterns that immediately precede or follow the intended match, but are not considered to be part of the match itself.
“Quantifiers” on page 2-68	Various ways of expressing the number of times a character or pattern is to occur for there to be a match (e.g., exact number, minimum, maximum, zero or one, zero or more, one or more, etc.)
“Tokens” on page 2-71	Characters or patterns selected from the string being parsed that you can use to match other characters in the string.
“Named Capture” on page 2-76	Operators used in assigning names to matched tokens, thus making your code more maintainable and the output easier to interpret.
“Conditional Expressions” on page 2-78	Operators that express conditions under which a certain match is considered to be acceptable.

Category	Metacharacters and Operators
“Dynamic Regular Expressions” on page 2-80	Operators that include a subexpression or command that MATLAB parses or executes. MATLAB uses the result of that operation in parsing the overall expression.
“String Replacement” on page 2-89	Operators used with the <code>regexprep</code> function to specify the content of the replacement text.

Character Type Operators

Tables and examples in this and subsequent sections show the operators and syntax supported by the MATLAB `regexp`, `regexpi`, and `regexprep` functions. Expressions shown in the left column have special meaning and match one or more characters according to the usage described in the right column. Any character not having a special meaning, for example, any alphabetic character, matches that same character literally. To force one of the regular expression functions to interpret a sequence of characters literally (rather than as an operator) use the `regextranslate` function.

Character types represent either a specific set of characters (e.g., uppercase) or a certain type of character (e.g., nonwhitespace).

Operator	Usage
.	Any single character, including white space
[<i>c</i> ₁ <i>c</i> ₂ <i>c</i> ₃]	Any character contained within the brackets: <i>c</i> ₁ or <i>c</i> ₂ or <i>c</i> ₃
[[^] <i>c</i> ₁ <i>c</i> ₂ <i>c</i> ₃]	Any character not contained within the brackets: anything but <i>c</i> ₁ or <i>c</i> ₂ or <i>c</i> ₃
[<i>c</i> ₁ - <i>c</i> ₂]	Any character in the range of <i>c</i> ₁ through <i>c</i> ₂
\s	Any white-space character; equivalent to [<code>\f\n\r\t\v</code>]
\S	Any nonwhitespace character; equivalent to [[^] <code>\f\n\r\t\v</code>]

Operator	Usage
<code>\w</code>	Any alphabetic, numeric, or underscore character. For English character sets, this is equivalent to <code>[a-zA-Z_0-9]</code> .
<code>\W</code>	Any character that is not alphabetic, numeric, or underscore. For English character sets, this is equivalent to <code>[^a-zA-Z_0-9]</code> .
<code>\d</code>	Any numeric digit; equivalent to <code>[0-9]</code>
<code>\D</code>	Any nondigit character; equivalent to <code>[^0-9]</code>

The following examples demonstrate how to use the character classes listed above. See the `regexp` reference page for help with syntax. Most of these examples use the following string:

```
str = 'The rain in Spain falls mainly on the plain.';
```

Any Character — .

The `.` operator matches any single character, including whitespace.

Example 1 — Matching Any Character. Use the dot (`.`) operator to locate sequences of five consecutive characters that end with `'ain'`. The regular expression used in this example is

```
expr = '..ain';
```

Find each occurrence of the expression `expr` within the input string `str`. Return a vector of the indices at which any matches begin:

```
str = 'The rain in Spain falls mainly on the plain.';
```

```
startIndex = regexp(str, expr)
startIndex =
     4     13     24     39
```

Here is the input string with the returned `startIndex` values shown below it. Note that the dot operator not only matches the letters in the string, but whitespace characters as well:

```
The rain in Spain falls mainly on the plain.
    |         |         |         |
    4         13        24         39
```

If you would prefer to have MATLAB return the text of the matching substrings, use the 'match' qualifier in the command:

```
matchStr = regexp(str, expr, 'match')
matchStr =
    'rain'    'Spain'    'main'    'plain'
```

Example 2 – Returning Strings Rather than Indices. Here is the same example, this time specifying the command qualifier 'match'. In this case, `regexp` returns the *text* of the matching strings rather than the starting index:

```
regexp(str, '..ain', 'match')
ans =
    'rain'    'Spain'    'main'    'plain'
```

Selected Characters – [c₁c₂c₃]

Use [c₁c₂c₃] in an expression to match selected characters r, p, or m followed by 'ain'. Specify two qualifiers this time, 'match' and 'start', along with an output argument for each, `mat` and `idx`. This returns the matching strings and the starting indices of those strings:

```
[mat idx] = regexp(str, '[rpm]ain', 'match', 'start')
mat =
    'rain'    'pain'    'main'
idx =
    5     14     25
```

Range of Characters — [c1 - c2]

Use [c₁-c₂] in an expression to find words that begin with a letter in the range of A through Z:

```
[mat idx] = regexp(str, '[A-Z]\w*', 'match', 'start')
mat =
    'The'      'Spain'
idx =
     1      13
```

Word and White-Space Characters — \w, \s

Use \w and \s in an expression to find words that end with the letter n followed by a white-space character. Add a new qualifier, 'end', to return the str index that marks the end of each match:

```
[mat ix1 ix2] = regexp(str, '\w*n\s', 'match', 'start', 'end')
mat =
    'rain '    'in '    'Spain '    'on '
ix1 =
     5     10     13     32
ix2 =
     9     12     18     34
```

Numeric Digits — \d

Use \d to find numeric digits in the following string:

```
numstr = 'Easy as 1, 2, 3';

[mat idx] = regexp(numstr, '\d', 'match', 'start')
mat =
    '1'    '2'    '3'
idx =
     9     12     15
```

Character Representation

The following character combinations represent specific character and numeric values.

Operator	Usage
\a	Alarm (beep)
\\	Backslash
\\$	Dollar sign
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\oN or \o{N}	Character of octal value N
\xN or \x{N}	Character of hexadecimal value N
\char	If a character has special meaning in a regular expression, precede it with backslash (\) to match it literally.

Octal and Hexadecimal – \o, \x

Use \x and \o in an expression to find a comma (hex 2C) followed by a space (octal 40) followed by the character 2:

```
numstr = 'Easy as 1, 2, 3';
```

```
[mat idx] = regexp(numstr, '\x2C{o{40}2', 'match', 'start')
mat =
    ', 2'
idx =
    10
```

Grouping Operators

When you need to use one of the regular expression operators on a number of consecutive elements in an expression, group these elements together with one of the grouping operators and apply the operation to the entire group. For example, this command matches a capital letter followed by a numeral and then an optional space character. These elements have to occur at least two times in succession for there to be a match. To apply the {2,} multiplier to all three consecutive characters, you can first make a group of the characters and then apply the (?:) quantifier to this group:

```
regexp('B5 A2 6F 63 R6 P4 B2 BC', '(?:[A-Z]\d\s){2,}', 'match')
ans =
    'B5 A2 '      'R6 P4 B2 '
```

There are three types of explicit grouping operators that you can use when you need to apply an operation to more than just one element in an expression. Also in the grouping category is the alternative match (logical OR) operator, |. This creates two or more groups of elements in the expression and applies an operation to one of the groups.

Operator	Usage
(expr)	Group regular expressions and capture tokens.
(?:expr)	Group regular expressions, but do not capture tokens.
(?>expr)	Group atomically.
expr ₁ expr ₂	Match expression expr ₁ or expression expr ₂ .

Grouping and Capture – (expr)

When you enclose an expression in parentheses, MATLAB not only treats all of the enclosed elements as a group, but also captures a token from these elements whenever a match with the input string is found. For an example of how to use this, see “Using Tokens — Example 1” on page 2-73.

Grouping Only – (?:expr)

Use (?:expr) to group a non-vowel (consonant, numeric, whitespace, punctuation, etc.) followed by a vowel in the palindrome pstr. Specify at least

two consecutive occurrences (`{2,}`) of this group. Return the starting and ending indices of the matched substrings:

```
pstr = 'Marge lets Norah see Sharon''s telegram';
expr = '(?:[aeiou][aeiou]){2,}';

[mat ix1 ix2] = regexp(pstr, expr, 'match', 'start', 'end')
mat =
    'Nora'    'haro'    'tele'
ix1 =
    12     23     31
ix2 =
    15     26     34
```

Remove the grouping, and the `{2,}` now applies only to `[aeiou]`. The command is entirely different now as it looks for a non-vowel followed by at least two consecutive vowels:

```
expr = '[^aeiou][aeiou]{2,}';

[mat ix1 ix2] = regexp(pstr, expr, 'match', 'start', 'end')
mat =
    'see'
ix1 =
    18
ix2 =
    20
```

Alternative Match – `expr1|expr2`

Use `p1|p2` to pick out words in the string that start with `let` or `tel`:

```
regexpi(pstr, '(let|tel)\w+', 'match')
ans =
    'lets'    'telegram'
```

Note The expressions `A | B` and `B | A` may return different answers. If there is a match with the first part of the expression (before the `|` symbol), then the second part (that follows the `|` symbol) is not considered.

See the following example. Both calls to `regexp` parse the same string, and, except for the order of the OR conditions, the same expression. But the first call returns two values and the second returns just one:

```
string = 'one two';    expr1 = '(\w+\s\w+)';    expr2 = '(\w+)';

regexp(string, [expr1 '|' expr2], 'match')
ans =
    'one two'

regexp(string, [expr2 '|' expr1], 'match')
ans =
    'one'    'two'
```

Nonmatching Operators

The comment operator enables you to insert comments into your code to make it more maintainable. The text of the comment is ignored by MATLAB when matching against the input string.

Operator	Usage
(?#comment)	Insert a comment into the expression. Comments are ignored in matching.

Including Comments — (?#expr)

Use `(?#expr)` to add a comment to this expression that matches capitalized words in `pstr`. Comments are ignored in the process of finding a match:

```
regexp(pstr, '(?# Match words in caps)[A-Z]\w+', 'match')
ans =
    'Marge'    'Norah'    'Sharon'
```

Positional Operators

Positional operators in an expression match parts of the input string not by content, but by where they occur in the string (e.g., the first N characters in the string).

Operator	Usage
<code>^expr</code>	Match <code>expr</code> if it occurs at the beginning of the input string.
<code>expr\$</code>	Match <code>expr</code> if it occurs at the end of the input string.
<code>\<expr</code>	Match <code>expr</code> when it occurs at the beginning of a word.
<code>expr\></code>	Match <code>expr</code> when it occurs at the end of a word.
<code>\<expr\></code>	Match <code>expr</code> when it represents the entire word.

Start and End of String Match — `^expr`, `expr$`

Use `^expr` to match words starting with the letter `m` or `M` only when it begins the string, and `expr$` to match words ending with `m` or `M` only when it ends the string:

```
regexpi(pstr, '^m\w*|\w*m$', 'match')
ans =
    'Marge'      'telegram'
```

Start and End of Word Match — `\<expr`, `expr\>`

Use `\<expr` to match any words starting with `n` or `N`, or ending with `e` or `E`:

```
regexpi(pstr, '\<n\w*|\w*e\>', 'match')
ans =
    'Marge'      'Norah'      'see'
```

Exact Word Match — `\<expr\>`

Use `\<expr\>` to match a word starting with an `n` or `N` and ending with an `h` or `H`:

```
regexpi(pstr, '\<n\w*h\>', 'match')
ans =
    'Norah'
```

Lookaround Operators

Lookaround operators tell MATLAB to look either ahead or behind the current location in the string for a specified expression. If the expression is found, MATLAB attempts to match a given pattern.

This table shows the four lookaround expressions: lookahead, negative lookahead, lookbehind, and negative lookbehind.

Operator	Usage
(?=expr)	Look ahead from current position and test if expr is found.
(?!expr)	Look ahead from current position and test if expr is not found
(?<=expr)	Look behind from current position and test if expr is found.
(?<!expr)	Look behind from current position and test if expr is not found.

Lookaround operators do not change the current parsing location in the input string. They are more of a condition that must be satisfied for a match to occur.

For example, the following command uses an expression that matches alphabetic, numeric, or underscore characters (`\w*`) that meet the condition that they *look ahead to* (i.e., are immediately followed by) the letters `vision`. The resulting match includes only that part of the string that matches the `\w*` operator; it does not include those characters that match the lookahead expression (`?=vision`):

```
[s e] = regexp('telegraph television telephone', ...
               '\w*(?=vision)', 'start', 'end')
s =
    11
e =
```

14

If you repeat this command and match one character beyond the lookahead expression, you can see that parsing of the input string resumes at the letter v, thus demonstrating that matching the lookahead operator has not consumed any characters in the string:

```
regexp('telegraph television telephone', ...
       '\w*(?=vision).', 'match')
ans =
    'telev'
```

Note You can also use lookaround operators to perform a logical AND of two elements. See “Using Lookaround as a Logical Operator” on page 2-67.

Using the Lookahead Operator – `expr(?=test)`

Example 1 – Lookahead. Look ahead to a file name (`fileread.m`), and return only the name of the folder in which it resides, not the file name itself. Note that the lookahead part of the expression serves only as a condition for the match; it is not part of the match itself:

```
str = which('fileread')
str =
    C:\Program Files\MATLAB\toolbox\matlab\iofun\fileread.m

% Look ahead to a backslash (\\), followed by a file name (\w+)
% with an .m or .p extension (\.[mp]). Capture the letters
% that precede this sequence.
regexp(str, '\w+(?=\w+\.[mp])', 'match')
ans =
    'iofun'
```

Example 2a – Matching Sequential Character Groups. MATLAB parses a string from left to right, “consuming” the string as it goes. If matching characters are found, `regexp` records the location and resumes parsing the string from the location of the most recent match. There is no overlapping of characters in this process.

Find all sequences of 6 nonwhitespace characters in the input string shown below. Following the MATLAB default behavior, do not allow for overlap. That is, begin looking for your next match starting just after the *end* of the current match:

```
string = 'Locate several 6-char. phrases';
regexpi(string, '\S{6}')
ans =
     1     8    16    24
```

This statement finds the phrases:

```
Locate  severa  6-char  phrase
```

Example 2b – Using Lookahead to Match Overlapping Character

Groups. If you need to find *every* sequence of characters that match a pattern, including sequences that overlap another, capture only the first character and look ahead for the remainder of the pattern. In other words, begin looking for your next match starting after the *next character* of the current match:

```
string = 'Locate several 6-char. phrases';
regexpi(string, '\S(?=\S{5})')
ans =
     1     8     9    16    17    24    25
```

This statement finds the phrases:

```
Locate  severa  everal  6-char  -char.  phrase  hrases
```

Using the Negative Lookahead Operator – `expr(?!test)`

Example – Negative Lookbehind and Lookahead. Generate a series of sequential numbers:

```
n = num2str(5:15)
n =
     5     6     7     8     9    10    11    12    13    14    15
```

Use both the negative lookbehind and negative lookahead operators together to precede only the single-digit numbers with zero:

```

regexprep(n, '(?<!\d)(\d)(?!\d)', '0$1')
ans =
    05    06    07    08    09    10    11    12    13    14    15

```

Using the Lookbehind Operator – (?<=test)expr

Example 1 – Positive and Negative Lookbehind Operators. Using the lookbehind operator, find the letter r that is preceded by the letter u:

```

str = 'Neural Network Toolbox';

startIndex = regexp(str, '(?<=u)r', 'start')
startIndex =
     4

```

Using the negative lookbehind operator, find the letter r that is *not* preceded by the letter u:

```

startIndex = regexp(str, '(?!u)r', 'start')
startIndex =
    13

```

Example 2 – Lookbehind. Return the names and 7-digit telephone numbers for those people in the list that are in the 617 area code. The lookbehind (?<=^617-) finds those lines that begin with the number 617:

```

phone_list = {...
'978-389-2457 Kevin';      '617-922-3091 Ruth'; ...
'781-147-1748 Alan';      '508-643-9648 George'; ...
'617-774-6642 Lisa';      '617-241-0275 Greg'; ...
'413-995-9114 Jason';      '781-276-0482 Victoria'};
len = length(phone_list);

ph617 = regexp(phone_list, '(?<=^617-).*', 'match');

for k=1:len
str = char(ph617{k});
if ~isempty(str), fprintf(' %s\n', str), end
end

```

MATLAB returns the three numbers that have a 617 area code:

```
922-3091 Ruth
774-6642 Lisa
241-0275 Greg
```

Using the Negative Lookbehind Operator— (?<!test)expr

Example — Negative Lookbehind. This example uses negative lookbehind to find those tasks that are not labelled as Done or Pending, Create a list of tasks, each with status information to the left:

```
tasks = {...
'ToDo      3892457';      'Done      9223091'; ...
'Pending   1471748';      'Maybe    7746642'; ...
'ToDo      2410275';      'Pending   4723596'; ...
'ToDo      9959114';      'Maybe    2760482'; ...
'ToDo      3080027';      'Done      1221941'};
count = length(tasks);
```

The regular expression looks for those task numbers that do not have a Done or Pending status. Note that you can use the or (|) operator in a lookaround to check for more than one condition:

```
doNow = regexp(tasks, '(?<!^(Done|Pending).*)\d+', 'match');
```

Now print out the results:

```
disp 'The following tasks need attention:'
for k=1:count
    s = char(doNow{k});
    if ~isempty(s), fprintf('  %s\n', s), end
end
```

The output displays all but the Done and Pending tasks:

```
The following tasks need attention:
3892457
7746642
```



```
2410275
9959114
2760482
3080027
```

Using Lookaround as a Logical Operator

One way in which a lookahead operation can be useful is to perform a logical AND between two conditions. This example initially attempts to locate all lowercase consonants in a text string. The text string is the first 50 characters of the help for the `normest` function:

```
helptext = help('normest');
str = helptext(1:50)
str =
  NORMEST Estimate the matrix 2-norm.
  NORMEST(S
```

Merely searching for non-vowels (`[^aeiouAEIOU]`) does not return the expected answer, as the output includes capital letters, space characters, and punctuation:

```
c = regexp(str, '[^aeiouAEIOU]', 'match')
c =
  Columns 1 through 12
   ' ' 'N' 'R' 'M' 'S' 'T' ' ' 's' 't' 'm' 't'
   -- etc. --
```

Try this again, using a lookahead operator to create the following AND condition:

```
(lowercase letter) AND (not a vowel).
```

This time, the result is correct:

```
c = regexp(str, '(?=[a-z])[^aeiou]', 'match')
c =
  's' 't' 'm' 't' 't' 'h' 'm' 't' 'r' 'x'
  'n' 'r' 'm'
```

Note that when using a lookahead operator to perform an AND, you need to place the match expression *expr* *after* the test expression *test*:

`(?=test)expr` or `(?!test)expr`

Quantifiers

With the quantifiers shown below, you can specify how many instances of an element are to be matched. The basic quantifying operators are listed in the first six rows of the table.

By default, MATLAB matches as much of an expression as possible. Using the operators shown in the last two rows of the table, you can override this default behavior. Specify these options by appending a `+` or `?` immediately following one of the six basic quantifying operators.

Operator	Usage
<code>expr{m,n}</code>	Must occur at least <i>m</i> times but no more than <i>n</i> times.
<code>expr{m,}</code>	Must occur at least <i>m</i> times.
<code>expr{n}</code>	Must match exactly <i>n</i> times. Equivalent to <code>{n,n}</code> .
<code>expr?</code>	Match the preceding element 0 times or 1 time. Equivalent to <code>{0,1}</code> .
<code>expr*</code>	Match the preceding element 0 or more times. Equivalent to <code>{0,}</code> .
<code>expr+</code>	Match the preceding element 1 or more times. Equivalent to <code>{1,}</code> .
<code>q_expr+</code>	Match as much of the quantified expression as possible, but do not rescan any portions of the string if the initial match fails. The term <code>q_expr</code> represents any of the expressions shown in the top six rows of this table.
<code>q_expr?</code>	Match only as much of the quantified expression as necessary. The term <code>q_expr</code> represents any of the expressions shown in the top six rows of this table. For an example, see “Lazy Quantifiers — <code>expr*?</code> ” on page 2-70, below.

Zero or One – expr?

Use ? to make the HTML <code> and </code> tags optional in the string. The first string, hstr1, contains one occurrence of each tag. Since the expression uses ()? around the tags, one occurrence is a match:

```
hstr1 = '<td><a name="18854"></a><code>%%</code><br></td>';
expr = '</a>( <code>)?..(</code>)?<br>';
```

```
regexp(hstr1, expr, 'match')
ans =
    '</a><code>%%</code><br>'
```

The second string, hstr2, does not contain the code tags at all. Just the same, the expression matches because ()? allows for zero occurrences of the tags:

```
hstr2 = '<td><a name="18854"></a>%%<br></td>';
expr = '</a>( <code>)?..(</code>)?<br>';
```

```
regexp(hstr2, expr, 'match')
ans =
    '</a>%%<br>'
```

Zero or More – expr*

The first regexp command looks for at least one occurrence of
 and finds it. The second command parses a different string for at least one
 and fails. The third command uses * to parse the same line for zero or more line breaks and this time succeeds.

```
hstr1 = '<p>This string has <br><br>line breaks</p>';
regexp(hstr1, '<p>.*( <br>).*</p>', 'match')
ans =
    '<p>This string has <br><br>line breaks</p>';
```

```
hstr2 = '<p>This string has no line breaks</p>';
regexp(hstr2, '<p>.*( <br>).*</p>', 'match')
ans =
    {}
```

```
regexp(hstr2, '<p>.*( <br>)*.</p>', 'match')
```

```
ans =  
  '<p>This string has no line breaks</p>';
```

One or More – expr+

Use + to verify that the HTML image source is not empty. This looks for one or more characters in the gif filename:

```
hstr = '<a href="s12.html">';  
expr = '</a><a href="s13.html#18760">';  
expr = '<a href="\w{1,}(\.html){1}(\#\d{5,8}){0,1}";
```

```
regexp(hstr, expr, 'match')  
ans =  
  '<a href="s13.html#18760"'
```

Lazy Quantifiers – expr*?

This example shows the difference between the default (*greedy*) quantifier and the *lazy* quantifier (?). The first part of the example uses the default quantifier to match all characters from the opening <tr to the ending </td>:

```
hstr = '<tr valign=top><td><a name="19184"></a><br></td>';  
regexp(hstr, '</?t.*>', 'match')  
ans =
```

```
'<tr valign=top><td><a name="19184"></a><br></td>'
```

The second part uses the lazy quantifier to match the minimum number of characters between `<tr`, `<td`, or `</td` tags:

```
regexp(hstr, '</?t.*?>', 'match')
ans =
    '<tr valign=top>'    '<td>'    '</td>'
```

Tokens

Parentheses used in a regular expression not only group elements of that expression together, but also designate any matches found for that group as *tokens*. You can use tokens to match other parts of the same string. One advantage of using tokens is that they remember what they matched, so you can recall and reuse matched text in the process of searching or replacing.

This section covers

- “Operators Used with Tokens” on page 2-71
- “Introduction to Using Tokens” on page 2-72
- “Using Tokens — Example 1” on page 2-73
- “Using Tokens — Example 2” on page 2-73
- “Tokens That Are Not Matched” on page 2-74
- “Using Tokens in a Replacement String” on page 2-76

Operators Used with Tokens

Here are the operators you can use with tokens in MATLAB.

Operator	Usage
(expr)	Capture in a token all characters matched by the expression within the parentheses.
\N	Match the N th token generated by this command. That is, use \1 to match the first token, \2 to match the second, and so on.

Operator	Usage
\$N	Insert the match for the N th token in the replacement string. Used only by the <code>regexprep</code> function. If N is equal to zero, then insert the entire match in the replacement string.
(? (N) s1 s2)	If N th token is found, then match s1, else match s2

Introduction to Using Tokens

You can turn any pattern being matched into a token by enclosing the pattern in parentheses within the expression. For example, to create a token for a dollar amount, you could use `'(\$\d+)'`. Each token in the expression is assigned a number, starting from 1, going from left to right. To make a reference to a token later in the expression, refer to it using a backslash followed by the token number. For example, when referencing a token generated by the third set of parentheses in the expression, use `\3`.

As a simple example, if you wanted to search for identical sequential letters in a string, you could capture the first letter as a token and then search for a matching character immediately afterwards. In the expression shown below, the `(\S)` phrase creates a token whenever `regexp` matches any nonwhitespace character in the string. The second part of the expression, `'\1'`, looks for a second instance of the same character immediately following the first:

```
poestr = ['While I nodded, nearly napping, ' ...
         'suddenly there came a tapping,'];

[mat tok ext] = regexp(poestr, '(\S)\1', 'match', ...
    'tokens', 'tokenExtents');
mat
mat =
    'dd'    'pp'    'dd'    'pp'
```

The tokens returned in cell array `tok` are:

```
'd', 'p', 'd', 'p'
```

Starting and ending indices for each token in the input string `poestr` are:

11 11, 26 26, 35 35, 57 57

Using Tokens – Example 1

Here is an example of how tokens are assigned values. Suppose that you are going to search the following text:

andy ted bob jim andrew andy ted mark

You choose to search the above text with the following search pattern:

`and(y|rew)|(t)e(d)`

This pattern has three parenthetical expressions that generate tokens. When you finally perform the search, the following tokens are generated for each match.

Match	Token 1	Token 2
andy	y	
ted	t	d
andrew	rew	
andy	y	
ted	t	d

Only the highest level parentheses are used. For example, if the search pattern `and(y|rew)` finds the text `andrew`, token 1 is assigned the value `rew`. However, if the search pattern `(and(y|rew))` is used, token 1 is assigned the value `andrew`.

Using Tokens – Example 2

Use `(expr)` and `\N` to capture pairs of matching HTML tags (e.g., `<a>` and ``) and the text between them. The expression used for this example is

```
expr = '<(\w+).*?>.*?</\1>';
```

The first part of the expression, '`(<(\w+)`', matches an opening bracket (`<`) followed by one or more alphabetic, numeric, or underscore characters. The enclosing parentheses capture token characters following the opening bracket.

The second part of the expression, '`. *?>. *?`', matches the remainder of this HTML tag (characters up to the `>`), and any characters that may precede the next opening bracket.

The last part, '`</\1>`', matches all characters in the ending HTML tag. This tag is composed of the sequence `</tag>`, where `tag` is whatever characters were captured as a token.

```
hstr = '!comment<a name="752507"></a><b>Default</b><br>';  
expr = '<(\w+). *?>. *?</\1>';
```

```
[mat tok] = regexp(hstr, expr, 'match', 'tokens');
```

```
mat{:}
```

```
ans =
```

```
    <a name="752507"></a>
```

```
ans =
```

```
    <b>Default</b>
```

```
tok{:}
```

```
ans =
```

```
    'a'
```

```
ans =
```

```
    'b'
```

Tokens That Are Not Matched

For those tokens specified in the regular expression that have no match in the string being evaluated, `regexp` and `regexpi` return an empty string ('') as the token output, and an extent that marks the position in the string where the token was expected.

The example shown here executes `regexp` on the path string `str` returned from the MATLAB `tempdir` function. The regular expression `expr` includes six token specifiers, one for each piece of the path string. The third specifier

[a-z]+ has no match in the string because this part of the path, Profiles, begins with an uppercase letter:

```
str = tempdir
str =
    C:\WINNT\Profiles\bascal\LOCALS-1\Temp\

expr = ['([A-Z:)]\ (WINNT)\ ([a-z]+)?.*\ ' ...
        '([a-z]+)\ ([A-Z]+~\d)\ (Temp)\ '];

[tok ext] = regexp(str, expr, 'tokens', 'tokenExtents');
```

When a token is not found in a string, MATLAB still returns a token string and token extent. The returned token string is an empty character string (''). The first number of the extent is the string index that marks where the token was expected, and the second number of the extent is equal to one less than the first.

In the case of this example, the empty token is the third specified in the expression, so the third token string returned is empty:

```
tok{:}
ans =
    'C:'      'WINNT'      ''      'bascal'      'LOCALS-1'      'Temp'
```

The third token extent returned in the variable `ext` has the starting index set to 10, which is where the nonmatching substring, Profiles, begins in the string. The ending extent index is set to one less than the starting index, or 9:

```
ext{:}
ans =
     1     2
     4     8
    10     9
    19    25
    27    34
    36    39
```

Using Tokens in a Replacement String

When using tokens in a replacement string, reference them using \$1, \$2, etc. instead of \1, \2, etc. This example captures two tokens and reverses their order. The first, \$1, is 'Norma Jean' and the second, \$2, is 'Baker'. Note that `regexprep` returns the modified string, not a vector of starting indices.

```
regexprep('Norma Jean Baker', '(\w+\s\w+)\s(\w+)', '$2, $1')
ans =
    Baker, Norma Jean
```

Named Capture

If you use a lot of tokens in your expressions, it may be helpful to assign them names rather than having to keep track of which token number is assigned to which token. Use the following operator to assign a name to a token that finds a match.

Operator	Usage
(?<name>expr)	Capture in a token all characters matched by the expression within the parentheses. Assign a name to the token.
\k<name>	Match the token referred to by name.
\$<name>	Insert the match for named token in a replacement string. Used only with the <code>regexprep</code> function.
(?(name)s1 s2)	If named token is found, then match s1; otherwise, match s2

When referencing a named token within the expression, use the syntax `\k<name>` instead of the numeric `\1`, `\2`, etc.:

```
poestr = ['While I nodded, nearly napping, ' ...
         'suddenly there came a tapping,'];

regexprep(poestr, '(?<anychar>.)\k<anychar>', 'match')
ans =
    'dd'    'pp'    'dd'    'pp'
```

Labeling Your Output

Named tokens can also be useful in labeling the output from the MATLAB regular expression functions. This is especially true when you are processing numerous strings.

This example parses different pieces of street addresses from several strings. A short name is assigned to each token in the expression string:

```
str1 = '134 Main Street, Boulder, CO, 14923';
str2 = '26 Walnut Road, Topeka, KA, 25384';
str3 = '847 Industrial Drive, Elizabeth, NJ, 73548';

p1 = '(?<adrs>\d+\s\S+\s(Road|Street|Avenue|Drive))';
p2 = '(?<city>[A-Z][a-z]+)';
p3 = '(?<state>[A-Z]{2})';
p4 = '(?<zip>\d{5})';

expr = [p1 ' ', ' p2 ', ' p3 ', ' p4];
```

As the following results demonstrate, you can make your output easier to work with by using named tokens:

```
loc1 = regexp(str1, expr, 'names')
loc1 =
    adrs: '134 Main Street'
    city: 'Boulder'
    state: 'CO'
    zip: '14923'

loc2 = regexp(str2, expr, 'names')
loc2 =
    adrs: '26 Walnut Road'
    city: 'Topeka'
    state: 'KA'
    zip: '25384'

loc3 = regexp(str3, expr, 'names')
loc3 =
    adrs: '847 Industrial Drive'
    city: 'Elizabeth'
```

```
state: 'NJ'
zip: '73548'
```

Conditional Expressions

With conditional expressions, you can tell MATLAB to match an expression only if a certain condition is true. A conditional expression is similar to an if-then or an if-then-else clause in programming. MATLAB first tests the state of a given condition, and the outcome of this tests determines what, if anything, is to be matched next. The following table shows the two conditional syntaxes you can use with MATLAB.

Operator	Usage
(?(cond)expr)	If condition cond is true, then match expression expr
(?(cond)expr ₁ expr ₂)	If condition cond is true, then match expression expr ₁ . Otherwise match expression expr ₂

The first entry in this table is the same as an if-then statement. MATLAB tests the state of condition cond and then matches expression expr only if the condition was found to be true. In the form of an if-then statement, it would look like this:

```
if cond then expr
```

The second entry in the table is the same as an if-then-else statement. If the condition is true, MATLAB matches expr₁; if false, it matches expr₂ instead. This syntax is equivalent to the following programming statement:

```
if cond then expr1 else expr2
```

The condition cond in either of these syntaxes can be any one of the following:

- A specific token, identified by either number or name, is located in the input string. See “Conditions Based on Tokens” on page 2-79, below.
- A lookaround operation results in a match. See “Conditions Based on a Lookaround Match” on page 2-80, below.
- A dynamic expression of the form (?@cmd) returns a nonzero numeric value. See “Conditions Based on Return Values” on page 2-80, below.

Conditions Based on Tokens

In a conditional expression, MATLAB matches the expression only if the condition associated with it is met. If the condition is based on a token, then the condition is met if MATLAB matches more than one character for the token in the input string.

To specify a token in a condition, use either the token number or, for tokens that you have assigned a name to, its name. Token numbers are determined by the order in which they appear in an expression. For example, if you specify three tokens in an expression (that is, if you enclose three parts of the expression in parentheses), then you would refer to these tokens in a condition statement as 1, 2, and 3.

The following example uses the conditional statement `(?(1)her|his)` to match the string regardless of the gender used. You could translate this into the phrase, “**if** token 1 is found (i.e., Mr is followed by the letter s), **then** match her, **else** match his:

```
expr = 'Mr(s?)\..*?(?(1)her|his) son';

[mat tok] = regexp('Mr. Clark went to see his son', ...
    expr, 'match', 'tokens')
mat =
    'Mr. Clark went to see his son'
tok =
    {1x2 cell}

tok{:}
ans =
    ''      'his'
```

In the second part of the example, the token s is found and MATLAB matches the word her:

```
[mat tok] = regexp('Mrs. Clark went to see her son', ...
    expr, 'match', 'tokens')
mat =
    'Mrs. Clark went to see her son'
tok =
    {1x2 cell}
```

```
tok{:}
ans =
     's'     'her'
```

Note When referring to a token within a condition, use just the number of the token. For example, refer to token 2 by using the number 2 alone, and not `\2` or `$2`.

Conditions Based on a Lookaround Match

Lookaround statements look for text that either precedes or follows an expression. If this lookaround text is located, then MATLAB proceeds to match the expression. You can also use lookarounds in conditional statements. In this case, if the lookaround text is located, then MATLAB considers the condition to be met and matches the associated expression. If the condition is not met, then MATLAB matches the `else` part of the expression.

Conditions Based on Return Values

MATLAB supports different types of dynamic expressions. One type of dynamic expression, having the form `(?@cmd)`, enables you to execute a MATLAB command (shown here as `cmd`) while matching an expression. You can use this type of dynamic expression in a conditional statement if the command in the expression returns a numeric value. The condition is considered to be met if the return value is nonzero.

Dynamic Regular Expressions

In a dynamic expression, you can make the pattern that you want `regexp` to match dependent on the content of the input string. In this way, you can more closely match varying input patterns in the string being parsed. You can also use dynamic expressions in replacement strings for use with the `regexprep` function. This gives you the ability to adapt the replacement text to the parsed input.

You can include any number of dynamic expressions in the `match_expr` or `replace_expr` arguments of these commands:

```

regexp(string, match_expr)
regexpi(string, match_expr)
regexprep(string, match_expr, replace_expr)

```

MATLAB supports three types of dynamic operators for use in a match expression. See “Dynamic Operators for the Match Expression” on page 2-82 for more information.

Operator	Usage
(? <i>expr</i>)	Parse <i>expr</i> as a separate regular expression, and include the resulting string in the match expression. This gives you the same results as if you called <code>regexprep</code> inside of a <code>regexp</code> match expression.
(? <i>@cmd</i>)	Execute the MATLAB command <i>cmd</i> , discarding any output that may be returned. This is often used for diagnosing a regular expression.
(? <i>@cmd</i>)	Execute the MATLAB command <i>cmd</i> , and include the string returned by <i>cmd</i> in the match expression. This is a combination of the two dynamic syntaxes shown above: (<i>?<i>expr</i></i>) and (<i>?<i>@cmd</i></i>).

MATLAB supports one type of dynamic expression for use in the replacement expression of a `regexprep` command. See “Dynamic Operators for the Replacement Expression” on page 2-87 for more information.

Operator	Usage
<code>\${<i>cmd</i>}</code>	Execute the MATLAB command <i>cmd</i> , and include the string returned by <i>cmd</i> in the replacement expression.

Example of a Dynamic Expression

As an example of a dynamic expression, the following `regexprep` command correctly replaces the term `internationalization` with its abbreviated form, `i18n`. However, to use it on a different term such as `globalization`, you have to use a different replacement expression:

```
match_expr = '^(\w)(\w*)(\w$)';
```

```
replace_expr1 = '$118$3';
regexprep('internationalization', match_expr, replace_expr1)
ans =
    i18n
```

```
replace_expr2 = '$111$3';
regexprep('globalization', match_expr, replace_expr2)
ans =
    g11n
```

Using a dynamic expression `${num2str(length($2))}` enables you to base the replacement expression on the input string so that you do not have to change the expression each time. This example uses the dynamic syntax `{cmd}` from the second table shown above:

```
match_expr = '^(\\w)(\\w*)(\\w$)';
replace_expr = '$1${num2str(length($2))}$3';

regexprep('internationalization', match_expr, replace_expr)
ans =
    i18n

regexprep('globalization', match_expr, replace_expr)
ans =
    g11n
```

Dynamic Operators for the Match Expression

There are three types of dynamic expressions you can use when composing a match expression:

- “Dynamic Expressions That Modify the Match Expression — `(??expr)`” on page 2-83
- “Dynamic Commands That Modify the Match Expression — `(??@cmd)`” on page 2-84
- “Dynamic Commands That Serve a Functional Purpose — `(?@cmd)`” on page 2-85

The first two of these actually modify the match expression itself so that it can be made specific to changes in the contents of the input string. When

MATLAB evaluates one of these dynamic statements, the results of that evaluation are included in the same location within the overall match expression.

The third operator listed here does not modify the overall expression, but instead enables you to run MATLAB commands during the parsing of a regular expression. This functionality can be useful in diagnosing your regular expressions.

Dynamic Expressions That Modify the Match Expression – (??expr).

The (??expr) operator parses expression expr, and inserts the results back into the match expression. MATLAB then evaluates the modified match expression.

Here is an example of the type of expression that you can use with this operator:

```
str = {'5XXXXX', '8XXXXXXXX', '1X'};
regexp(str, '^(\\d+)(??X{\\$1})$', 'match', 'once')
```

The purpose of this particular command is to locate a series of X characters in each of the strings stored in the input cell array. Note however that the number of Xs varies in each string. If the count did not vary, you could use the expression X{n} to indicate that you want to match n of these characters. But, a constant value of n does not work in this case.

The solution used here is to capture the leading count number (e.g., the 5 in the first string of the cell array) in a token, and then to use that count in a dynamic expression. The dynamic expression in this example is (??X{\\\$1}), where \\\$1 is the value captured by the token \\d+. The operator {\\\$1} makes a quantifier of that token value. Because the expression is dynamic, the same pattern works on all three of the input strings in the cell array. With the first input string, regexp looks for five X characters; with the second, it looks for eight, and with the third, it looks for just one:

```
regexp(str, '^(\\d+)(??X{\\$1})$', 'match', 'once')
ans =
    '5XXXXX'    '8XXXXXXXX'    '1X'
```

Dynamic Commands That Modify the Match Expression – (??@cmd).

MATLAB uses the (??@function) operator to include the results of a MATLAB command in the match expression. This command must return a string that can be used within the match expression.

The `regexp` command below uses the dynamic expression (??@fliplr(\$1)) to locate a palindrome string, “Never Odd or Even”, that has been embedded into a larger string:

```
regexp(pstr, '(.{3,}).?(??@fliplr($1))', 'match')
```

The dynamic expression reverses the order of the letters that make up the string, and then attempts to match as much of the reversed-order string as possible. This requires a dynamic expression because the value for \$1 relies on the value of the token (.{3,}):

```
% Put the string in lowercase.
str = lower(...
    'Find the palindrome Never Odd or Even in this string');

% Remove all nonword characters.
str = regexprep(str, '\W*', '')
str =
    findthepalindromeneveroddoeveninthisstring

% Now locate the palindrome within the string.
palstr = regexp(str, '(.{3,}).?(??@fliplr($1))', 'match')
str =
    'neveroddoeven'
```

Dynamic expressions in MATLAB have access to the currently active workspace. This means that you can change any of the functions or variables used in a dynamic expression just by changing variables in the workspace. Repeat the last command of the example above, but this time define the function to be called within the expression using a function handle stored in the base workspace:

```
fun = @fliplr;

palstr = regexp(str, '(.{3,}).?(??@fun($1))', 'match')
palstr =
```

```
'neverodddoreven'
```

Dynamic Commands That Serve a Functional Purpose — (?@cmd). The (?@cmd) operator specifies a MATLAB command that `regexp` or `regexprep` is to run while parsing the overall match expression. Unlike the other dynamic expressions in MATLAB, this operator does not alter the contents of the expression it is used in. Instead, you can use this functionality to get MATLAB to report just what steps it is taking as it parses the contents of one of your regular expressions.

The following example parses a word for zero or more characters followed by two identical characters followed again by zero or more characters:

```
regexp('mississippi', '\w*(\w)\1\w*', 'match')
ans =
    'mississippi'
```

To track the exact steps that MATLAB takes in determining the match, the example inserts a short script (?@disp(\$1)) in the expression to display the characters that finally constitute the match. Because the example uses greedy quantifiers, MATLAB attempts to match as much of the string as possible. So, even though MATLAB finds a match toward the beginning of the string, it continues to look for more matches until it arrives at the very end of the string. From there, it backs up through the letters `i` then `p` and the next `p`, stopping at that point because the match is finally satisfied:

```
regexp('mississippi', '\w*(\w)(?@disp($1))\1\w*');
i
p
p
```

Now try the same example again, this time making the first quantifier lazy (*?). Again, MATLAB makes the same match:

```
regexp('mississippi', '\w*(\w)\1\w*', 'match')
ans =
    'mississippi'
```

But by inserting a dynamic script, you can see that this time, MATLAB has matched the string quite differently. In this case, MATLAB uses the very first match it can find, and does not even consider the rest of the string:

```
regexp('mississippi', '\w*(\w)(?@disp($1))\1\w*');  
m  
i  
s
```

To demonstrate how versatile this type of dynamic expression can be, consider the next example that progressively assembles a cell array as MATLAB iteratively parses the input string. The (?!) operator found at the end of the expression is actually an empty lookahead operator, and forces a failure at each iteration. This forced failure is necessary if you want to trace the steps that MATLAB is taking to resolve the expression.

MATLAB makes a number of passes through the input string, each time trying another combination of letters to see if a fit better than last match can be found. On any passes in which no matches are found, the test results in an empty string. The dynamic script (?@if(~isempty(\$&))) serves to omit these strings from the matches cell array:

```
matches = {};  
expr = ['(Euler\s)?(Cauchy\s)?(Boole)?(?@if(~isempty($&)), ' ...  
      'matches{end+1}=$&;end) (?!)'];  
  
regexp('Euler Cauchy Boole', expr);  
  
matches  
matches =  
      'Euler Cauchy Boole'      'Euler Cauchy '      'Euler '  
'Cauchy Boole'      'Cauchy '      'Boole'
```

The operators \$& (or the equivalent \$0), \$`, and \$' refer to that part of the input string that is currently a match, all characters that precede the current match, and all characters to follow the current match, respectively. These operators are sometimes useful when working with dynamic expressions, particularly those that employ the (?@cmd) operator.

This example parses the input string looking for the letter g. At each iteration through the string, regexp compares the current character with g, and not finding it, advances to the next character. The example tracks the progress of scan through the string by marking the current location being parsed with a ^ character.

(The `$`` and `$·` operators capture that part of the string that precedes and follows the current parsing location. You need two single-quotation marks (`$'`) to express the sequence `$·` when it appears within a string.)

```
str = 'abcdefghij';
expr = '(?@disp(sprintf('starting match: [%s^%s]','$', $'))g';

regexp(str, expr, 'once');
starting match: [^abcdefghij]
starting match: [a^abcdefghij]
starting match: [ab^cdefghij]
starting match: [abc^defghij]
starting match: [abcd^efghij]
starting match: [abcde^fghij]
starting match: [abcdef^ghij]
```

Dynamic Operators for the Replacement Expression

The three types of dynamic expressions discussed above can be used only in the match expression (second input) argument of the regular expression functions. MATLAB provides one more type of dynamic expression; this one is for use in a replacement string (third input) argument of the `regexprep` function.

Dynamic Commands That Modify the Replacement Expression —

`${cmd}`. The `${cmd}` operator modifies the contents of a regular expression replacement string, making this string adaptable to parameters in the input string that might vary from one use to the next. As with the other dynamic expressions used in MATLAB, you can include any number of these expressions within the overall replacement expression.

In the `regexprep` call shown here, the replacement string is `'${convert($1,$2)}'`. In this case, the entire replacement string is a dynamic expression:

```
regexprep('This highway is 125 miles long', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
```

The dynamic expression tells MATLAB to execute a function named `convert` using the two tokens `(\d+\.?\d*)` and `(\w+)`, derived from the string being

matched, as input arguments in the call to `convert`. The replacement string requires a dynamic expression because the values of `$1` and `$2` are generated at runtime.

The following example defines the file named `convert` that converts measurements from imperial units to metric. To convert values from the string being parsed, `regexprep` calls the `convert` function, passing in values for the quantity to be converted and name of the imperial unit:

```
function valout = convert(valin, units)
switch(units)
    case 'inches'
        fun = @(in)in .* 2.54;    uout = 'centimeters';
    case 'miles'
        fun = @(mi)mi .* 1.6093; uout = 'kilometers';
    case 'pounds'
        fun = @(lb)lb .* 0.4536; uout = 'kilograms';
    case 'pints'
        fun = @(pt)pt .* 0.4731; uout = 'litres';
    case 'ounces'
        fun = @(oz)oz .* 28.35;  uout = 'grams';
end
val = fun(str2num(valin));
valout = [num2str(val) ' ' uout];
```

```
regexprep('This highway is 125 miles long', ...
          '(\d+\.\?\d*)\W(\w+)', '{convert($1,$2)}')
ans =
    This highway is 201.1625 kilometers long
```

```
regexprep('This pitcher holds 2.5 pints of water', ...
          '(\d+\.\?\d*)\W(\w+)', '{convert($1,$2)}')
ans =
    This pitcher holds 1.1828 litres of water
```

```
regexprep('This stone weighs about 10 pounds', ...
          '(\d+\.\?\d*)\W(\w+)', '{convert($1,$2)}')
```

```
ans =
    This stone weighs about 4.536 kilograms
```

As with the (??@) operator discussed in an earlier section, the \${ } operator has access to variables in the currently active workspace. The following regexprep command uses the array A defined in the base workspace:

```
A = magic(3)
A =
     8     1     6
     3     5     7
     4     9     2

regexprep('The columns of matrix _nam are _val', ...
          {'_nam', '_val'}, ...
          {'A', '${sprintf('%d%d%d ', A)}'})
ans =
The columns of matrix A are 834 159 672
```

String Replacement

The regexprep function enables you to replace a string that is identified by a regular expression with another string. The following syntax replaces all occurrences of the regular expression `expr` in string `str` with the string `repstr`. The new string is returned in `s`. If no matches are found, return string `s` is the same as input string `str`.

```
s = regexprep('str', 'expr', 'repstr')
```

The replacement string can include any ordinary characters and also any of the operators shown in the following table.

Operator	Usage
Operators from the “Character Representation” on page 2-57 table	The character represented by the operator sequence
\$`	That part of the input string that precedes the current match

Operator	Usage
<code>\$& or \$0</code>	That part of the input string that is currently a match
<code>\$'</code>	That part of the input string that follows the current match. In MATLAB, use <code>\$'</code> to represent the character sequence <code>\$'</code>
<code>\$N</code>	The string represented by the token identified by the number <code>N</code>
<code>\$<name></code>	The string represented by the token identified by name
<code>\${cmd}</code>	The string returned when MATLAB executes the command <code>cmd</code>

You can capture parts of the input string as tokens and then reuse them in the replacement string. Specify the parts of the string to capture using the token capture operator (`. . .`). Specify the tokens to use in the replacement string using the operators `$1`, `$2`, `$N` to reference the first, second, and `N`th tokens captured. (See the section on “Tokens” on page 2-71 and the example “Using Tokens in a Replacement String” on page 2-76 in this documentation for information on using tokens.)

Note When referring to a token within a replacement string, use the number of the token preceded by a dollar sign. For example, refer to token 2 by using `$2`, and not `2` or `\2`.

The following example uses both the `${cmd}` and `$N` operators in the replacement strings of nested `regexprep` commands to capitalize the first letter of each sentence. The inner `regexprep` looks for the start of the entire string and capitalizes the single instance; the outer `regexprep` looks for the first letter following a period and capitalizes the two instances:

```
s1 = 'here are a few sentences.';
s2 = 'none are capitalized.';
s3 = 'let''s change that.';
```



```
str = [s1 ' ' s2 ' ' s3]

regexprep(regexprep(str, '^.', '${upper($1)}'), ...
    '(?<=\\.\\s*)([a-z])', '${upper($1)}')

ans =
Here are a few sentences. None are capitalized. Let's change that.
```

Make `regexprep` more specific to your needs by specifying any of a number of options with the command. See the `regexprep` reference page for more information on these options.

Handling Multiple Strings

You can use any of the MATLAB regular expression functions with cell arrays of strings as well as with single strings. Any or all of the input parameters (the string, expression, or replacement string) can be a cell array of strings. The `regexp` function requires that the string and expression arrays have the same number of elements. The `regexprep` function requires that the expression and replacement arrays have the same number of elements. (The cell arrays do not have to have the same shape.)

Whenever either input argument in a call to `regexp`, or the first input argument in a call to `regexprep` function is a cell array, all output values are cell arrays of the same size.

Function, Mode Options, Operator, Return Value Summaries

- “Function Summary” on page 2-92
- “Mode Options Summary” on page 2-92
- “Operator Summary” on page 2-93
- “Return Value Summary” on page 2-99

Function Summary

MATLAB Regular Expression Functions

Function	Description
regexp	Match regular expression.
regexpi	Match regular expression, ignoring case.
regexprep	Replace string using regular expression.
regexprtranslate	Translate string into regular expression.

Mode Options Summary

Mode Keyword	Flag	Description
'ignorecase'	(?i)	Do not consider letter case when matching patterns to a string (the default for regexpi).
'matchcase'	(?-i)	Letter case must match when matching patterns to a string (the default for regexp).
'noemptymatch'	N/A	Do not allow successful matches of length zero (the default).
'emptymatch'	N/A	Allow successful matches of length zero.
'dotall'	(?s)	Match dot ('.') in the pattern string with any character (the default).
'dotexceptnewline'	(?-s)	Match dot in the pattern with any character that is not a newline.
'lineanchors'	(?m)	Match the ^ and \$ metacharacters at the beginning and end of a line.
'stringanchors'	(?-m)	Match the ^ and \$ metacharacters at the beginning and end of a string (the default).

Mode Keyword	Flag	Description
'freespacing'	(?x)	Ignore spaces and comments when parsing the string. (You must use '\ ' and '\#' to match space and # characters.)
'literalspacing'	(?-x)	Parse space characters and comments (the # character and any text to the right of it) in the same way as any other characters in the string (the default).

Operator Summary

Character Types

Operator	Usage
.	Any single character, including white space
[c ₁ c ₂ c ₃]	Any character contained within the brackets: c ₁ or c ₂ or c ₃
[^c ₁ c ₂ c ₃]	Any character not contained within the brackets: anything but c ₁ or c ₂ or c ₃
[c ₁ -c ₂]	Any character in the range of c ₁ through c ₂
\s	Any white-space character; equivalent to [\f\n\r\t\v]
\S	Any nonwhitespace character; equivalent to [^\f\n\r\t\v]
\w	Any alphabetic, numeric, or underscore character (For English character sets, this is equivalent to [a-zA-Z_0-9].)
\W	Any character that is not alphabetic, numeric, or underscore (For English character sets, this is equivalent to [^a-zA-Z_0-9].)
\d	Any numeric digit; equivalent to [0-9]
\D	Any nondigit character; equivalent to [^0-9]

Character Types (Continued)

Operator	Usage
<code>\oN</code> or <code>\o{N}</code>	Character of octal value N
<code>\xN</code> or <code>\x{N}</code>	Character of hexadecimal value N

Character Representation

Operator	Usage
<code>\\</code>	Backslash
<code>\a</code>	Alarm (beep)
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Horizontal tab
<code>\v</code>	Vertical tab
<code>\char</code>	If a character has special meaning in a regular expression, precede it with backslash (<code>\</code>) to match it literally.

Grouping Operators

Operator	Usage
<code>(expr)</code>	Group regular expressions and capture tokens.
<code>(?:expr)</code>	Group regular expressions, but do not capture tokens.
<code>(?>expr)</code>	Group atomically.
<code>expr₁ expr₂</code>	Match expression <code>expr₁</code> or expression <code>expr₂</code> .

Nonmatching Operators

Operator	Usage
(?#comment)	Insert a comment into the expression. Comments are ignored in matching.

Positional Operators

Operator	Usage
^expr	Match expr if it occurs at the beginning of the input string.
expr\$	Match expr if it occurs at the end of the input string.
\<expr	Match expr when it occurs at the beginning of a word.
expr\>	Match expr when it occurs at the end of a word.
\<expr\>	Match expr when it represents the entire word.

Lookaround Operators

Operator	Usage
(?=expr)	Look ahead from current position and test if expr is found.
(?!expr)	Look ahead from current position and test if expr is not found
(?<=expr)	Look behind from current position and test if expr is found.
(?<!expr)	Look behind from current position and test if expr is not found.

Quantifiers

Operator	Usage
<code>expr{m,n}</code>	Match <code>expr</code> when it occurs at least <code>m</code> times but no more than <code>n</code> times consecutively.
<code>expr{m,}</code>	Match <code>expr</code> when it occurs at least <code>m</code> times consecutively.
<code>expr{n}</code>	Match <code>expr</code> when it occurs exactly <code>n</code> times consecutively. Equivalent to <code>{n,n}</code> .
<code>expr?</code>	Match <code>expr</code> when it occurs 0 times or 1 time. Equivalent to <code>{0,1}</code> .
<code>expr*</code>	Match <code>expr</code> when it occurs 0 or more times consecutively. Equivalent to <code>{0,}</code> .
<code>expr+</code>	Match <code>expr</code> when it occurs 1 or more times consecutively. Equivalent to <code>{1,}</code> .
<code>q_expr</code>	Match as much of the quantified expression as possible, where <code>q_expr</code> represents any of the expressions shown in the first six rows of this table.
<code>q_expr+</code>	Match as much of the quantified expression as possible, but do not rescan any portions of the string if the initial match fails.
<code>q_expr?</code>	Match only as much of the quantified expression as necessary.

Ordinal Token Operators

Operator	Usage
<code>(expr)</code>	Capture in a token all characters matched by the expression within the parentheses.
<code>\N</code>	Match the <code>Nth</code> token generated by this command. That is, use <code>\1</code> to match the first token, <code>\2</code> to match the second, and so on.

Ordinal Token Operators (Continued)

Operator	Usage
\$N	Insert the match for the N th token in the replacement string. If N is equal to zero, then insert the entire match in the replacement string. (Used only by the <code>regexprep</code> function.)
(?(N)s1 s2)	If N th token is found, then match s1; otherwise, match s2.

Named Token Operators

Operator	Usage
(?<name>expr)	Capture in a token all characters matched by the expression within the parentheses. Assign a name value to the token.
\k<name>	Match the token referred to by name.
\$<name>	Insert the match for named token in a replacement string. (Used only with the <code>regexprep</code> function.)
(?(name)s1 s2)	If named token is found, then match s1; otherwise, match s2.

Conditional Expression Operators

Operator	Usage
(?(cond)expr)	If condition <code>cond</code> is true, then match expression <code>expr</code> .
(?(cond)expr ₁ expr ₂)	If condition <code>cond</code> is true, then match expression <code>expr₁</code> . Otherwise match expression <code>expr₂</code> .

Dynamic Expression Operators

Operator	Usage
(??expr)	Parse expr as a separate regular expression, and include the resulting string in the match expression. This gives you the same results as if you called <code>regexprep</code> inside of a <code>regexp</code> match expression.
(??@cmd)	Execute the MATLAB command represented by cmd, and include the string returned by the command in the match expression. This is a combination of the two dynamic syntaxes shown previously: (??expr) and (?@cmd).
(?@cmd)	Execute the MATLAB command represented by cmd and discard any output the command returns. (Helpful for diagnosing regular expressions).
\${cmd}	Execute the MATLAB command represented by cmd, and include the string returned by the command in the replacement expression.

Replacement String Operators

Operator	Usage
Operators from “Character Representation” on page 2-57 table	The character represented by the operator sequence
\$'	That part of the input string that precedes the current match
\$& or \$0	That part of the input string that is currently a match
\$'	That part of the input string that follows the current match (In MATLAB, use \$' ' to represent the character sequence \$'.)
\$N	The string represented by the token identified by name

Replacement String Operators (Continued)

Operator	Usage
<code>\$<name></code>	The string represented by the token identified by name
<code>\${cmd}</code>	The string returned when MATLAB executes the command <code>cmd</code>

Return Value Summary

Qualifier	Description	Default Order
start	Row vector containing the starting index of each substring of <code>str</code> that matches <code>expr</code>	1
end	Row vector containing the ending index of each substring of <code>str</code> that matches <code>expr</code>	2
tokenExtents	Cell array containing the starting and ending indices of each substring of <code>str</code> that matches a token in <code>expr</code> (This is a double array when used with 'once'.)	3
match	Cell array containing the text of each substring of <code>str</code> that matches <code>expr</code> (This is a string when used with 'once'.)	4
tokens	Cell array of cell arrays of strings containing the text of each token captured by <code>regexp</code> (This is a cell array of strings when used with 'once'.)	5
names	Structure array containing the name and text of each <i>named</i> token captured by <code>regexp</code> (If there are no named tokens in <code>expr</code> , <code>regexp</code> returns a structure array with no fields.) Field names of the returned structure are set to the token names, and field values are the text of those tokens. Named tokens are generated by the expression <code>(?<tokenname>)</code> .	6
split	Cell array containing those parts of the input string that are delimited by substrings returned when using the <code>regexp</code> 'match' option	7

Comma-Separated Lists

In this section...
“What Is a Comma-Separated List?” on page 2-100
“Generating a Comma-Separated List” on page 2-100
“Assigning Output from a Comma-Separated List” on page 2-102
“Assigning to a Comma-Separated List” on page 2-103
“How to Use the Comma-Separated Lists” on page 2-104
“Fast Fourier Transform Example” on page 2-106

What Is a Comma-Separated List?

Typing in a series of numbers separated by commas gives you what is called a *comma-separated list*. The MATLAB software returns each value individually:

```
1, 2, 3
ans =
    1
ans =
    2
ans =
    3
```

Such a list, by itself, is not very useful. But when used with large and more complex data structures like MATLAB structures and cell arrays, the comma-separated list can enable you to simplify your MATLAB code.

Generating a Comma-Separated List

This section describes how to generate a comma-separated list from either a cell array or a MATLAB structure.

Generating a List from a Cell Array

Extracting multiple elements from a cell array yields a comma-separated list. Given a 4-by-6 cell array as shown here

```
C = cell(4, 6);
for k = 1:24, C{k} = k * 2; end
```

```
C
C =
     [2]     [10]     [18]     [26]     [34]     [42]
     [4]     [12]     [20]     [28]     [36]     [44]
     [6]     [14]     [22]     [30]     [38]     [46]
     [8]     [16]     [24]     [32]     [40]     [48]
```

extracting the fifth column generates the following comma-separated list:

```
C{:, 5}
ans =
    34
ans =
    36
ans =
    38
ans =
    40
```

This is the same as explicitly typing

```
C{1, 5}, C{2, 5}, C{3, 5}, C{4, 5}
```

Generating a List from a Structure

For structures, extracting a field of the structure that exists across one of its dimensions yields a comma-separated list.

Start by converting the cell array used above into a 4-by-1 MATLAB structure with six fields: `f1` through `f6`. Read field `f5` for all rows and MATLAB returns a comma-separated list:

```
S = cell2struct(C, {'f1', 'f2', 'f3', 'f4', 'f5', 'f6'}, 2);

S.f5
ans =
    34
ans =
```

```
36
ans =
38
ans =
40
```

This is the same as explicitly typing

```
S(1).f5, S(2).f5, S(3).f5, S(4).f5
```

Assigning Output from a Comma-Separated List

You can assign any or all consecutive elements of a comma-separated list to variables with a simple assignment statement. Using the cell array `C` from the previous section, assign the first row to variables `c1` through `c6`:

```
C = cell(4, 6);
for k = 1:24, C{k} = k * 2; end

[c1 c2 c3 c4 c5 c6] = C{1,1:6};

c5
c5 =
34
```

If you specify fewer output variables than the number of outputs returned by the expression, MATLAB assigns the first `N` outputs to those `N` variables, and then discards any remaining outputs. In this next example, MATLAB assigns `C{1,1:3}` to the variables `c1`, `c2`, and `c3`, and then discards `C{1,4:6}`:

```
[c1 c2 c3] = C{1,1:6};
```

You can assign structure outputs in the same manner:

```
S = cell2struct(C, {'f1', 'f2', 'f3', 'f4', 'f5', 'f6'}, 2);

[sf1 sf2 sf3] = S.f5;

sf3
sf3 =
38
```

You also can use the `deal` function for this purpose.

Assigning to a Comma-Separated List

The simplest way to assign multiple values to a comma-separated list is to use the `deal` function. This function distributes all of its input arguments to the elements of a comma-separated list.

This example initializes a comma-separated list to a set of vectors in a cell array, and then uses `deal` to overwrite each element in the list:

```
c{1} = [31 07];    c{2} = [03 78];
```

```
c{:}
ans =
    31     7
ans =
     3    78
```

```
[c{:}] = deal([10 20],[14 12]);
```

```
c{:}
ans =
    10    20
ans =
    14    12
```

This example does the same as the one above, but with a comma-separated list of vectors in a structure field:

```
s(1).field1 = [31 07];    s(2).field1 = [03 78];
```

```
s.field1
ans =
    31     7
ans =
     3    78
```

```
[s.field1] = deal([10 20],[14 12]);
```

```
s.field1
ans =
    10    20
ans =
    14    12
```

How to Use the Comma-Separated Lists

Common uses for comma-separated lists are

- “Constructing Arrays” on page 2-104
- “Displaying Arrays” on page 2-105
- “Concatenation” on page 2-105
- “Function Call Arguments” on page 2-105
- “Function Return Values” on page 2-106

The following sections provide examples of using comma-separated lists with cell arrays. Each of these examples applies to MATLAB structures as well.

Constructing Arrays

You can use a comma-separated list to enter a series of elements when constructing a matrix or array. Note what happens when you insert a *list* of elements as opposed to adding the cell itself.

When you specify a list of elements with `C{: , 5}`, MATLAB inserts the four individual elements:

```
A = {'Hello', C{: , 5}, magic(4)}
A =
    'Hello'    [34]    [36]    [38]    [40]    [4x4 double]
```

When you specify the `C` cell itself, MATLAB inserts the entire cell array:

```
A = {'Hello', C, magic(4)}
A =
    'Hello'    {4x6 cell}    [4x4 double]
```

Displaying Arrays

Use a list to display all or part of a structure or cell array:

```
A{:}
ans =
    Hello
ans =
    34
ans =
    36
ans =
    38
.
.
.
```

Concatenation

Putting a comma-separated list inside square brackets extracts the specified elements from the list and concatenates them:

```
A = [C{:}, 5:6]
A =
    34    36    38    40    42    44    46    48

whos A
  Name      Size      Bytes  Class

  A         1x8         64    double array
```

Function Call Arguments

When writing the code for a function call, you enter the input arguments as a list with each argument separated by a comma. If you have these arguments stored in a structure or cell array, then you can generate all or part of the argument list from the structure or cell array instead. This can be especially useful when passing in variable numbers of arguments.

This example passes several attribute-value arguments to the `plot` function:

```
X = -pi:pi/10:pi;
Y = tan(sin(X)) - sin(tan(X));

C{1,1} = 'LineWidth';           C{2,1} = 2;
C{1,2} = 'MarkerEdgeColor';    C{2,2} = 'k';
C{1,3} = 'MarkerFaceColor';    C{2,3} = 'g';

plot(X, Y, '--rs', C{:})
```

Function Return Values

MATLAB functions can also return more than one value to the caller. These values are returned in a list with each value separated by a comma. Instead of listing each return value, you can use a comma-separated list with a structure or cell array. This becomes more useful for those functions that have variable numbers of return values.

This example returns three values to a cell array:

```
C = cell(1, 3);
[C{:}] = fileparts('work/mytests/strArrays.mat')
C =
    'work/mytests'    'strArrays'    '.mat'
```

Fast Fourier Transform Example

The `fftshift` function swaps the left and right halves of each dimension of an array. For a simple vector such as `[0 2 4 6 8 10]` the output would be `[6 8 10 0 2 4]`. For a multidimensional array, `fftshift` performs this swap along each dimension.

`fftshift` uses vectors of indices to perform the swap. For the vector shown above, the index `[1 2 3 4 5 6]` is rearranged to form a new index `[4 5 6 1 2 3]`. The function then uses this index vector to reposition the elements. For a multidimensional array, `fftshift` must construct an index vector for each dimension. A comma-separated list makes this task much simpler.

Here is the `fftshift` function:

```
function y = fftshift(x)
```



```
numDims = ndims(x);
idx = cell(1, numDims);

for k = 1:numDims
    m = size(x, k);
    p = ceil(m/2);
    idx{k} = [p+1:m 1:p];
end

y = x(idx{:});
```

The function stores the index vectors in cell array `idx`. Building this cell array is relatively simple. For each of the N dimensions, determine the size of that dimension and find the integer index nearest the midpoint. Then, construct a vector that swaps the two halves of that dimension.

By using a cell array to store the index vectors and a comma-separated list for the indexing operation, `fftshift` shifts arrays of any dimension using just a single operation: `y = x(idx{:})`. If you were to use explicit indexing, you would need to write one `if` statement for each dimension you want the function to handle:

```
if ndims(x) == 1
    y = x(index1);
else if ndims(x) == 2
    y = x(index1, index2);
end
```

Another way to handle this without a comma-separated list would be to loop over each dimension, converting one dimension at a time and moving data each time. With a comma-separated list, you move the data just once. A comma-separated list makes it very easy to generalize the swapping operation to an arbitrary number of dimensions.

String Evaluation

In this section...
“Evaluate Expressions in Text Strings” on page 2-108
“Alternatives to the eval Function” on page 2-108

Evaluate Expressions in Text Strings

This example shows how to evaluate a text string that contains a MATLAB expression, statement, or function call using the `eval` function.

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 order `n`.

```
t = '1/(m + n - 1)';  
for m = 1:k  
    for n = 1:k  
        a(m,n) = eval(t);  
    end  
end
```

String evaluation allows you to execute user-supplied strings from an input statement or a GUI, or to construct commands by concatenating variable strings.

Alternatives to the eval Function

- “Why Avoid the eval Function?” on page 2-109
- “Variables with Sequential Names” on page 2-109
- “Files with Sequential Names” on page 2-110
- “Function Names in Variables” on page 2-111
- “Field Names in Variables” on page 2-111

- “Error Handling” on page 2-112

Why Avoid the eval Function?

Although the `eval` function is very powerful and flexible, it not always the best solution to a programming problem. Code that calls `eval` is often less efficient and more difficult to read and debug than code that uses other functions or language constructs. For example:

- MATLAB compiles code the first time you run it to enhance performance for future runs. However, because code in an `eval` statement can change at run time, it is not compiled.
- Code within an `eval` statement can unexpectedly create or assign to a variable already in the current workspace, overwriting existing data.
- Concatenating strings within an `eval` statement is often difficult to read. Other language constructs can simplify the syntax in your code.

For many common uses of `eval`, there are preferred alternate approaches, as shown in the following examples.

Variables with Sequential Names

A frequent use of the `eval` function is to create sets of variables such as `A1`, `A2`, . . . , `An`, but this approach does not use the array processing power of MATLAB and is not recommended. The preferred method is to store related data in a single array. If the data sets are of different types or sizes, use a structure or cell array.

For example, create a cell array that contains 10 elements, where each element is a numeric array:

```
numArrays = 10;
A = cell(numArrays,1);
for n = 1:numArrays
    A{n} = magic(n);
end
```

Access the data in the cell array by indexing with curly braces. For example, display the fifth element of `A`:

```
A{5}
```

```
ans =  
    17    24     1     8    15  
    23     5     7    14    16  
     4     6    13    20    22  
    10    12    19    21     3  
    11    18    25     2     9
```

The assignment statement `A{n} = magic(n)` is more elegant and efficient than this call to `eval`:

```
eval(['A', int2str(n), ' = magic(n)'])    % Not recommended
```

For more information, see:

- “Create a Cell Array” on page 8-3
- “Create a Structure Array” on page 7-2

Files with Sequential Names

Related data files often have a common root name with an integer index, such as `myfile1.mat` through `myfileN.mat`. A common (but not recommended) use of the `eval` function is to construct and pass each file name to a function using command syntax, such as

```
eval(['save myfile',int2str(n),'.mat'])    % Not recommended
```

The best practice is to use function syntax, which allows you to pass variables as inputs. For example:

```
currentFile = 'myfile1.mat';  
save(currentFile)
```

You can construct file names within a loop using the `sprintf` function (which is usually more efficient than `int2str`), and then call the `save` function without `eval`. This code creates 10 files in the current folder:

```
numFiles = 10;  
for n = 1:numFiles  
    randomData = rand(n);
```

```

        currentFile = sprintf('myfile%d.mat',n);
        save(currentFile,'randomData')
    end

```

For more information, see:

- “Command vs. Function Syntax” on page 1-13
- “Processing a Sequence of Files”

Function Names in Variables

A common use of `eval` is to execute a function when the name of the function is in a variable string. There are two ways to evaluate functions from variables that are more efficient than using `eval`:

- Create function handles with the `@` symbol or with the `str2func` function. For example, run a demo function from a list stored in a cell array:

```

demoFile = {@odedemo,@sunspots,@fitdemo};
n = input('Select a demo number (1, 2, or 3): ');
demoFile{n}()

```

- Use the `feval` function. For example, call a plot function (such as `plot`, `bar`, or `pie`) with data that you specify at run time:

```

plotFunction = input('Specify a plotting function: ','s');
data = input('Enter data to plot: ');
feval(plotFunction,data)

```

Field Names in Variables

Access data in a structure with a variable field name by enclosing the expression for the field in parentheses. For example:

```

myData.height = [67, 72, 58];
myData.weight = [140, 205, 90];

fieldName = input('Select data (height or weight): ','s');
dataToUse = myData.(fieldName);

```

If you enter `weight` at the input prompt, then you can find the minimum `weight` value with the following command.

```
min(dataToUse)
```

```
ans =  
    90
```

For an additional example, see “Generate Field Names from Variables” on page 7-11.

Error Handling

The preferred method for error handling in MATLAB is to use a `try/catch` statement. For example:

```
try  
    B = A;  
catch exception  
    disp('A is undefined')  
end
```

If your workspace does not contain variable `A`, then this code returns:

```
A is undefined
```

Previous versions of the documentation for the `eval` function include the syntax `eval(expression,catch_expr)`. If evaluating the `expression` input returns an error, then `eval` evaluates `catch_expr`. However, an explicit `try/catch` is significantly clearer than an implicit catch in an `eval` statement. Using the implicit catch is not recommended.

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 function

- 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 for `garfield.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 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 `gardata` MAT-file described in “Custom Applications to Read and Write MAT-Files” to obtain the results.

Symbol Reference

In this section...
“Asterisk — <code>*</code> ” on page 2-115
“At — <code>@</code> ” on page 2-115
“Colon — <code>:</code> ” on page 2-116
“Comma — <code>,</code> ” on page 2-117
“Curly Braces — <code>{ }</code> ” on page 2-118
“Dot — <code>.</code> ” on page 2-118
“Dot-Dot — <code>..</code> ” on page 2-119
“Dot-Dot-Dot (Ellipsis) — <code>...</code> ” on page 2-119
“Dot-Parentheses — <code>.()</code> ” on page 2-121
“Exclamation Point — <code>!</code> ” on page 2-121
“Parentheses — <code>()</code> ” on page 2-121
“Percent — <code>%</code> ” on page 2-122
“Percent-Brace — <code>%{ %}</code> ” on page 2-123
“Plus — <code>+</code> ” on page 2-123
“Semicolon — <code>;</code> ” on page 2-123
“Single Quotes — <code>' '</code> ” on page 2-124
“Space Character” on page 2-125
“Slash and Backslash — <code>/ \</code> ” on page 2-125
“Square Brackets — <code>[]</code> ” on page 2-126
“Tilde — <code>~</code> ” on page 2-126

This section does not include symbols used in arithmetic, relational, and logical operations. For a description of these symbols, see the top of the Alphabetical List of functions in the MATLAB Help browser.

Asterisk — *

An asterisk in a filename specification is used as a wildcard specifier, as described below.

Filename Wildcard

Wildcards are generally used in file operations that act on multiple files or folders. They usually appear in the string containing the file or folder specification. MATLAB matches all characters in the name exactly except for the wildcard character *, which can match any one or more characters.

To locate all files with names that start with 'january_' and have a mat file extension, use

```
dir('january_*.mat')
```

You can also use wildcards with the `who` and `whos` functions. To get information on all variables with names starting with 'image' and ending with 'Offset', use

```
whos image*Offset
```

At — @

The @ sign signifies either a function handle constructor or a folder that supports a MATLAB class.

Function Handle Constructor

The @ operator forms a handle to either the named function that follows the @ sign, or to the anonymous function that follows the @ sign.

Function Handles in General. Function handles are commonly used in passing functions as arguments to other functions. Construct a function handle by preceding the function name with an @ sign:

```
fhandle = @myfun
```

You can read more about function handles in Chapter 9, “Function Handles”.

Handles to Anonymous Functions. Anonymous functions give you a quick means of creating simple functions without having to create your function in a file each time. You can construct an anonymous function and a handle to that function using the syntax

```
fhandle = @(arglist) body
```

where `body` defines the body of the function and `arglist` is the list of arguments you can pass to the function.

See “Anonymous Functions” on page 15-3 for more information.

Class Folder Designator

An @ sign can indicate the name of a class folder, such as

```
\@myclass\get.m
```

See the documentation on “Options for Class Folders” for more information.

Colon — :

The colon operator generates a sequence of numbers that you can use in creating or indexing into arrays. See “Generating a Numeric Sequence” for more information on using the colon operator.

Numeric Sequence Range

Generate a sequential series of regularly spaced numbers from `first` to `last` using the syntax `first:last`. For an incremental sequence from 6 to 17, use

```
N = 6:17
```

Numeric Sequence Step

Generate a sequential series of numbers, each number separated by a `step` value, using the syntax `first:step:last`. For a sequence from 2 through 38, stepping by 4 between each entry, use

```
N = 2:4:38
```

Indexing Range Specifier

Index into multiple rows or columns of a matrix using the colon operator to specify a range of indices:

```
B = A(7, 1:5);           % Read columns 1-5 of row 7.  
B = A(4:2:8, 1:5);      % Read columns 1-5 of rows 4, 6, and 8.  
B = A(:, 1:5);          % Read columns 1-5 of all rows.
```

Conversion to Column Vector

Convert a matrix or array to a column vector using the colon operator as a single index:

```
A = rand(3,4);  
B = A(:);
```

Preserving Array Shape on Assignment

Using the colon operator on the left side of an assignment statement, you can assign new values to array elements without changing the shape of the array:

```
A = rand(3,4);  
A(:) = 1:12;
```

Comma — ,

A comma is used to separate the following types of elements.

Row Element Separator

When constructing an array, use a comma to separate elements that belong in the same row:

```
A = [5.92, 8.13, 3.53]
```

Array Index Separator

When indexing into an array, use a comma to separate the indices into each dimension:

```
X = A(2, 7, 4)
```

Function Input and Output Separator

When calling a function, use a comma to separate output and input arguments:

```
function [data, text] = xlsread(file, sheet, range, mode)
```

Command or Statement Separator

To enter more than one MATLAB command or statement on the same line, separate each command or statement with a comma:

```
for k = 1:10,    sum(A(k)),    end
```

Curly Braces — { }

Use curly braces to construct or get the contents of cell arrays.

Cell Array Constructor

To construct a cell array, enclose all elements of the array in curly braces:

```
C = {[2.6 4.7 3.9], rand(8)*6, 'C. Coolidge'}
```

Cell Array Indexing

Index to a specific cell array element by enclosing all indices in curly braces:

```
A = C{4,7,2}
```

See the documentation on Cell Arrays for more information.

Dot — .

The single dot operator has the following different uses in MATLAB.

Decimal Point

MATLAB uses a period to separate the integral and fractional parts of a number.

Structure Field Definition

Add fields to a MATLAB structure by following the structure name with a dot and then a field name:

```
funds(5,2).bondtype = 'Corporate';
```

See the documentation on Chapter 7, “Structures” for more information.

Object Method Specifier

Specify the properties of an instance of a MATLAB class using the object name followed by a dot, and then the property name:

```
val = asset.current_value
```

See Chapter 13, “Defining Your Own Classes” for more information.

Dot-Dot – ..

Two dots in sequence refer to the parent of the current folder.

Parent Folder

Specify the folder immediately above your current folder using two dots. For example, to go up two levels in the folder tree and down into the `test` folder, use

```
cd ..\..\test
```

Dot-Dot-Dot (Ellipsis) – ...

A series of three consecutive periods (`...`) is the line continuation operator in MATLAB. This is often referred to as an *ellipsis*, but it should be noted that the line continuation operator is a three-character operator and is different from the single-character ellipsis represented in ASCII by the hexadecimal number 2026.

Line Continuation

Continue any MATLAB command or expression by placing an ellipsis at the end of the line to be continued:

```
fprintf('The current value of %s is %d', ...
       vname, value)
```

Entering Long Strings. You cannot use an ellipsis within single quotes to continue a string to the next line:

```
string = 'This is not allowed and will generate an ...
        error in MATLAB.'
```

To enter a string that extends beyond a single line, piece together shorter strings using either the concatenation operator ([]) or the `fprintf` function.

Here are two examples:

```
quote1 = [
    'Tiger, tiger, burning bright in the forests of the night,' ...
    'what immortal hand or eye could frame thy fearful symmetry?'];
quote2 = fprintf('%s%s%s', ...
    'In Xanadu did Kubla Khan a stately pleasure-dome decree,', ...
    'where Alph, the sacred river, ran ', ...
    'through caverns measureless to man down to a sunless sea.');
```

Defining Arrays. MATLAB interprets the ellipsis as a space character. For statements that define arrays or cell arrays within [] or {} operators, a space character separates array elements. For example,

```
not_valid = [1 2 zeros...
            (1,3)]
```

is equivalent to

```
not_valid = [1 2 zeros (1,3)]
```

which returns an error. Place the ellipses so that the interpreted statement is valid, such as

```
valid = [1 2 ...
```

```
zeros(1,3)]
```

Dot-Parentheses — .()

Use dot-parentheses to specify the name of a dynamic structure field.

Dynamic Structure Fields

Sometimes it is useful to reference structures with field names that can vary. For example, the referenced field might be passed as an argument to a function. Dynamic field names specify a variable name for a structure field.

The variable `fundtype` shown here is a dynamic field name:

```
type = funds(5,2).(fundtype);
```

See “Generate Field Names from Variables” on page 7-11 for more information.

Exclamation Point — !

The exclamation point precedes operating system commands that you want to execute from within MATLAB.

Shell Escape

The exclamation point initiates a shell escape function. Such a function is to be performed directly by the operating system:

```
!rmdir oldtests
```

See “Shell Escape Functions” on page 2-113 for more information.

Parentheses — ()

Parentheses are used mostly for indexing into elements of an array or for specifying arguments passed to a called function. Parenthesis also control the order of operations, and can group a vector visually (such as `x = (1:10)`) without calling a concatenation function.

Array Indexing

When parentheses appear to the right of a variable name, they are indices into the array stored in that variable:

```
A(2, 7, 4)
```

Function Input Arguments

When parentheses follow a function name in a function declaration or call, the enclosed list contains input arguments used by the function:

```
function sendmail(to, subject, message, attachments)
```

Percent — %

The percent sign is most commonly used to indicate nonexecutable text within the body of a program. This text is normally used to include comments in your code. Two percent signs, `%%`, serve as a cell delimiter described in “Evaluate Subsections of Files Using Code Cells”. Some functions also interpret the percent sign as a conversion specifier.

Single Line Comments

Precede any one-line comments in your code with a percent sign. MATLAB does not execute anything that follows a percent sign (that is, unless the sign is quoted, `'%'`):

```
% The purpose of this routine is to compute  
% the value of ...
```

See “Help Text” on page 14-13 for more information.

Conversion Specifiers

Some functions, like `sscanf` and `sprintf`, precede conversion specifiers with the percent sign:

```
sprintf('%s = %d', name, value)
```


Percent-Brace — %{ %}

The `%{` and `%}` symbols enclose a block of comments that extend beyond one line.

Block Comments

Enclose any multiline comments with percent followed by an opening or closing brace.

```
%{  
The purpose of this routine is to compute  
the value of ...  
%}
```

Note With the exception of whitespace characters, the `%{` and `%}` operators must appear alone on the lines that immediately precede and follow the block of help text. Do not include any other text on these lines.

Plus — +

The `+` sign appears most frequently as an arithmetic operator, but is also used to designate the names of package folders. For more information, see “Create a Namespace with Packages”.

Semicolon — ;

The semicolon can be used to construct arrays, suppress output from a MATLAB command, or to separate commands entered on the same line.

Array Row Separator

When used within square brackets to create a new array or concatenate existing arrays, the semicolon creates a new row in the array:

```
A = [5, 8; 3, 4]  
A =  
    5    8  
    3    4
```

Output Suppression

When placed at the end of a command, the semicolon tells MATLAB not to display any output from that command. In this example, MATLAB does not display the resulting 100-by-100 matrix:

```
A = ones(100, 100);
```

Command or Statement Separator

Like the comma operator, you can enter more than one MATLAB command on a line by separating each command with a semicolon. MATLAB suppresses output for those commands terminated with a semicolon, and displays the output for commands terminated with a comma.

In this example, assignments to variables **A** and **C** are terminated with a semicolon, and thus do not display. Because the assignment to **B** is comma-terminated, the output of this one command is displayed:

```
A = 12.5; B = 42.7, C = 1.25;  
B =  
    42.7000
```

Single Quotes – ' '

Single quotes are the constructor symbol for MATLAB character arrays.

Character and String Constructor

MATLAB constructs a character array from all characters enclosed in single quotes. If only one character is in quotes, then MATLAB constructs a 1-by-1 array:

```
S = 'Hello World'
```

See Chapter 6, “Characters and Strings” for more information.

Space Character

The space character serves a purpose similar to the comma in that it can be used to separate row elements in an array constructor, or the values returned by a function.

Row Element Separator

You have the option of using either commas or spaces to delimit the row elements of an array when constructing the array. To create a 1-by-3 array, use

```
A = [5.92 8.13 3.53]
A =
    5.9200    8.1300    3.5300
```

When indexing into an array, you must always use commas to reference each dimension of the array.

Function Output Separator

Spaces are allowed when specifying a list of values to be returned by a function. You can use spaces to separate return values in both function declarations and function calls:

```
function [data text] = xlsread(file, sheet, range, mode)
```

Slash and Backslash – / \

The slash (/) and backslash (\) characters separate the elements of a path or folder string. On Microsoft Windows-based systems, both slash and backslash have the same effect. On The Open Group UNIX-based systems, you must use slash only.

On a Windows system, you can use either backslash or slash:

```
dir([matlabroot '\toolbox\matlab\elmat\shiftdim.m'])
dir([matlabroot '/toolbox/matlab/elmat/shiftdim.m'])
```

On a UNIX system, use only the forward slash:

```
dir([matlabroot '/toolbox/matlab/elmat/shiftdim.m'])
```

Square Brackets – []

Square brackets are used in array construction and concatenation, and also in declaring and capturing values returned by a function.

Array Constructor

To construct a matrix or array, enclose all elements of the array in square brackets:

```
A = [5.7, 9.8, 7.3; 9.2, 4.5, 6.4]
```

Concatenation

To combine two or more arrays into a new array through concatenation, enclose all array elements in square brackets:

```
A = [B, eye(6), diag([0:2:10])]
```

Function Declarations and Calls

When declaring or calling a function that returns more than one output, enclose each return value that you need in square brackets:

```
[data, text] = xlsread(file, sheet, range, mode)
```

Tilde – ~

The tilde character is used in comparing arrays for unequal values, finding the logical NOT of an array, and as a placeholder for an input or output argument you want to omit from a function call.

Not Equal to

To test for inequality values of elements in arrays a and b for inequality, use `a~=b`:

```
a = primes(29);    b = [2 4 6 7 11 13 20 22 23 29];  
not_prime = b(a~=b)  
not_prime =
```

4 6 20 22

Logical NOT

To find those elements of an array that are zero, use:

```
a = [35 42 0 18 0 0 0 16 34 0];  
~a  
ans =  
    0    0    1    0    1    1    1    0    0    1
```

Argument Placeholder

To have the `fileparts` function return its third output value and skip the first two, replace arguments one and two with a tilde character:

```
[~, ~, filenameExt] = fileparts(fileSpec);
```

See “Ignore Function Outputs” on page 1-9 in the MATLAB Programming documentation for more information.

Classes (Data Types)

- Chapter 3, “Overview of MATLAB Classes”
- Chapter 4, “Numeric Classes”
- Chapter 5, “The Logical Class”
- Chapter 6, “Characters and Strings”
- Chapter 7, “Structures”
- Chapter 8, “Cell Arrays”
- Chapter 9, “Function Handles”
- Chapter 10, “Map Containers”
- Chapter 11, “Combining Unlike Classes”
- Chapter 12, “Using Objects”
- Chapter 13, “Defining Your Own Classes”

Overview of MATLAB Classes

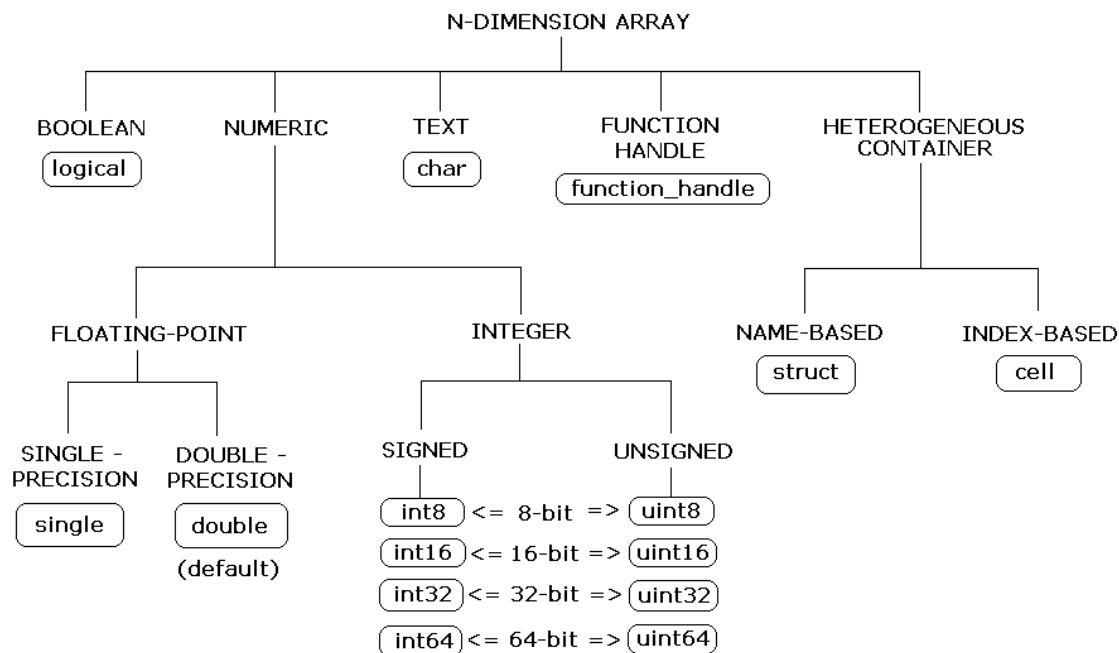
- “Fundamental MATLAB Classes” on page 3-2
- “How to Use the Different Classes” on page 3-4

Fundamental MATLAB Classes

There are many different data types, or *classes*, that you can work with in the MATLAB software. You can build matrices and arrays of floating-point and integer data, characters and strings, and logical `true` and `false` states. Function handles connect your code with any MATLAB function regardless of the current scope. Structures and cell arrays, provide a way to store dissimilar types of data in the same array.

There are 15 fundamental classes in MATLAB. Each of these classes is in the form of a matrix or array. With the exception of function handles, this matrix or array is a minimum of 0-by-0 in size and can grow to an n-dimensional array of any size. A function handle is always scalar (1-by-1).

All of the fundamental MATLAB classes are circled in the diagram below:



Numeric classes in the MATLAB software include signed and unsigned integers, and single- and double-precision floating-point numbers. By default, MATLAB stores all numeric values as double-precision floating point. (You cannot change the default type and precision.) You can choose to store any number, or array of numbers, as integers or as single-precision. Integer and single-precision arrays offer more memory-efficient storage than double-precision.

All numeric types support basic array operations, such as subscripting, reshaping, and mathematical operations.

You can create two-dimensional `double` and `logical` matrices using one of two storage formats: `full` or `sparse`. For matrices with mostly zero-valued elements, a sparse matrix requires a fraction of the storage space required for an equivalent full matrix. Sparse matrices invoke methods especially tailored to solve sparse problems

These classes require different amounts of storage, the smallest being a `logical` value or 8-bit integer which requires only 1 byte. It is important to keep this minimum size in mind if you work on data in files that were written using a precision smaller than 8 bits.

How to Use the Different Classes

The following table describes these classes in more detail.

Class Name	Documentation	Intended Use
double, single	Floating-Point Numbers	<ul style="list-style-type: none"> • Required for fractional numeric data. • Double and Single precision. • Use <code>realmin</code> and <code>realmax</code> to show range of values. • Two-dimensional arrays can be sparse. • Default numeric type in MATLAB.
int8, uint8, int16, uint16, int32, uint32, int64, uint64	Integers	<ul style="list-style-type: none"> • Use for signed and unsigned whole numbers. • More efficient use of memory. • Use <code>intmin</code> and <code>intmax</code> to show range of values. • Choose from 4 sizes (8, 16, 32, and 64 bits).
char	Chapter 6, “Characters and Strings”	<ul style="list-style-type: none"> • Required for text. • Native or unicode. • Converts to/from numeric. • Use with regular expressions. • For multiple strings, use cell arrays.
logical	Logical Class	<ul style="list-style-type: none"> • Use in relational conditions or to test state. • Can have one of two values: <code>true</code> or <code>false</code>. • Also useful in array indexing. • Two-dimensional arrays can be sparse.

Class Name	Documentation	Intended Use
function_handle	Chapter 9, “Function Handles”	<ul style="list-style-type: none"> • Pointer to a function. • Enables passing a function to another function • Can also call functions outside usual scope. • Useful in Handle Graphics callbacks. • Save to MAT-file and restore later.
struct	Structures	<ul style="list-style-type: none"> • Fields store arrays of varying classes and sizes. • Access multiple fields/indices in single operation. • Field names identify contents. • Simple method of passing function arguments. • Use in comma-separated lists for efficiency. • More memory required for overhead
cell	Cell Arrays	<ul style="list-style-type: none"> • Cells store arrays of varying classes and sizes. • Allows freedom to package data as you want. • Manipulation of elements is similar to arrays. • Simple method of passing function arguments. • Use in comma-separated lists for efficiency. • More memory required for overhead

Numeric Classes

- “Overview of Numeric Classes” on page 4-2
- “Integers” on page 4-3
- “Floating-Point Numbers” on page 4-7
- “Complex Numbers” on page 4-18
- “Infinity and NaN” on page 4-20
- “Identifying Numeric Classes” on page 4-22
- “Display Format for Numeric Values” on page 4-23
- “Function Summary” on page 4-26

Overview of Numeric Classes

Numeric classes in the MATLAB software include signed and unsigned integers, and single- and double-precision floating-point numbers. By default, MATLAB stores all numeric values as double-precision floating point. (You cannot change the default type and precision.) You can choose to store any number, or array of numbers, as integers or as single-precision. Integer and single-precision arrays offer more memory-efficient storage than double-precision.

All numeric types support basic array operations, such as subscripting, reshaping, and mathematical operations.

Integers

In this section...

“Integer Classes” on page 4-3

“Creating Integer Data” on page 4-4

“Arithmetic Operations on Integer Classes” on page 4-5

“Largest and Smallest Values for Integer Classes” on page 4-6

“Integer Functions” on page 4-6

Integer Classes

MATLAB has four signed and four unsigned integer classes. Signed types enable you to work with negative integers as well as positive, but cannot represent as wide a range of numbers as the unsigned types because one bit is used to designate a positive or negative sign for the number. Unsigned types give you a wider range of numbers, but these numbers can only be zero or positive.

MATLAB supports 1-, 2-, 4-, and 8-byte storage for integer data. You can save memory and execution time for your programs if you use the smallest integer type that accommodates your data. For example, you do not need a 32-bit integer to store the value 100.

Here are the eight integer classes, the range of values you can store with each type, and the MATLAB conversion function required to create that type:

Class	Range of Values	Conversion Function
Signed 8-bit integer	-2^7 to 2^7-1	int8
Signed 16-bit integer	-2^{15} to $2^{15}-1$	int16
Signed 32-bit integer	-2^{31} to $2^{31}-1$	int32
Signed 64-bit integer	-2^{63} to $2^{63}-1$	int64
Unsigned 8-bit integer	0 to 2^8-1	uint8
Unsigned 16-bit integer	0 to $2^{16}-1$	uint16

Class	Range of Values	Conversion Function
Unsigned 32-bit integer	0 to $2^{32}-1$	uint32
Unsigned 64-bit integer	0 to $2^{64}-1$	uint64

Creating Integer Data

MATLAB stores numeric data as double-precision floating point (`double`) by default. To store data as an integer, you need to convert from `double` to the desired integer type. Use one of the conversion functions shown in the table above.

For example, to store 325 as a 16-bit signed integer assigned to variable `x`, type

```
x = int16(325);
```

If the number being converted to an integer has a fractional part, MATLAB rounds to the nearest integer. If the fractional part is exactly 0.5, then from the two equally nearby integers, MATLAB chooses the one for which the absolute value is larger in magnitude:

```
x = 325.499;          x = x + .001;

int16(x)              int16(x)
ans =                  ans =
    325                  326
```

If you need to round a number using a rounding scheme other than the default, MATLAB provides four rounding functions: `round`, `fix`, `floor`, and `ceil`. The `fix` function enables you to override the default and round *towards zero* when there is a nonzero fractional part:

```
x = 325.9;

int16(fix(x))
ans =
    325
```

Arithmetic operations that involve both integers and floating-point always result in an integer data type. MATLAB rounds the result, when necessary,

according to the default rounding algorithm. The example below yields an exact answer of 1426.75 which MATLAB then rounds to the next highest integer:

```
int16(325) * 4.39
ans =
    1427
```

The integer conversion functions are also useful when converting other classes, such as strings, to integers:

```
str = 'Hello World';

int8(str)
ans =
    72  101  108  108  111  32  87  111  114  108  100
```

Arithmetic Operations on Integer Classes

MATLAB can perform integer arithmetic on the following types of data:

- Integers or integer arrays of the same integer data type. This yields a result that has the same data type as the operands:

```
x = uint32([132 347 528]) .* uint32(75);
class(x)
ans =
    uint32
```

- Integers or integer arrays and scalar double-precision floating-point numbers. This yields a result that has the same data type as the integer operands:

```
x = uint32([132 347 528]) .* 75.49;
class(x)
ans =
    uint32
```

For all binary operations in which one operand is an array of integer data type (except 64-bit integers) and the other is a scalar double, MATLAB computes the operation using elementwise double-precision arithmetic, and then converts the result back to the original integer data type. For binary

operations involving a 64-bit integer array and a scalar double, MATLAB computes the operation as if 80-bit extended-precision arithmetic were used, to prevent loss of precision.

For a list of the operations that support integer classes, see Nondouble Data Type Support in the arithmetic operators reference page.

Largest and Smallest Values for Integer Classes

For each integer data type, there is a largest and smallest number that you can represent with that type. The table shown under “Integers” on page 4-3 lists the largest and smallest values for each integer data type in the “Range of Values” column.

You can also obtain these values with the `intmax` and `intmin` functions:

```
intmax('int8')           intmin('int8')
ans =                    ans =
    127                  -128
```

If you convert a number that is larger than the maximum value of an integer data type to that type, MATLAB sets it to the maximum value. Similarly, if you convert a number that is smaller than the minimum value of the integer data type, MATLAB sets it to the minimum value. For example,

```
x = int8(300)           x = int8(-300)
x =                    x =
    127                  -128
```

Also, when the result of an arithmetic operation involving integers exceeds the maximum (or minimum) value of the data type, MATLAB sets it to the maximum (or minimum) value:

```
x = int8(100) * 3       x = int8(-100) * 3
x =                    x =
    127                  -128
```

Integer Functions

See Integer Functions on page 4-26 for a list of functions most commonly used with integers in MATLAB.

Floating-Point Numbers

In this section...
“Double-Precision Floating Point” on page 4-7
“Single-Precision Floating Point” on page 4-8
“Creating Floating-Point Data” on page 4-8
“Arithmetic Operations on Floating-Point Numbers” on page 4-10
“Largest and Smallest Values for Floating-Point Classes” on page 4-11
“Accuracy of Floating-Point Data” on page 4-12
“Avoiding Common Problems with Floating-Point Arithmetic” on page 4-14
“Floating-Point Functions” on page 4-16
“References” on page 4-16

MATLAB represents floating-point numbers in either double-precision or single-precision format. The default is double precision, but you can make any number single precision with a simple conversion function.

Double-Precision Floating Point

MATLAB constructs the double-precision (or `double`) data type according to IEEE® Standard 754 for double precision. Any value stored as a `double` requires 64 bits, formatted as shown in the table below:

Bits	Usage
63	Sign (0 = positive, 1 = negative)
62 to 52	Exponent, biased by 1023
51 to 0	Fraction <code>f</code> of the number <code>1.f</code>

Single-Precision Floating Point

MATLAB constructs the single-precision (or `single`) data type according to IEEE Standard 754 for single precision. Any value stored as a `single` requires 32 bits, formatted as shown in the table below:

Bits	Usage
31	Sign (0 = positive, 1 = negative)
30 to 23	Exponent, biased by 127
22 to 0	Fraction f of the number $1.f$

Because MATLAB stores numbers of type `single` using 32 bits, they require less memory than numbers of type `double`, which use 64 bits. However, because they are stored with fewer bits, numbers of type `single` are represented to less precision than numbers of type `double`.

Creating Floating-Point Data

Use double-precision to store values greater than approximately 3.4×10^{38} or less than approximately -3.4×10^{38} . For numbers that lie between these two limits, you can use either double- or single-precision, but single requires less memory.

Creating Double-Precision Data

Because the default numeric type for MATLAB is `double`, you can create a `double` with a simple assignment statement:

```
x = 25.783;
```

The `whos` function shows that MATLAB has created a 1-by-1 array of type `double` for the value you just stored in `x`:

```
whos x
  Name      Size      Bytes  Class
  x         1x1         8      double
```

Use `isfloat` if you just want to verify that `x` is a floating-point number. This function returns logical 1 (true) if the input is a floating-point number, and logical 0 (false) otherwise:

```
isfloat(x)
ans =
     1
```

You can convert other numeric data, characters or strings, and logical data to double precision using the MATLAB function, `double`. This example converts a signed integer to double-precision floating point:

```
y = int64(-589324077574);           % Create a 64-bit integer

x = double(y)                       % Convert to double
x =
    -5.8932e+11
```

Creating Single-Precision Data

Because MATLAB stores numeric data as a `double` by default, you need to use the `single` conversion function to create a single-precision number:

```
x = single(25.783);
```

The `whos` function returns the attributes of variable `x` in a structure. The `bytes` field of this structure shows that when `x` is stored as a single, it requires just 4 bytes compared with the 8 bytes to store it as a `double`:

```
xAttrib = whos('x');
xAttrib.bytes
ans =
     4
```

You can convert other numeric data, characters or strings, and logical data to single precision using the `single` function. This example converts a signed integer to single-precision floating point:

```
y = int64(-589324077574);           % Create a 64-bit integer

x = single(y)                       % Convert to single
```

```
x =  
-5.8932e+11
```

Arithmetic Operations on Floating-Point Numbers

This section describes which classes you can use in arithmetic operations with floating-point numbers.

Double-Precision Operations

You can perform basic arithmetic operations with `double` and any of the following other classes. When one or more operands is an integer (scalar or array), the `double` operand must be a scalar. The result is of type `double`, except where noted otherwise:

- `single` — The result is of type `single`
- `double`
- `int*` or `uint*` — The result has the same data type as the integer operand
- `char`
- `logical`

This example performs arithmetic on data of types `char` and `double`. The result is of type `double`:

```
c = 'uppercase' - 32;
```

```
class(c)  
ans =  
    double
```

```
char(c)  
ans =  
    UPPERCASE
```

Single-Precision Operations

You can perform basic arithmetic operations with `single` and any of the following other classes. The result is always `single`:

- single
- double
- char
- logical

In this example, 7.5 defaults to type double, and the result is of type single:

```
x = single([1.32 3.47 5.28]) .* 7.5;

class(x)
ans =
    single
```

Largest and Smallest Values for Floating-Point Classes

For the double and single classes, there is a largest and smallest number that you can represent with that type.

Largest and Smallest Double-Precision Values

The MATLAB functions `realmax` and `realmin` return the maximum and minimum values that you can represent with the double data type:

```
str = 'The range for double is:\n\t%g to %g and\n\t %g to %g';
sprintf(str, -realmax, -realmin, realmin, realmax)

ans =
The range for double is:
-1.79769e+308 to -2.22507e-308 and
 2.22507e-308 to 1.79769e+308
```

Numbers larger than `realmax` or smaller than `-realmax` are assigned the values of positive and negative infinity, respectively:

```
realmax + .0001e+308
ans =
    Inf

-realmax - .0001e+308
```

```
ans =  
    -Inf
```

Largest and Smallest Single-Precision Values

The MATLAB functions `realmax` and `realmin`, when called with the argument `'single'`, return the maximum and minimum values that you can represent with the `single` data type:

```
str = 'The range for single is:\n\t%g to %g and\n\t %g to %g';  
sprintf(str, -realmax('single'), -realmin('single'), ...  
        realmin('single'), realmax('single'))
```

```
ans =  
The range for single is:  
-3.40282e+38 to -1.17549e-38 and  
 1.17549e-38 to  3.40282e+38
```

Numbers larger than `realmax('single')` or smaller than `-realmax('single')` are assigned the values of positive and negative infinity, respectively:

```
realmax('single') + .0001e+038
```

```
ans =  
    Inf
```

```
-realmax('single') - .0001e+038
```

```
ans =  
   -Inf
```

Accuracy of Floating-Point Data

If the result of a floating-point arithmetic computation is not as precise as you had expected, it is likely caused by the limitations of your computer's hardware. Probably, your result was a little less exact because the hardware had insufficient bits to represent the result with perfect accuracy; therefore, it truncated the resulting value.

Double-Precision Accuracy

Because there are only a finite number of double-precision numbers, you cannot represent all numbers in double-precision storage. On any computer, there is a small gap between each double-precision number and the next larger double-precision number. You can determine the size of this gap, which limits the precision of your results, using the `eps` function. For example, to find the distance between 5 and the next larger double-precision number, enter

```
format long

eps(5)
ans =
    8.881784197001252e-16
```

This tells you that there are no double-precision numbers between 5 and $5 + \text{eps}(5)$. If a double-precision computation returns the answer 5, the result is only accurate to within $\text{eps}(5)$.

The value of $\text{eps}(x)$ depends on x . This example shows that, as x gets larger, so does $\text{eps}(x)$:

```
eps(50)
ans =
    7.105427357601002e-15
```

If you enter `eps` with no input argument, MATLAB returns the value of $\text{eps}(1)$, the distance from 1 to the next larger double-precision number.

Single-Precision Accuracy

Similarly, there are gaps between any two single-precision numbers. If x has type `single`, $\text{eps}(x)$ returns the distance between x and the next larger single-precision number. For example,

```
x = single(5);
eps(x)

returns

ans =
    4.7684e-07
```

Note that this result is larger than `eps(5)`. Because there are fewer single-precision numbers than double-precision numbers, the gaps between the single-precision numbers are larger than the gaps between double-precision numbers. This means that results in single-precision arithmetic are less precise than in double-precision arithmetic.

For a number `x` of type `double`, `eps(single(x))` gives you an upper bound for the amount that `x` is rounded when you convert it from `double` to `single`. For example, when you convert the double-precision number `3.14` to `single`, it is rounded by

```
double(single(3.14) - 3.14)
ans =
    1.0490e-07
```

The amount that `3.14` is rounded is less than

```
eps(single(3.14))
ans =
    2.3842e-07
```

Avoiding Common Problems with Floating-Point Arithmetic

Almost all operations in MATLAB are performed in double-precision arithmetic conforming to the IEEE standard 754. Because computers only represent numbers to a finite precision (double precision calls for 52 mantissa bits), computations sometimes yield mathematically nonintuitive results. It is important to note that these results are not bugs in MATLAB.

Use the following examples to help you identify these cases:

Example 1 — Round-Off or What You Get Is Not What You Expect

The decimal number $4/3$ is not exactly representable as a binary fraction. For this reason, the following calculation does not give zero, but rather reveals the quantity `eps`.

```
e = 1 - 3*(4/3 - 1)
```

```
e =
  2.2204e-16
```

Similarly, 0.1 is not exactly representable as a binary number. Thus, you get the following nonintuitive behavior:

```
a = 0.0;
for i = 1:10
  a = a + 0.1;
end
a == 1
```

```
ans =
  0
```

Note that the order of operations can matter in the computation:

```
b = 1e-16 + 1 - 1e-16;
c = 1e-16 - 1e-16 + 1;
b == c
```

```
ans =
  0
```

There are gaps between floating-point numbers. As the numbers get larger, so do the gaps, as evidenced by:

```
(2^53 + 1) - 2^53
```

```
ans =
  0
```

Since π is not really π , it is not surprising that $\sin(\pi)$ is not exactly zero:

```
sin(pi)
```

```
ans =
  1.224646799147353e-16
```

Example 2 – Catastrophic Cancellation

When subtractions are performed with nearly equal operands, sometimes cancellation can occur unexpectedly. The following is an example of a cancellation caused by swamping (loss of precision that makes the addition insignificant).

```
sqrt(1e-16 + 1) - 1
```

```
ans =  
    0
```

Some functions in MATLAB, such as `expm1` and `log1p`, may be used to compensate for the effects of catastrophic cancellation.

Example 3 – Floating-Point Operations and Linear Algebra

Round-off, cancellation, and other traits of floating-point arithmetic combine to produce startling computations when solving the problems of linear algebra. MATLAB warns that the following matrix `A` is ill-conditioned, and therefore the system $Ax = b$ may be sensitive to small perturbations:

```
A = diag([2 eps]);  
b = [2; eps];  
y = A\b;  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.110223e-16.
```

These are only a few of the examples showing how IEEE floating-point arithmetic affects computations in MATLAB. Note that all computations performed in IEEE 754 arithmetic are affected, this includes applications written in C or FORTRAN, as well as MATLAB.

Floating-Point Functions

See [Floating-Point Functions](#) on page 4-27 for a list of functions most commonly used with floating-point numbers in MATLAB.

References

The following references provide more information about floating-point arithmetic.

[1] Moler, Cleve, "Floating Points," *MATLAB News and Notes*, Fall, 1996. A PDF version is available on the MathWorks Web site at http://www.mathworks.com/company/newsletters/news_notes/pdf/Fall96Cleve.pdf

[2] Moler, Cleve, *Numerical Computing with MATLAB*, S.I.A.M. A PDF version is available on the MathWorks Web site at <http://www.mathworks.com/moler/>.

Complex Numbers

In this section...

“Creating Complex Numbers” on page 4-18

“Complex Number Functions” on page 4-19

Creating Complex Numbers

Complex numbers consist of two separate parts: a real part and an imaginary part. The basic imaginary unit is equal to the square root of -1. This is represented in MATLAB by either of two letters: `i` or `j`.

The following statement shows one way of creating a complex value in MATLAB. The variable `x` is assigned a complex number with a real part of 2 and an imaginary part of 3:

```
x = 2 + 3i;
```

Another way to create a complex number is using the `complex` function. This function combines two numeric inputs into a complex output, making the first input real and the second imaginary:

```
x = rand(3) * 5;
y = rand(3) * -8;
```

```
z = complex(x, y)
```

```
z =
    4.7842 -1.0921i    0.8648 -1.5931i    1.2616 -2.2753i
    2.6130 -0.0941i    4.8987 -2.3898i    4.3787 -3.7538i
    4.4007 -7.1512i    1.3572 -5.2915i    3.6865 -0.5182i
```

You can separate a complex number into its real and imaginary parts using the `real` and `imag` functions:

```
zr = real(z)
```

```
zr =
    4.7842    0.8648    1.2616
    2.6130    4.8987    4.3787
    4.4007    1.3572    3.6865
```



```
zi = imag(z)
zi =
    -1.0921    -1.5931    -2.2753
    -0.0941    -2.3898    -3.7538
    -7.1512    -5.2915    -0.5182
```

Complex Number Functions

See Complex Number Functions on page 4-27 for a list of functions most commonly used with MATLAB complex numbers in MATLAB.

Infinity and NaN

In this section...

“Infinity” on page 4-20

“NaN” on page 4-20

“Infinity and NaN Functions” on page 4-21

Infinity

MATLAB represents infinity by the special value `inf`. Infinity results from operations like division by zero and overflow, which lead to results too large to represent as conventional floating-point values. MATLAB also provides a function called `inf` that returns the IEEE arithmetic representation for positive infinity as a `double` scalar value.

Several examples of statements that return positive or negative infinity in MATLAB are shown here.

<pre>x = 1/0 x = Inf</pre>	<pre>x = 1.e1000 x = Inf</pre>
<pre>x = exp(1000) x = Inf</pre>	<pre>x = log(0) x = -Inf</pre>

Use the `isinf` function to verify that `x` is positive or negative infinity:

```
x = log(0);
```

```
isinf(x)
ans =
    1
```

NaN

MATLAB represents values that are not real or complex numbers with a special value called NaN, which stands for Not a Number. Expressions like `0/0` and `inf/inf` result in NaN, as do any arithmetic operations involving a NaN:

```
x = 0/0
x =
    NaN
```

Use the `isnan` function to verify that the real part of `x` is NaN:

```
isnan(x)
ans =
     1
```

MATLAB also provides a function called `NaN` that returns the IEEE arithmetic representation for NaN as a `double` scalar value:

```
x = NaN;

whos x
      Name      Size      Bytes  Class
      x         1x1         8      double
```

Logical Operations on NaN

Because two NaNs are not equal to each other, logical operations involving NaN always return false, except for a test for inequality, (`NaN ~= NaN`):

```
NaN > NaN
ans =
     0
```

```
NaN ~= NaN
ans =
     1
```

Infinity and NaN Functions

See Infinity and NaN Functions on page 4-28 for a list of functions most commonly used with `inf` and `NaN` in MATLAB.

Identifying Numeric Classes

You can check the data type of a variable `x` using any of these commands.

Command	Operation
<code>whos x</code>	Display the data type of <code>x</code> .
<code>xType = class(x);</code>	Assign the data type of <code>x</code> to a variable.
<code>isnumeric(x)</code>	Determine if <code>x</code> is a numeric type.
<code>isa(x, 'integer')</code> <code>isa(x, 'uint64')</code> <code>isa(x, 'float')</code> <code>isa(x, 'double')</code> <code>isa(x, 'single')</code>	Determine if <code>x</code> is the specified numeric type. (Examples for any integer, unsigned 64-bit integer, any floating point, double precision, and single precision are shown here).
<code>isreal(x)</code>	Determine if <code>x</code> is real or complex.
<code>isnan(x)</code>	Determine if <code>x</code> is Not a Number (NaN).
<code>isinf(x)</code>	Determine if <code>x</code> is infinite.
<code>isfinite(x)</code>	Determine if <code>x</code> is finite.

Display Format for Numeric Values

In this section...

“Default Display” on page 4-23

“Display Format Examples” on page 4-23

“Setting Numeric Format in a Program” on page 4-24

Default Display

By default, MATLAB displays numeric output as 5-digit scaled, fixed-point values. You can change the way numeric values are displayed to any of the following:

- 5-digit scaled fixed point, floating point, or the best of the two
- 15-digit scaled fixed point, floating point, or the best of the two
- A ratio of small integers
- Hexadecimal (base 16)
- Bank notation

All available formats are listed on the format reference page.

To change the numeric display setting, use either the `format` function or the **Preferences** dialog box (accessible from the MATLAB **File** menu). The `format` function changes the display of numeric values for the duration of a single MATLAB session, while your Preferences settings remain active from one session to the next. These settings affect only how numbers are displayed, not how MATLAB computes or saves them.

Display Format Examples

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.

Check the current format setting:

```
get(0, 'format')
```

```
ans =  
short
```

Set the value for x and display in 5-digit scaled fixed point:

```
x = [4/3 1.2345e-6]  
x =  
1.3333    0.0000
```

Set the format to 5-digit floating point:

```
format short e  
x  
x =  
1.3333e+00    1.2345e-06
```

Set the format to 15-digit scaled fixed point:

```
format long  
x  
x =  
1.333333333333333    0.000001234500000
```

Set the format to 'rational' for small integer ratio output:

```
format rational  
x  
x =  
4/3          1/810045
```

Set an integer value for x and display it in hexadecimal (base 16) format:

```
format hex  
x = uint32(876543210)  
x =  
343efcea
```

Setting Numeric Format in a Program

To temporarily change the numeric format inside a program, get the original format using the `get` function and save it in a variable. When you finish

working with the new format, you can restore the original format setting using the `set` function as shown here:

```
origFormat = get(0, 'format');  
format('rational');  
  
-- Work in rational format --  
  
set(0,'format', origFormat);
```

Function Summary

MATLAB provides these functions for working with numeric classes:

- Integer Functions on page 4-26
- Floating-Point Functions on page 4-27
- Complex Number Functions on page 4-27
- Infinity and NaN Functions on page 4-28
- Class Identification Functions on page 4-28
- Output Formatting Functions on page 4-28

Integer Functions

Function	Description
int8, int16, int32, int64	Convert to signed 1-, 2-, 4-, or 8-byte integer.
uint8, uint16, uint32, uint64	Convert to unsigned 1-, 2-, 4-, or 8-byte integer.
ceil	Round towards plus infinity to nearest integer
class	Return the data type of an object.
fix	Round towards zero to nearest integer
floor	Round towards minus infinity to nearest integer
isa	Determine if input value has the specified data type.
isinteger	Determine if input value is an integer array.
isnumeric	Determine if input value is a numeric array.
round	Round towards the nearest integer

Floating-Point Functions

Function	Description
double	Convert to double precision.
single	Convert to single precision.
class	Return the data type of an object.
isa	Determine if input value has the specified data type.
isfloat	Determine if input value is a floating-point array.
isnumeric	Determine if input value is a numeric array.
eps	Return the floating-point relative accuracy. This value is the tolerance MATLAB uses in its calculations.
realmax	Return the largest floating-point number your computer can represent.
realmin	Return the smallest floating-point number your computer can represent.

Complex Number Functions

Function	Description
complex	Construct complex data from real and imaginary components.
i or j	Return the imaginary unit used in constructing complex data.
real	Return the real part of a complex number.
imag	Return the imaginary part of a complex number.
isreal	Determine if a number is real or imaginary.

Infinity and NaN Functions

Function	Description
inf	Return the IEEE value for infinity.
isnan	Detect NaN elements of an array.
isinf	Detect infinite elements of an array.
isfinite	Detect finite elements of an array.
nan	Return the IEEE value for Not a Number.

Class Identification Functions

Function	Description
class	Return data type (or class).
isa	Determine if input value is of the specified data type.
isfloat	Determine if input value is a floating-point array.
isinteger	Determine if input value is an integer array.
isnumeric	Determine if input value is a numeric array.
isreal	Determine if input value is real.
whos	Display the data type of input.

Output Formatting Functions

Function	Description
format	Control display format for output.

The Logical Class

- “Overview of the Logical Class” on page 5-2
- “Identifying Logical Arrays” on page 5-4
- “Functions that Return a Logical Result” on page 5-6
- “Using Logical Arrays in Conditional Statements” on page 5-9
- “Using Logical Arrays in Indexing” on page 5-10

Overview of the Logical Class

The logical data type represents a logical true or false state using the numbers 1 and 0, respectively. Certain MATLAB functions and operators return logical true or false to indicate whether a certain condition was found to be true or not. For example, the statement `50>40` returns a logical true value.

Logical data does not have to be scalar; MATLAB supports arrays of logical values as well. For example, the following statement returns a vector of logicals indicating false for the first two elements and true for the last three:

```
[30 40 50 60 70] > 40
ans =
     0     0     1     1     1
```

This statement returns a 4-by-4 array of logical values:

```
x = magic(4) >= 9
x =
     1     0     0     1
     0     1     1     0
     1     0     0     1
     0     1     1     0
```

The MATLAB functions that have names beginning with `is` (e.g., `ischar`, `issparse`) also return a logical value or array:

```
a = [2.5 6.7 9.2 inf 4.8];
isfinite(a)
ans =
     1     1     1     0     1
```

Logical arrays can also be sparse as long as they have no more than two dimensions:

```
x = sparse(magic(20) > 395)
x =
 (1,1)      1
 (1,4)      1
```

(1,5)	1
(20,18)	1
(20,19)	1

Identifying Logical Arrays

In this section...

“Function Summary” on page 5-4

“Examples of Identifying Logical Arrays” on page 5-4

Function Summary

This table shows the commands you can use to determine whether or not an array `x` is logical. The last function listed, `cellfun`, operates on cell arrays, which you can read about in the section on cell arrays.

Command	Operation
<code>whos(x)</code>	Display value and data type for <code>x</code> .
<code>islogical(x)</code>	Return true if array is logical.
<code>isa(x, 'logical')</code>	Return true if array is logical.
<code>class(x)</code>	Return string with data type name.
<code>cellfun('islogical', x)</code>	Check each cell array element for logical.

Examples of Identifying Logical Arrays

Create a 3-by-6 array of logicals and use the `whos` function to identify the size, byte count, and class (i.e., data type) of the array.

```
% Initialize the state of the random number generator.
```

```
rand('state',0);
```

```
A = rand(3,6) > .5
```

```
A =
```

```
    1    0    0    0    1    0
```

```
    0    1    0    1    1    1
```

```
    1    1    1    1    0    1
```

```
whos A
```

```
  Name      Size      Bytes  Class      Attributes
```

```
  A         3x6         18  logical
```

Find the class of each of these expressions:

```
B = logical(-2.8); C = false; D = 50>40; E = isinteger(4.9);
```

```
whos B C D E
      Name      Size      Bytes  Class  Attributes
      B          1x1          1  logical
      C          1x1          1  logical
      D          1x1          1  logical
      E          1x1          1  logical
```

Display the class of A:

```
% Initialize the state of the random number generator.
rand('state',0);
A = rand(3,6) > .5

fprintf('A is a %s\n', class(A))
      A is a logical
```

Create cell array C and use `islogical` to identify the logical elements:

```
C = {1, 0, true, false, pi, A};
cellfun('islogical', C)
ans =
      0      0      1      1      0      1
```

Functions that Return a Logical Result

In this section...
“Overview” on page 5-6
“Examples of Functions that Return a Logical Result” on page 5-6

Overview

This table shows some of the MATLAB operations that return a logical true or false. Most mathematics operations are not supported on logical values.

Function	Operation
true, false	Setting value to true or false
logical	Numeric to logical conversion
& (and), (or), ~ (not), xor, any, all	Logical operations
&&,	Short-circuit AND and OR
== (eq), ~= (ne), < (lt), > (gt), <= (le), >= (ge)	Relational operations
All is* functions, cellfun	Test operations
strcmp, strncmp, strcmpi, strncmpi	String comparisons

Examples of Functions that Return a Logical Result

MATLAB functions that test the state of a variable or expression return a logical result:

```
A = isstrprop('abc123def', 'alpha')
A =
     1     1     1     0     0     0     1     1     1
```

Logical functions such as xor return a logical result:

```
xor([1 0 'ab' 2.4], [ 0 0 'ab', 0])
ans =
     1     0     0     1
```


Note however that the bitwise operators do not return a logical:

```
X = bitxor(3, 12);
whos X
  Name      Size      Bytes  Class  Attributes
  X         1x1         8      double
```

String comparison functions also return a logical:

```
S = 'D:\matlab\mfiles\test19.m';
strncmp(S, 'D:\matlab', 9)
ans =
     1
```

Note the difference between the elementwise and short-circuit logical operators. Short-circuit operators, such as `&&` and `||`, test only as much of the input expression as necessary. In the second part of this example, it makes no difference that `B` is undefined because the state of `A` alone determines that the expression is false:

```
A = 0;
A & B
??? Undefined function or variable 'B'.

A && B
ans =
     0
```

One way of implementing an infinite loop is to use the `while` function along with the logical constant `true`:

```
while true
a = []; b = [];
a = input('Enter username: ', 's');

    if ~isempty(a)
        b = input('Enter password: ', 's');
    end

    if ~isempty(b)
```

```
        disp 'Attempting to log in to account ...'  
        break  
    end  
end
```

Using Logical Arrays in Conditional Statements

Conditional statements are useful when you want to execute a block of code only when a certain condition is met. For example, the `sprintf` command shown below is valid only if `str` is a nonempty string:

```
str = input('Enter input string: ', 's');  
if ~isempty(str) && ischar(str)  
    sprintf('Input string is ''%s''', str)  
end
```

Now run the code:

```
Enter input string: Hello  
ans =  
    Input string is 'Hello'
```

Using Logical Arrays in Indexing

A logical matrix provides a different type of array indexing in MATLAB. While most indices are numeric, indicating a certain row or column number, logical indices are positional. That is, it is the *position* of each 1 in the logical matrix that determines which array element is being referred to.

See “Using Logicals in Array Indexing” for more information on this subject.

Characters and Strings

- “Creating Character Arrays” on page 6-2
- “Cell Arrays of Strings” on page 6-7
- “Formatting Strings” on page 6-10
- “String Comparisons” on page 6-25
- “Searching and Replacing” on page 6-28
- “Converting from Numeric to String” on page 6-30
- “Converting from String to Numeric” on page 6-32
- “Function Summary” on page 6-35

Creating Character Arrays

In this section...

“Creating a Character String” on page 6-2

“Creating a Rectangular Character Array” on page 6-3

“Identifying Characters in a String” on page 6-4

“Working with Space Characters” on page 6-5

“Expanding Character Arrays” on page 6-6

Creating a Character String

Create a string by enclosing a sequence of letters in single quotation marks. MATLAB represents the five-character string shown below as a 1-by-5 vector of class `char`. It occupies 2 bytes of memory for each character in the string:

```
str = 'Hello';
```

```
whos str
  Name      Size      Bytes  Class  Attributes
  str       1x5          10   char
```

Functions such as `uint16` convert characters to their numeric codes:

```
str_numeric = uint16(str)
```

```
str_numeric =
    72  101  108  108  111
```

The `char` function converts the integer vector back to characters:

```
str_alpha = char([72 101 108 108 111])
```

```
str_alpha =
    Hello
```

Creating a Rectangular Character Array

You can join two or more strings together to create a new character array. This is called *concatenation* and is explained for numeric arrays in the section “Concatenating Matrices”. As with numeric arrays, you can combine character arrays vertically or horizontally to create a new character array.

Alternatively, combine strings into a cell array. Cell arrays are flexible containers that allow you to easily combine strings of varying length.

Combining Strings Vertically

To combine strings into a two-dimensional character array, use either of these methods:

- Apply the MATLAB concatenation operator, `[]`. Separate each row with a semicolon (`;`). Each row must contain the same number of characters. For example, combine three strings of equal length:

```
dev_title = ['Thomas R. Lee'; ...
            'Sr. Developer'; ...
            'SFTware Corp.'];
```

If the strings have different lengths, pad with space characters as needed. For example:

```
mgr_title = ['Harold A. Jorgensen      '; ...
            'Assistant Project Manager'; ...
            'SFTware Corp.              '];
```

- Call the `char` function. If the strings are different length, `char` pads the shorter strings with trailing blanks so that each row has the same number of characters. For example, combine three strings of different lengths:

```
mgr_title = char('Harold A. Jorgensen', ...
                'Assistant Project Manager', 'SFTware Corp.');
```

The `char` function creates a 3-by-25 character array `mgr_title`.

Combining Strings Horizontally

To combine strings into a single row vector, use either of these methods:

- Apply the MATLAB concatenation operator, `[]`. Separate the input strings with a comma or a space. This method preserves any trailing spaces in the input arrays. For example, combine several strings:

```
name = 'Thomas R. Lee';  
title = 'Sr. Developer';  
company = 'SFTware Corp.';  
  
full_name = [name ' ', ' title ', ' company]
```

MATLAB returns

```
full_name =  
    Thomas R. Lee, Sr. Developer, SFTware Corp.
```

- Call the string concatenation function, `strcat`. This method removes trailing spaces in the inputs. For example, combine strings to create a hypothetical email address:

```
name = 'myname  ';  
domain = 'mydomain  ';  
ext = 'com  ';  
  
address = strcat(name, '@', domain, '.', ext)
```

MATLAB returns

```
address =  
    myname@mydomain.com
```

Identifying Characters in a String

Use any of the following functions to identify a character or string, or certain characters in a string:

Function	Description
<code>ischar</code>	Determine whether the input is a character array.
<code>isletter</code>	Find all alphabetic letters in the input string.
<code>isspace</code>	Find all space characters in the input string.
<code>isstrprop</code>	Find all characters of a specific category.


```

str = 'Find the space characters in this string';
%           |   |           |   |   |
%           5   9   15       26 29   34

find(isspace(str))
ans =
     5     9    15    26    29    34

```

Working with Space Characters

The `blanks` function creates a string of space characters. The following example creates a string of 15 space characters:

```

s = blanks(15)
s =

```

To make the example more useful, append a `|` character to the beginning and end of the blank string so that you can see the output:

```

['|' s '|']           % Make result visible.
ans =
 |                   |

```

Insert a few nonspace characters in the middle of the blank string:

```

s(6:10) = 'AAAAA';

['|' s '|']           % Make result visible.
ans =
 |      AAAAA      |

```

You can justify the positioning of these characters to the left or right using the `strjust` function:

```

sLeft = strjust(s, 'left');

['|' sLeft '|']       % Make result visible.
ans =
 |AAAAA              |

sRight = strjust(s, 'right');

```

```
['|' sRight '|']      % Make result visible.  
ans =  
    |          AAAAA|
```

Remove all trailing space characters with `deblank`:

```
sDeblank = deblank(s);
```

```
['|' sDeblank '|']    % Make result visible.  
ans =  
    |          AAAAA|
```

Remove all leading and trailing spaces with `strtrim`:

```
sTrim = strtrim(s);
```

```
['|' sTrim '|']      % Make result visible.  
ans =  
    |AAAAA|
```

Expanding Character Arrays

Generally, MathWorks® does not recommend expanding the size of an existing character array by assigning additional characters to indices beyond the bounds of the array such that part of the array becomes padded with zeros.

Cell Arrays of Strings

In this section...

“Converting to a Cell Array of Strings” on page 6-7

“Functions for Cell Arrays of Strings” on page 6-8

Converting to a Cell Array of Strings

Creating strings in a regular MATLAB array requires that all strings in the array be of the same length. This often means that you have to pad blanks at the end of strings to equalize their length. However, another type of MATLAB array, the cell array, can hold different sizes and types of data in an array without padding. Cell arrays provide a more flexible way to store strings of varying length.

The `cellstr` function converts a character array into a cell array of strings. Consider this 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
```

The `iscellstr` function determines if the input argument is a cell array of strings. It returns a logical 1 (true) in the case of `celldata`:

```
iscellstr(celldata)
ans =
     1
```

Use `char` to convert back to a standard padded character array:

```
strings = char(celldata)
strings =
    Allison Jones
    Development
    Phoenix
```

```
length(strings(3,:))
ans =
    13
```

For more information on cell arrays, see “Access Data in a Cell Array” on page 8-5.

Functions for Cell Arrays of Strings

This table describes the MATLAB functions for working with cell arrays.

Function	Description
<code>cellstr</code>	Convert a character array to a cell array of strings.
<code>char</code>	Convert a cell array of strings to a character array.
<code>deblank</code>	Remove trailing blanks from a string.
<code>iscellstr</code>	Return <code>true</code> for a cell array of strings.
<code>sort</code>	Sort elements in ascending or descending order.
<code>strcat</code>	Concatenate strings.
<code>strcmp</code>	Compare strings.

You can also use the following `set` functions with cell arrays of strings.

Function	Description
<code>intersect</code>	Set the intersection of two vectors.
<code>ismember</code>	Detect members of a set.
<code>setdiff</code>	Return the set difference of two vectors.
<code>setxor</code>	Set the exclusive OR of two vectors.
<code>union</code>	Set the union of two vectors.
<code>unique</code>	Set the unique elements of a vector.

Formatting Strings

In this section...
“Functions that Use Format Strings” on page 6-10
“The Format String” on page 6-11
“Input Value Arguments” on page 6-12
“The Formatting Operator” on page 6-13
“Constructing the Formatting Operator” on page 6-14
“Setting Field Width and Precision” on page 6-20
“Restrictions for Using Identifiers” on page 6-23

Functions that Use Format Strings

The following MATLAB functions offer the capability to compose a string that includes ordinary text and data formatted to your specification:

- `fprintf` — Write formatted data to an output string
- `fprintf` — Write formatted data to an output file or the Command Window
- `warning` — Display formatted data in a warning message
- `error` — Display formatted data in an error message and abort
- `assert` — Generate an error when a condition is violated
- `MException` — Capture error information

The syntax of each of these functions includes formatting operators similar to those used by the `printf` function in the C programming language. For example, `%s` tells MATLAB to interpret an input value as a string, and `%d` means to format an integer using decimal notation.

The general formatting syntax for these functions is

```
functionname(..., format_string, value1, value2, ..., valueN)
```

where the `format_string` argument expresses the basic content and formatting of the final output string, and the `value` arguments that follow supply data values to be inserted into the string.

Here is a sample `sprintf` statement, also showing the resulting output string:

```
sprintf('The price of %s on %d/%d/%d was $%.2f.', ...
        'bread', 7, 1, 2006, 2.49)
ans =
    The price of bread on 7/1/2006 was $2.49.
```

Note The examples in this section of the documentation use only the `sprintf` function to demonstrate how string formatting works. However, you can run the examples using the `fprintf`, `warning`, and `error` functions as well.

The Format String

The first input argument in the `sprintf` statement shown above is the `format_string`:

```
'The price of %s on %d/%d/%d was $%.2f.'
```

This argument can include ordinary text, formatting operators and, in some cases, special characters. The formatting operators for this particular string are: `%s`, `%d`, `%d`, `%d`, and `%.2f`.

Following the `format_string` argument are five additional input arguments, one for each of the formatting operators in the string:

```
'bread', 7, 1, 2006, 2.49
```

When MATLAB processes the format string, it replaces each operator with one of these input values.

Special Characters

Special characters are a part of the text in the string. But, because they cannot be entered as ordinary text, they require a unique character sequence to represent them. Use any of the following character sequences to insert special characters into the output string.

To Insert a . . .	Use . . .
Single quotation mark	' '
Percent character	%
Backslash	\\
Alarm	\a
Backspace	\b
Form feed	\f
New line	\n
Carriage return	\r
Horizontal tab	\t
Vertical tab	\v
Hexadecimal number, N	\xN
Octal number, N	\N

Input Value Arguments

In the syntax

```
functionname(..., format_string, value1, value2, ..., valueN)
```

The value arguments must immediately follow `format_string` in the argument list. In most instances, you supply one of these value arguments for each formatting operator used in the `format_string`. Scalars, vectors, and numeric and character arrays are valid value arguments. You cannot use cell arrays or structures.

If you include fewer formatting operators than there are values to insert, MATLAB reuses the operators on the additional values. This example shows two formatting operators and six values to insert into the string:

```
sprintf('%s = %d\n', 'A', 479, 'B', 352, 'C', 651)
ans =
  A = 479
  B = 352
  C = 651
```


You can also specify multiple value arguments as a vector or matrix. The `format_string` needs one `%s` operator for the entire matrix or vector:

```
mvec = [77 65 84 76 65 66];

sprintf('%s ', char(mvec))
ans =
    MATLAB
```

Sequential and Numbered Argument Specification

You can place value arguments in the argument list either sequentially (that is, in the same order in which their formatting operators appear in the string), or by identifier (adding a number to each operator that identifies which value argument to replace it with). By default, MATLAB uses sequential ordering.

To specify arguments by a numeric identifier, add a positive integer followed by a `$` sign immediately after the `%` sign in the operator. Numbered argument specification is explained in more detail under the topic “Value Identifiers” on page 6-20.

Ordered Sequentially	Ordered By Identifier
<pre>sprintf('%s %s %s', ... '1st', '2nd', '3rd') ans = 1st 2nd 3rd</pre>	<pre>sprintf('%3\$s %2\$s %1\$s', ... '1st', '2nd', '3rd') ans = 3rd 2nd 1st</pre>

The Formatting Operator

Formatting operators tell MATLAB how to format the numeric or character value arguments and where to insert them into the string. These operators control the notation, alignment, significant digits, field width, and other aspects of the output string.

A formatting operator begins with a `%` character, which may be followed by a series of one or more numbers, characters, or symbols, each playing a role in further defining the format of the insertion value. The final entry in this series is a single *conversion character* that MATLAB uses to define the notation

style for the inserted data. Conversion characters used in MATLAB are based on those used by the `printf` function in the C programming language.

Here is a simple example showing five formatting variations for a common value:

```
A = pi*100*ones(1,5);

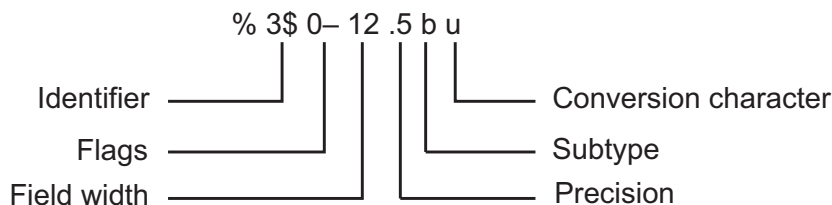
sprintf(' %f \n %.2f \n %+.2f \n %12.2f \n %012.2f \n', A)
ans =
    314.159265      % Display in fixed-point notation (%f)
    314.16         % Display 2 decimal digits (%.2f)
   +314.16        % Display + for positive numbers (%+.2f)
           314.16  % Set width to 12 characters (%12.2f)
000000314.16     % Replace leading spaces with 0 (%012.2f)
```

Constructing the Formatting Operator

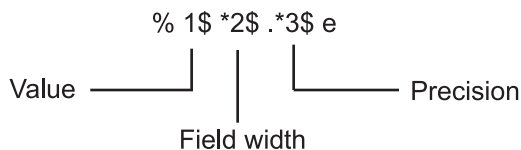
The fields that make up a formatting operator in MATLAB are as shown here, in the order they appear from right to left in the operator. The rightmost field, the conversion character, is required; the five to the left of that are optional. Each of these fields is explained in a section below:

- Conversion Character — Specifies the notation of the output.
- Subtype — Further specifies any nonstandard types.
- Precision — Sets the number of digits to display to the right of the decimal point, or the number of significant digits to display.
- Field Width — Sets the minimum number of digits to display.
- Flags — Controls the alignment, padding, and inclusion of plus or minus signs.
- Value Identifiers — Map formatting operators to value input arguments. Use the identifier field when value arguments are not in a sequential order in the argument list.

Here is an example of a formatting operator that uses all six fields. (Space characters are not allowed in the operator. They are shown here only to improve readability of the figure).



An alternate syntax, that enables you to supply values for the field width and precision fields from values in the argument list, is shown below. See the section “Specifying Field Width and Precision Outside the format String” on page 6-21 for information on when and how to use this syntax. (Again, space characters are shown only to improve readability of the figure.)



Each field of the formatting operator is described in the following sections. These fields are covered as they appear going from right to left in the formatting operator, starting with the **Conversion Character** and ending with the **Identifier** field.

Conversion Character

The conversion character specifies the notation of the output. It consists of a single character and appears last in the format specifier. It is the only required field of the format specifier other than the leading `%` character.

Specifier	Description
c	Single character
d	Decimal notation (signed)
e	Exponential notation (using a lowercase e as in 3.1415e+00)
E	Exponential notation (using an uppercase E as in 3.1415E+00)
f	Fixed-point notation

Specifier	Description
g	The more compact of %e or %f. (Insignificant zeros do not print.)
G	Same as %g, but using an uppercase E
o	Octal notation (unsigned)
s	String of characters
u	Decimal notation (unsigned)
x	Hexadecimal notation (using lowercase letters a–f)
X	Hexadecimal notation (using uppercase letters A–F)

This example uses conversion characters to display the number 46 in decimal, fixed-point, exponential, and hexadecimal formats:

```
A = 46*ones(1,4);  
  
sprintf('%d  %f  %e  %X', A)  
ans =  
46  46.000000  4.600000e+01  2E
```

Subtype

The subtype field is a single alphabetic character that immediately precedes the conversion character. The following nonstandard subtype specifiers are supported for the conversion characters %o, %x, %X, and %u.

b	The underlying C data type is a double rather than an unsigned integer. For example, to print a double-precision value in hexadecimal, use a format like '%bx'.
t	The underlying C data type is a float rather than an unsigned integer.

To specify the number of bits for the conversion of an integer value (corresponding to conversion characters %d, %i, %u, %o, %x, or %X), use one of the following subtypes.

l	64-bit value.
h	16-bit value.

Precision

precision in a formatting operator is a nonnegative integer that immediately follows a period. For example, the specifier `%7.3f`, has a precision of 3. For the `%g` specifier, precision indicates the number of significant digits to display. For the `%f`, `%e`, and `%E` specifiers, precision indicates how many digits to display to the right of the decimal point.

Here are some examples of how the precision field affects different types of notation:

```
sprintf('%g  %.2g  %f  %.2f', pi*50*ones(1,4))
ans =
157.08  1.6e+02  157.079633  157.08
```

Precision is not usually used in format specifiers for strings (i.e., `%s`). If you do use it on a string and if the value `p` in the precision field is less than the number of characters in the string, MATLAB displays only `p` characters of the string and truncates the rest.

You can also supply the value for a precision field from outside of the format specifier. See the section “Specifying Field Width and Precision Outside the format String” on page 6-21 for more information on this.

For more information on the use of precision in formatting, see “Setting Field Width and Precision” on page 6-20.

Field Width

Field width in a formatting operator is a nonnegative integer that tells MATLAB the minimum number of digits or characters to use when formatting the corresponding input value. For example, the specifier `%7.3f`, has a width of 7.

Here are some examples of how the width field affects different types of notation:

```
sprintf('%e|%15e|%f|%15f|', pi*50*ones(1,4))
ans =
|1.570796e+02| 1.570796e+02|157.079633| 157.079633|
```

When used on a string, the `field width` can determine whether MATLAB pads the string with spaces. If `width` is less than or equal to the number of characters in the string, it has no effect.

```
sprintf('%30s', 'Pad left with spaces')
ans =
    Pad left with spaces
```

You can also supply a value for `field width` from outside of the format specifier. See the section “Specifying Field Width and Precision Outside the format String” on page 6-21 for more information on this.

For more information on the use of `field width` in formatting, see “Setting Field Width and Precision” on page 6-20.

Flags

You can control the output using any of these optional flags:

Character	Description	Example
A minus sign (-)	Left-justifies the converted argument in its field.	<code>%-5.2d</code>
A plus sign (+)	Always prints a sign character (+ or -).	<code>%+5.2d</code>
A space ()	Inserts a space before the value.	<code>% 5.2f</code>

Character	Description	Example
Zero (0)	Pads with zeros rather than spaces.	<code>%05.2f</code>
A pound sign (#)	Modifies selected numeric conversions: <ul style="list-style-type: none"> • For <code>%o</code>, <code>%x</code>, or <code>%X</code>, print <code>0</code>, <code>0x</code>, or <code>0X</code> prefix. • For <code>%f</code>, <code>%e</code>, or <code>%E</code>, print decimal point even when precision is 0. • For <code>%g</code> or <code>%G</code>, do not remove trailing zeros or decimal point. 	<code>##5.0f</code>

Right- and left-justify the output. The default is to right-justify:

```
printf('right-justify: %12.2f\nleft-justify: %-12.2f', ...
      12.3, 12.3)
ans =
    right-justify:      12.30
    left-justify: 12.30
```

Display a + sign for positive numbers. The default is to omit the + sign:

```
printf('no sign: %12.2f\nsign: %+12.2f', ...
      12.3, 12.3)
ans =
    no sign:      12.30
    sign:      +12.30
```

Pad to the left with spaces or zeros. The default is to use space-padding:

```
printf('space-padded: %12.2f\nzero-padded: %012.2f', ...
      5.2, 5.2)
ans =
    space-padded:      5.20
```

zero-padded: 000000005.20

Note You can specify more than one flag in a formatting operator.

Value Identifiers

By default, MATLAB inserts data values from the argument list into the string in a sequential order. If you have a need to use the value arguments in a nonsequential order, you can override the default by using a numeric identifier in each format specifier. Specify nonsequential arguments with an integer immediately following the % sign, followed by a \$ sign.

Ordered Sequentially	Ordered By Identifier
<pre>sprintf('%s %s %s', ... '1st', '2nd', '3rd') ans = 1st 2nd 3rd</pre>	<pre>sprintf('%3\$s %2\$s %1\$s', ... '1st', '2nd', '3rd') ans = 3rd 2nd 1st</pre>

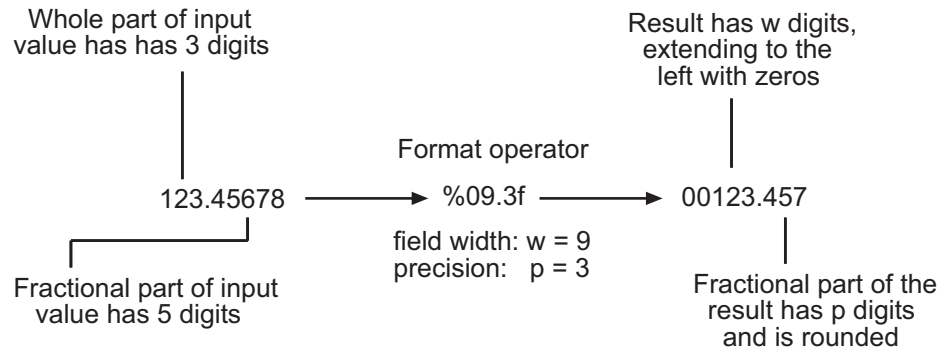
Setting Field Width and Precision

This section provides further information on the use of the field width and precision fields of the formatting operator:

- “Effect on the Output String” on page 6-20
- “Specifying Field Width and Precision Outside the format String” on page 6-21
- “Using Identifiers In the Width and Precision Fields” on page 6-22

Effect on the Output String

The figure below illustrates the way in which the field width and precision settings affect the output of the string formatting functions. In this figure, the zero following the % sign in the formatting operator means to add leading zeros to the output string rather than space characters:



General rules for formatting

- If precision is not specified, it defaults to 6.
- If precision (p) is less than the number of digits in the fractional part of the input value (f), then only p digits are shown to the right of the decimal point in the output, and that fractional value is rounded.
- If precision (p) is greater than the number of digits in the fractional part of the input value (f), then p digits are shown to the right of the decimal point in the output, and the fractional part is extended to the right with $p - f$ zeros.
- If field width is not specified, it defaults to $\text{precision} + 1 +$ the number of digits in the whole part of the input value.
- If field width (w) is greater than $p+1$ plus the number of digits in the whole part of the input value (n), then the whole part of the output value is extended to the left with $w - (n+1+p)$ space characters or zeros, depending on whether or not the zero flag is set in the `Flags` field. The default is to extend the whole part of the output with space characters.

Specifying Field Width and Precision Outside the format String

To specify field width or precision using values from a sequential argument list, use an asterisk (*) in place of the field width or precision field of the formatting operator.

This example formats and displays three numbers. The formatting operator for the first, `%*f`, has an asterisk in the field width location of the formatting operator, specifying that just the field width, 15, is to be taken from the argument list. The second operator, `%.*f` puts the asterisk after the decimal point meaning, that it is the precision that is to take its value from the argument list. And the third operator, `%*.*f`, specifies both field width and precision in the argument list:

```
printf('%*f  %.*f  %*.*f', ...
      15, 123.45678, ...      % Width for 123.45678 is 15
      3, 16.42837, ...      % Precision for rand*20 is .3
      6, 4, pi)             % Width & Precision for pi is 6.4
ans =
    123.456780    16.428    3.1416
```

You can mix the two styles. For example, this statement gets the field width from the argument list and the precision from the format string:

```
printf('%*.*2f', 5, 123.45678)
ans =
    123.46
```

Using Identifiers In the Width and Precision Fields

You can also derive field width and precision values from a nonsequential (i.e., numbered) argument list. Inside the formatting operator, specify field width and/or precision with an asterisk followed by an identifier number, followed by a \$ sign.

This example from the previous section shows how to obtain field width and precision from a sequential argument list:

```
printf('%*f  %.*f  %*.*f', ...
      15, 123.45678, ...
      3, 16.42837, ...
      6, 4, pi)

ans =
    123.456780    16.428    3.1416
```

Here is an example of how to do the same thing using numbered ordering. Field width for the first output value is 15, precision for the second value is 3, and field width and precision for the third value is 6 and 4, respectively. If you specify field width or precision with identifiers, then you must specify the value with an identifier as well:

```
sprintf('%1$*4$f %2$.*5$f %3$*6$.*7$f', ...
123.45678, 16.42837, pi, 15, 3, 6, 4)
```

```
ans =
    123.456780    16.428    3.1416
```

Restrictions for Using Identifiers

If any of the formatting operators in a string include an identifier field, then all of the operators in that string must do the same; you cannot use both sequential and nonsequential ordering in the same function call.

Valid Syntax	Invalid Syntax
<pre>sprintf('%d %d %d %d', ... 1, 2, 3, 4) ans = 1 2 3 4</pre>	<pre>sprintf('%d %3\$d %d %d', ... 1, 2, 3, 4) ans = 1</pre>

If your command provides more value arguments than there are formatting operators in the format string, MATLAB reuses the operators. However, MATLAB allows this only for commands that use sequential ordering. You cannot reuse formatting operators when making a function call with numbered ordering of the value arguments.

Valid Syntax	Invalid Syntax
<pre>sprintf('%d', 1, 2, 3, 4) ans = 1234</pre>	<pre>sprintf('%1\$d', 1, 2, 3, 4) ans = 1</pre>

Also, do not use identifiers when the value arguments are in the form of a vector or array:

Valid Syntax	Invalid Syntax
<pre>v = [1.4 2.7 3.1]; sprintf('%.4f %.4f %.4f', v) ans = 1.4000 2.7000 3.1000</pre>	<pre>v = [1.4 2.7 3.1]; sprintf('%3\$.4f %1\$.4f %2\$.4f', v) ans = Empty string: 1-by-0</pre>

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 white space.

These functions work for both character arrays and cell arrays of strings.

Comparing Strings for Equality

You can use any of four functions to 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 logical 0 (false). For example,

```
C = strcmp(str1,str2)  
C =  
    0
```

Note For C programmers, this is an important difference between the MATLAB `strcmp` and C `strcmp()` functions, where the latter returns 0 if the two strings are the same.

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 where the matching characters are in two strings:

```
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 three functions for categorizing characters inside a string:

- 1 `isletter` determines if a character is a letter.
- 2 `isspace` determines if a character is white space (blank, tab, or new line).
- 3 `isstrprop` checks characters in a string to see if they match a category you specify, such as
 - Alphabetic
 - Alphanumeric
 - Lowercase or uppercase
 - Decimal digits
 - Hexadecimal digits
 - Control characters
 - Graphic characters
 - Punctuation characters
 - Whitespace characters

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 logical 1 (true) because the first four characters of `mystring` are letters.

Searching and Replacing

MATLAB provides several functions for searching and replacing characters in a string. (MATLAB also supports search and replace operations using regular expressions. See Regular Expressions.)

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
```

`strfind` returns the starting position of a substring within a longer string. To find all occurrences of the string 'amp' inside `label`, use

```
position = strfind(label, 'amp')
position =
     2
```

The position within `label` where the only occurrence of 'amp' begins is the second character.

The `textscan` function parses a string to identify numbers or substrings. Describe each component of the string with conversion specifiers, such as %s for strings, %d for integers, or %f for floating-point numbers. Optionally, include any literal text to ignore.

For example, identify the sample number and date string from `label`:

```
parts = textscan(label, 'Sample %d, %s');
parts{:}

ans =
     1
ans =
    '10/28/95'
```


To parse strings in a cell array, use the `strtok` function. For example:

```
c = {'all in good time'; ...  
    'my dog has fleas'; ...  
    'leave no stone unturned'};  
first_words = strtok(c)
```

Converting from Numeric to String

In this section...
“Function Summary” on page 6-30
“Converting to a Character Equivalent” on page 6-31
“Converting to a String of Numbers” on page 6-31
“Converting to a Specific Radix” on page 6-31

Function Summary

The functions listed in this table provide a number of ways to convert numeric data to character strings.

Function	Description	Example
char	Convert a positive integer to an equivalent character. (Truncates any fractional parts.)	[72 105] → 'Hi'
int2str	Convert a positive or negative integer to a character type. (Rounds any fractional parts.)	[72 105] → '72 105'
num2str	Convert a numeric type to a character type of the specified precision and format.	[72 105] → '72/105/' (format set to %1d/)
mat2str	Convert a numeric type to a character type of the specified precision, returning a string MATLAB can evaluate.	[72 105] → '[72 105]'
dec2hex	Convert a positive integer to a character type of hexadecimal base.	[72 105] → '48 69'
dec2bin	Convert a positive integer to a character type of binary base.	[72 105] → '1001000 1101001'
dec2base	Convert a positive integer to a character type of any base from 2 through 36.	[72 105] → '110 151' (base set to 8)

Converting to a Character Equivalent

The `char` function converts integers to Unicode® character codes and returns a string composed of the equivalent characters:

```
x = [77 65 84 76 65 66];  
char(x)  
ans =  
    MATLAB
```

Converting to a String of Numbers

The `int2str`, `num2str`, and `mat2str` functions convert numeric values to strings where each character represents a separate digit of the input value. The `int2str` and `num2str` functions are often useful 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);
```

Converting to a Specific Radix

Another class of conversion functions changes numeric values into strings representing a decimal value in another base, such as binary or hexadecimal representation. This includes `dec2hex`, `dec2bin`, and `dec2base`.

Converting from String to Numeric

In this section...

“Function Summary” on page 6-32

“Converting from a Character Equivalent” on page 6-33

“Converting from a Numeric String” on page 6-33

“Converting from a Specific Radix” on page 6-34

Function Summary

The functions listed in this table provide a number of ways to convert character strings to numeric data.

Function	Description	Example
uintN (e.g., uint8)	Convert a character to an integer code that represents that character.	'Hi' → 72 105
str2num	Convert a character type to a numeric type.	'72 105' → [72 105]
str2double	Similar to str2num, but offers better performance and works with cell arrays of strings.	{'72' '105'} → [72 105]
hex2num	Convert a numeric type to a character type of specified precision, returning a string that MATLAB can evaluate.	'A' → '-1.4917e-154'
hex2dec	Convert a character type of hexadecimal base to a positive integer.	'A' → 10
bin2dec	Convert a positive integer to a character type of binary base.	'1010' → 10
base2dec	Convert a positive integer to a character type of any base from 2 through 36.	'12' → 10 (if base == 8)

Converting from a Character Equivalent

Character arrays store each character as a 16-bit numeric value. Use one of the integer conversion functions (e.g., `uint8`) or the `double` function to convert strings to their numeric values, and `char` to revert to character representation:

```
name = 'Thomas R. Lee';

name = double(name)
name =
    84  104  111  109  97  115  32  82  46  32  76  101  101

name = char(name)
name =
    Thomas R. Lee
```

Converting from a Numeric String

Use `str2num` to convert a character array to the numeric value represented by that string:

```
str = '37.294e-1';

val = str2num(str)
val =
    3.7294
```

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'};

d = str2double(c)
d =
    3.7294
   -58.3750
    13.7960

whos
  Name      Size      Bytes  Class
  ----      -
  c         3x1         224   cell
```

d 3x1 24 double

Converting from a Specific Radix

To convert from a character representation of a nondecimal number to the value of that number, use one of these functions: `hex2num`, `hex2dec`, `bin2dec`, or `base2dec`.

The `hex2num` and `hex2dec` functions both take hexadecimal (base 16) inputs, but `hex2num` returns the IEEE double-precision floating-point number it represents, while `hex2dec` converts to a decimal integer.

Function Summary

MATLAB provides these functions for working with character arrays:

- Functions to Create Character Arrays on page 6-35
- Functions to Modify Character Arrays on page 6-35
- Functions to Read and Operate on Character Arrays on page 6-36
- Functions to Search or Compare Character Arrays on page 6-36
- Functions to Determine Class or Content on page 6-36
- Functions to Convert Between Numeric and String Classes on page 6-37
- Functions to Work with Cell Arrays of Strings as Sets on page 6-37

Functions to Create Character Arrays

Function	Description
'str'	Create the string specified between quotes.
blanks	Create a string of blanks.
sprintf	Write formatted data to a string.
strcat	Concatenate strings.
char	Concatenate strings vertically.

Functions to Modify Character Arrays

Function	Description
deblank	Remove trailing blanks.
lower	Make all letters lowercase.
sort	Sort elements in ascending or descending order.
strjust	Justify a string.
strrep	Replace one string with another.
strtrim	Remove leading and trailing white space.
upper	Make all letters uppercase.

Functions to Read and Operate on Character Arrays

Function	Description
eval	Execute a string with MATLAB expression.
sscanf	Read a string under format control.

Functions to Search or Compare Character Arrays

Function	Description
regexp	Match regular expression.
strcmp	Compare strings.
strcmpi	Compare strings, ignoring case.
strfind	Find one string within another.
strncmp	Compare the first N characters of strings.
strncmpi	Compare the first N characters, ignoring case.
strtok	Find a token in a string.
textscan	Read data from a string.

Functions to Determine Class or Content

Function	Description
iscellstr	Return true for a cell array of strings.
ischar	Return true for a character array.
isletter	Return true for letters of the alphabet.
isstrprop	Determine if a string is of the specified category.
isspace	Return true for white-space characters.

Functions to Convert Between Numeric and String Classes

Function	Description
char	Convert to a character or string.
cellstr	Convert a character array to a cell array of strings.
double	Convert a string to numeric codes.
int2str	Convert an integer to a string.
mat2str	Convert a matrix to a string you can run eval on.
num2str	Convert a number to a string.
str2num	Convert a string to a number.
str2double	Convert a string to a double-precision value.

Functions to Work with Cell Arrays of Strings as Sets

Function	Description
intersect	Set the intersection of two vectors.
ismember	Detect members of a set.
setdiff	Return the set difference of two vectors.
setxor	Set the exclusive OR of two vectors.
union	Set the union of two vectors.
unique	Set the unique elements of a vector.

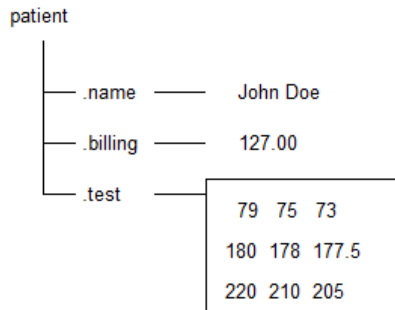
Structures

- “Create a Structure Array” on page 7-2
- “Access Data in a Structure Array” on page 7-6
- “Concatenate Structures” on page 7-9
- “Generate Field Names from Variables” on page 7-11
- “Access Data in Nested Structures” on page 7-12
- “Access Multiple Elements of a Nonscalar Struct Array” on page 7-14
- “Ways to Organize Data in Structure Arrays” on page 7-16
- “Memory Requirements for a Structure Array” on page 7-20

Create a Structure Array

This example shows how to create a structure array. A structure is a data type that groups related data using data containers called fields. Each field can contain data of any type or size.

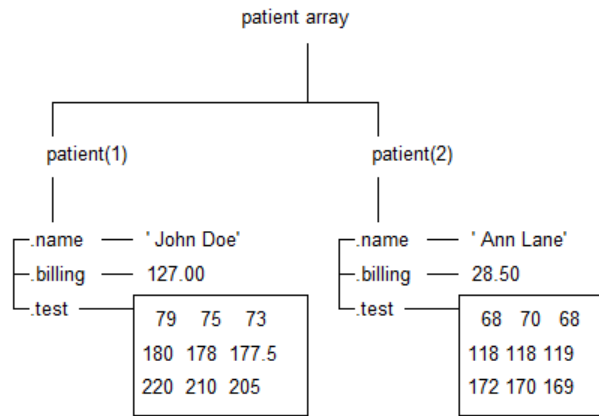
Store a patient record in a scalar structure with fields `name`, `billing`, and `test`.



```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79, 75, 73; 180, 178, 177.5; 220, 210, 205];
patient
```

```
patient =
    name: 'John Doe'
  billing: 127
    test: [3x3 double]
```

Add records for other patients to the array by including subscripts after the array name.



```

patient =

1x2 struct array with fields:
    name
    billing
    test

```

Each patient record in the array is a structure of class `struct`. An array of structures is often referred to as a struct array. Like other MATLAB arrays, a struct array can have any dimensions.

A struct array has the following properties:

- All structs in the array have the same number of fields.
- All structs have the same field names.

- Fields of the same name in different structs can contain different types or sizes of data.

Any unspecified fields for new structs in the array contain empty arrays.

```
patient(3).name = 'New Name';  
patient(3)
```

```
ans =
```

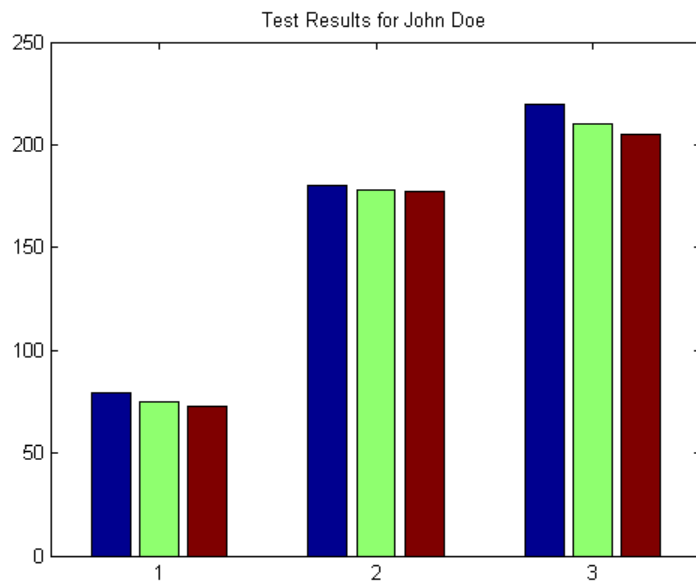
```
    name: 'New Name'  
  billing: []  
    test: []
```

Access data in the structure array to find how much the first patient owes, and to create a bar graph of his test results.

```
amount_due = patient(1).billing  
bar(patient(1).test)  
title(['Test Results for ', patient(1).name])
```

```
amount_due =
```

```
127
```



Access Data in a Structure Array

This example shows how to access the contents of a structure array. To run the code in this example, load several variables into a scalar (1-by-1) structure named `S`.

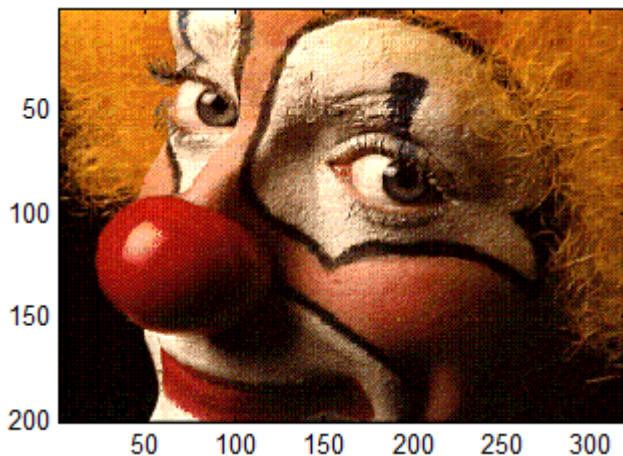
```
S = load('clown.mat')
```

The variables from the file (`X`, `caption`, and `map`) are now fields in the struct.

```
S =  
      X: [200x320 double]  
      map: [81x3 double]  
      caption: [2x1 char]
```

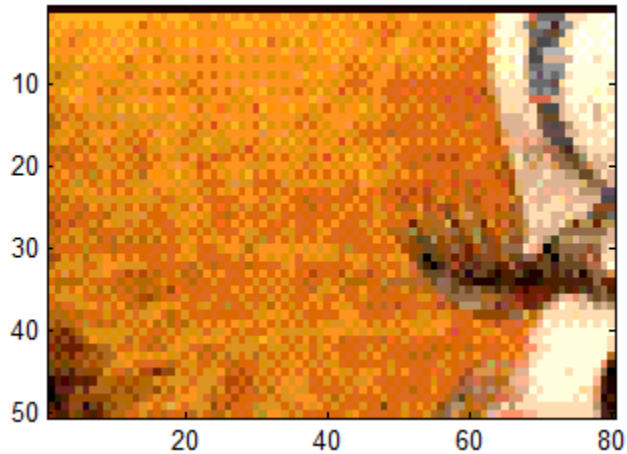
Access the data using dot notation of the form `structName.fieldName`. For example, pass the numeric data in field `X` to the `image` function:

```
image(S.X)  
colormap(S.map)
```



To access part of a field, add indices as appropriate for the size and type of data in the field. For example, pass the upper left corner of `X` to the `image` function:

```
upperLeft = S.X(1:50,1:80);  
image(upperLeft);
```

If a particular field contains a cell array, use curly braces to access the data, such as `S.cellField{1:50,1:80}`.

Data in Nonscalar Structure Arrays

Create a nonscalar array by loading data from the file `mandrill.mat` into a second element of array `S`:

```
S(2) = load('mandrill.mat')
```

Each element of a structure array must have the same fields. Both `clown.mat` and `mandrill.mat` contain variables `X`, `map`, and `caption`.

`S` is a 1-by-2 array.

```
S =
1x2 struct array with fields:
    X
    map
    caption
```

For nonscalar structures, the syntax for accessing a particular field is `structName(indices).fieldName`. Redisplay the clown image, specifying the index for the clown struct (1):

```
image(S(1).X)  
colormap(S(1).map)
```

Add indices to select and redisplay the upper left corner of the field contents:

```
upperLeft = S(1).X(1:50,1:80);  
image(upperLeft)
```

Note You can index into part of a field only when you refer to a single element of a structure array. MATLAB does not support statements such as `S(1:2).X(1:50,1:80)`, which attempt to index into a field for multiple elements of the structure.

Related Information

- “Access Data in Nested Structures” on page 7-12
- “Access Multiple Elements of a Nonscalar Struct Array” on page 7-14

Concatenate Structures

This example shows how to concatenate structure arrays using the `[]` operator. To concatenate structures, they must have the same set of fields, but the fields do not need to contain the same sizes or types of data.

Create scalar (1-by-1) structure arrays `struct1` and `struct2`, each with fields `a` and `b`:

```
struct1.a = 'first';  
struct1.b = [1,2,3];
```

```
struct2.a = 'second';  
struct2.b = rand(5);
```

Just as concatenating two scalar values such as `[1, 2]` creates a 1-by-2 numeric array, concatenating `struct1` and `struct2`,

```
combined = [struct1, struct2]
```

creates a 1-by-2 structure array:

```
combined =  
    1x2 struct array with fields:  
    a  
    b
```

When you want to access the contents of a particular field, specify the index of the structure in the array. For example, access field `a` of the first structure:

```
combined(1).a
```

This code returns

```
ans =  
    first
```

Concatenation also applies to nonscalar structure arrays. For example, create a 2-by-2 structure array named `new`:

```
new(1,1).a = 1;    new(1,1).b = 10;  
new(1,2).a = 2;    new(1,2).b = 20;
```

```
new(2,1).a = 3;    new(2,1).b = 30;  
new(2,2).a = 4;    new(2,2).b = 40;
```

Because the 1-by-2 structure `combined` and the 2-by-2 structure `new` both have two columns, you can concatenate them vertically with a semicolon separator:

```
larger = [combined; new]
```

This code returns a 3-by-2 structure array,

```
larger =  
    3x2 struct array with fields:  
        a  
        b
```

where, for example,

```
larger(2,1).a =  
    1
```

For related information, see:

- “Creating and Concatenating Matrices”
- “Access Data in a Structure Array” on page 7-6
- “Access Multiple Elements of a Nonscalar Struct Array” on page 7-14

Generate Field Names from Variables

This example shows how to derive a field name at run time from a variable or expression. The general syntax is

```
structName.(dynamicExpression)
```

where `dynamicExpression` is a variable or expression that returns a character or string. Field names that you reference with expressions are called *dynamic field names*.

For example, create a field name from the current date:

```
currentDate = datestr(now, 'mmmdd');  
myStruct.(currentDate) = [1,2,3]
```

If the current date reported by your system is February 29, then this code assigns data to a field named `Feb29`:

```
myStruct =  
    Feb29: [1 2 3]
```

Field names, like variable names, must begin with a letter, can contain letters, digits, or underscore characters, and are case sensitive. To avoid potential conflicts, do not use the names of existing variables or functions as field names. For more information, see “Variable Names” on page 1-10.

Access Data in Nested Structures

This example shows how to index into a structure that is nested within another structure. The general syntax is

```
structName(indices).nestedStructName.(indices).fieldName(indices)
```

When a structure is scalar (1-by-1), you do not need to include the indices to refer to the single element. For example, create a scalar structure `s`, where field `n` is a nested scalar structure with fields `a`, `b`, and `c`:

```
s.n.a = ones(3);  
s.n.b = eye(4);  
s.n.c = magic(5);
```

Access the third row of field `b`:

```
third_row_b = s.n.b(3,:)
```

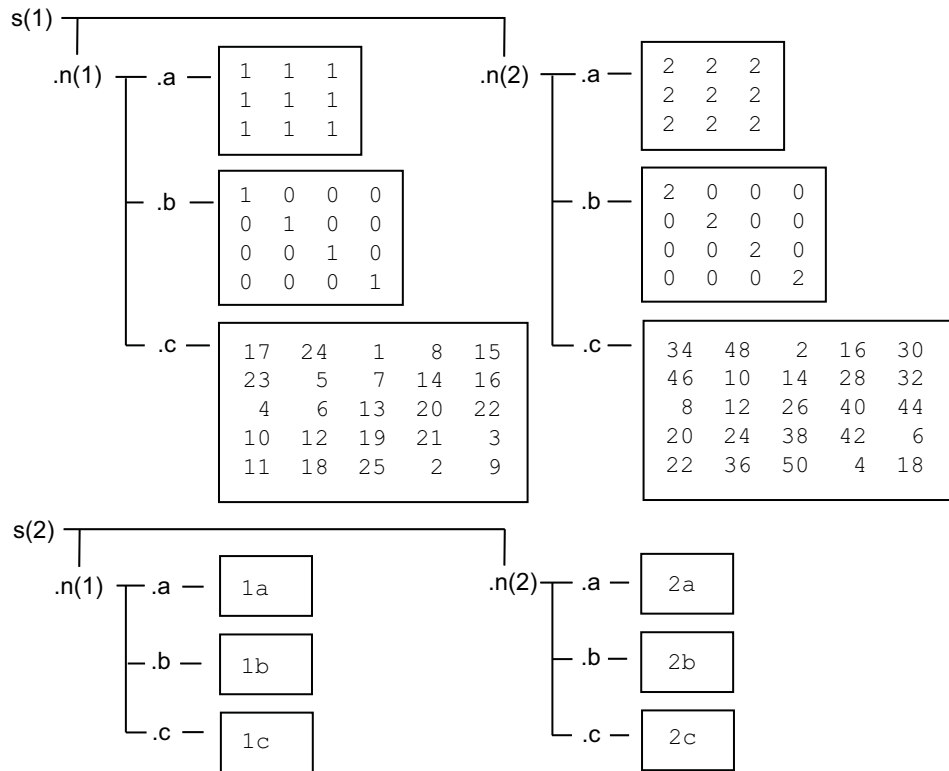
Variable `third_row_b` contains the third row of `eye(4)`.

```
third_row_b =  
    0     0     1     0
```

Expand `s` so that both `s` and `n` are nonscalar (1-by-2):

```
s(1).n(2).a = 2 * ones(3);  
s(1).n(2).b = 2 * eye(4);  
s(1).n(2).c = 2 * magic(5);  
  
s(2).n(1).a = '1a';    s(2).n(2).a = '2a';  
s(2).n(1).b = '1b';    s(2).n(2).b = '2b';  
s(2).n(1).c = '1c';    s(2).n(2).c = '2c';
```

Structure `s` now contains the data shown in the following figure.



Access part of the array in field `b` of the second element in `n` within the first element of `s`:

```
part_two_eye = s(1).n(2).b(1:2,1:2)
```

This returns the 2-by-2 upper left corner of `2 * eye(4)`:

```
part_two_eye =
    2    0
    0    2
```

Access Multiple Elements of a Nonscalar Struct Array

This example shows how to access and process data from multiple elements of a nonscalar structure array:

Create a 1-by-3 structure `s` with field `f`:

```
s(1).f = 1;  
s(2).f = 'two';  
s(3).f = 3 * ones(3);
```

Although each structure in the array must have the same number of fields and the same field names, the contents of the fields can be different types and sizes. When you refer to field `f` for multiple elements of the structure array, such as

```
s(1:3).f
```

or

```
s.f
```

MATLAB returns the data from the elements in a *comma-separated list*, which displays as follows:

```
ans =  
    1  
  
ans =  
    two  
  
ans =  
    3    3    3  
    3    3    3  
    3    3    3
```

You cannot assign the list to a single variable with the syntax `v = s.f` because the fields can contain different types of data. However, you can assign the list items to the same number of variables, such as

```
[v1, v2, v3] = s.f;
```


or assign to elements of a cell array, such as

```
c = {s.f};
```

If all of the fields contain the same type of data and can form a hyperrectangle, you can concatenate the list items. For example, create a structure `nums` with scalar numeric values in field `f`, and concatenate the data from the fields:

```
nums(1).f = 1;  
nums(2).f = 2;  
nums(3).f = 3;  
  
allNums = [nums.f]
```

This code returns

```
allNums =  
      1      2      3
```

If you want to process each element of an array with the same operation, use the `arrayfun` function. For example, count the number of elements in field `f` of each struct in array `s`:

```
numElements = arrayfun(@(x) numel(x.f), s)
```

The syntax `@(x)` creates an anonymous function. This code calls the `numel` function for each element of array `s`, such as `numel(s(1).f)`, and returns

```
numElements =  
      1      3      9
```

For related information, see:

- “Comma-Separated Lists” on page 2-100
- “Anonymous Functions” on page 15-3

Ways to Organize Data in Structure Arrays

There are at least two ways you can organize data in a structure array: plane organization and element-by-element organization. The method that best fits your data depends on how you plan to access the data, and, for very large data sets, whether you have system memory constraints.

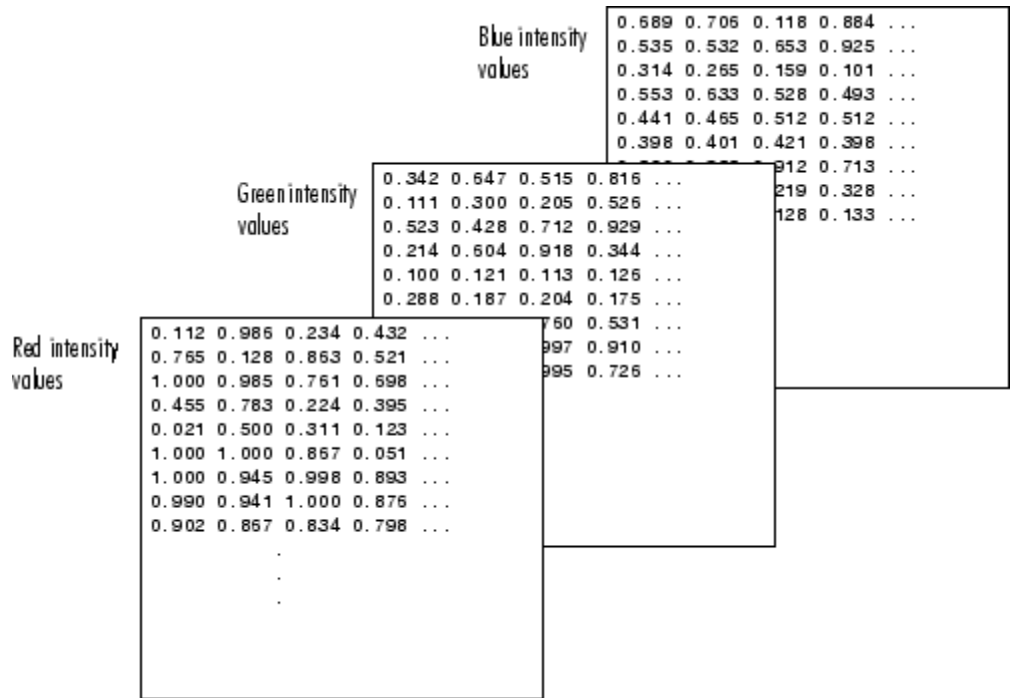
Plane organization allows easier access to all values within a field. Element-by-element organization allows easier access to all information related to a single element or record. The following sections include an example of each type of organization:

- “Plane Organization” on page 7-16
- “Element-by-Element Organization” on page 7-18

When you create a structure array, MATLAB stores information about each element and field in the array header. As a result, structures with more elements and fields require more memory than simpler structures that contain the same data. For more information on memory requirements for arrays, see “Memory Allocation” on page 21-2.

Plane Organization

Consider an RGB image with three arrays corresponding to color intensity values.



If you have arrays RED, GREEN, and BLUE in your workspace, then these commands create a scalar structure named `img` that uses plane organization:

```

img.red = RED;
img.green = GREEN;
img.blue = BLUE;

```

Plane organization allows you to easily extract entire image planes for display, filtering, or other processing. For example, multiply the red intensity values by 0.9:

```

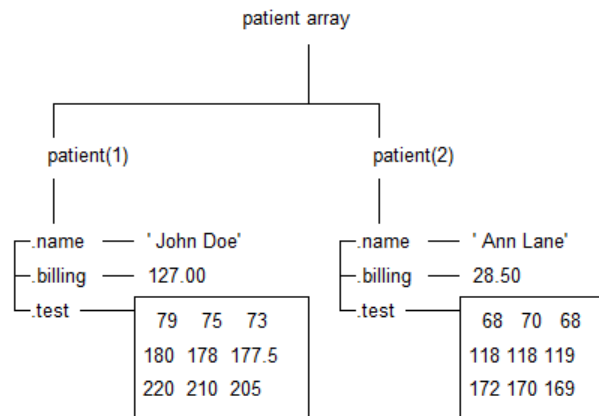
adjustedRed = .9 * img.red;

```

If you have multiple images, you can add them to the `img` structure, so that each element `img(1)`, ..., `img(n)` contains an entire image. For an example that adds elements to a structure, see the following section.

Element-by-Element Organization

Consider a database with patient information. Each record contains data for the patient's name, test results, and billing amount.



These statements create an element in a structure array named patient:

```

patient(1).name = 'John Doe';
patient(1).billing = 127.00;
patient(1).test = [79, 75, 73; 180, 178, 177.5; 220, 210, 205];
  
```

Additional patients correspond to new elements in the structure. For example, add an element for a second patient:

```

patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68, 70, 68; 118, 118, 119; 172, 170, 169];
  
```

Element-by-element organization supports simple indexing to access data for a particular patient. For example, find the average of the first patient's test results, calculating by rows (dimension 2) rather than by columns:

```
aveResultsDoe = mean(patient(1).test,2)
```

This code returns

```
aveResultsDoe =
```

75.6667
178.5000
212.0000

For information on processing data from more than one element at a time, see “Access Data in a Structure Array” on page 7-6.

Memory Requirements for a Structure Array

Structure arrays do not require completely contiguous memory. However, each field requires contiguous memory, as does the header that MATLAB creates to describe the array. For very large arrays, incrementally increasing the number of fields or the number of elements in a field results in Out of Memory errors.

Allocate memory for the contents by assigning initial values with the `struct` function, such as

```
newStruct(1:25,1:50) = struct('a',ones(20),'b',zeros(30),'c',rand(40));
```

This code creates and populates a 25-by-50 structure array `S` with fields `a`, `b`, and `c`.

If you prefer not to assign initial values, you can initialize a structure array by assigning empty arrays to each field of the last element in the structure array, such as

```
newStruct(25,50).a = [];  
newStruct(25,50).b = [];  
newStruct(25,50).c = [];
```

or, equivalently,

```
newStruct(25,50) = struct('a',[],'b',[],'c',[]);
```

However, in this case, MATLAB only allocates memory for the header, and not for the contents of the array.

For more information, see:

- “Preallocating Memory”
- “Memory Allocation” on page 21-2

Cell Arrays

- “What Is a Cell Array?” on page 8-2
- “Create a Cell Array” on page 8-3
- “Access Data in a Cell Array” on page 8-5
- “Add Cells to a Cell Array” on page 8-8
- “Delete Data from a Cell Array” on page 8-9
- “Combine Cell Arrays” on page 8-10
- “Pass Contents of Cell Arrays to Functions” on page 8-11
- “Preallocate Memory for a Cell Array” on page 8-14
- “Cell vs. Struct Arrays” on page 8-15
- “Multilevel Indexing to Access Parts of Cells” on page 8-17

What Is a Cell Array?

A cell array is a data type with indexed data containers called cells. Each cell can contain any type of data. Cell arrays commonly contain lists of text strings, combinations of text and numbers from spreadsheets or text files, or numeric arrays of different sizes.

There are two ways to refer to the elements of a cell array. Enclose indices in smooth parentheses, (), to refer to sets of cells — for example, to define a subset of the array. Enclose indices in curly braces, {}, to refer to the text, numbers, or other data within individual cells.

For more information, see:

- “Create a Cell Array” on page 8-3
- “Access Data in a Cell Array” on page 8-5

Create a Cell Array

This example shows how to create a cell array using the `{}` operator or the `cell` function.

When you have data to put into a cell array, create the array using the cell array construction operator, `{}`:

```
myCell = {1, 2, 3;
          'text', rand(5,10,2), {11; 22; 33}}
```

Like all MATLAB arrays, cell arrays are rectangular, with the same number of cells in each row. `myCell` is a 2-by-3 cell array:

```
myCell =
    [ 1]    [          2]    [          3]
    'text'  [5x10x2 double]  {3x1 cell}
```

You also can use the `{}` operator to create an empty 0-by-0 cell array,

```
C = {}
```

If you plan to add values to a cell array over time or in a loop, you can create an empty *n*-dimensional array using the `cell` function:

```
emptyCell = cell(3,4,2)
```

`emptyCell` is a 3-by-4-by-2 cell array, where each cell contains an empty array, `[]`:

```
emptyCell(:,:,1) =
    []    []    []    []
    []    []    []    []
    []    []    []    []
```

```
emptyCell(:,:,2) =
    []    []    []    []
    []    []    []    []
    []    []    []    []
```

For more information, see:

- “Access Data in a Cell Array” on page 8-5
- “Multidimensional Cell Arrays”

Access Data in a Cell Array

This example shows how to read and write data to and from a cell array. To run the code in this example, create a 2-by-3 cell array of text and numeric data:

```
C = {'one', 'two', 'three';
     1, 2, 3};
```

There are two ways to refer to the elements of a cell array. Enclose indices in smooth parentheses, `()`, to refer to sets of cells—for example, to define a subset of the array. Enclose indices in curly braces, `{}`, to refer to the text, numbers, or other data within individual cells.

Cell Indexing with Smooth Parentheses, `()`

Cell array indices in smooth parentheses refer to sets of cells. For example, the command

```
upperLeft = C(1:2,1:2)
```

creates a 2-by-2 cell array:

```
upperLeft =
    'one'    'two'
    [ 1]    [ 2]
```

Update sets of cells by replacing them with the same number of cells. For example, the statement

```
C(1,1:3) = {'first','second','third'}
```

replaces the cells in the first row of `C` with an equivalent-sized (1-by-3) cell array:

```
C =
    'first'    'second'    'third'
    [ 1]    [ 2]    [ 3]
```

If cells in your array contain numeric data, you can convert the cells to a numeric array using the `cell2mat` function:

```
numericCells = C(2,1:3)
numericVector = cell2mat(numericCells)
```

numericCells is a 1-by-3 cell array, but numericVector is a 1-by-3 array of type double:

```
numericCells =
    [1]    [2]    [3]

numericVector =
     1     2     3
```

Content Indexing with Curly Braces, {}

Access the contents of cells—the numbers, text, or other data within the cells—by indexing with curly braces. For example, the command

```
last = C{2,3}
```

creates a numeric variable of type double, because the cell contains a double value:

```
last =
     3
```

Similarly, this command

```
C{2,3} = 300
```

replaces the contents of the last cell of C with a new, numeric value:

```
C =
    'first'    'second'    'third'
    [    1]    [    2]    [  300]
```

When you access the contents of multiple cells, such as

```
C{1:2,1:2}
```

MATLAB creates a *comma-separated list*. Because each cell can contain a different type of data, you cannot assign this list to a single variable. However, you can assign the list to the same number of variables as cells. MATLAB assigns to the variables in column order. For example,

```
[r1c1, r2c1, r1c2, r2c2] = C{1:2,1:2}
```

returns

```
r1c1 =  
    first
```

```
r2c1 =  
    1
```

```
r1c2 =  
    second
```

```
r2c2 =  
    2
```

If each cell contains the same type of data, you can create a single variable by applying the array concatenation operator, `[]`, to the comma-separated list. For example,

```
nums = [C{2, :}]
```

returns

```
nums =  
    1     2   300
```

For more information, see:

- “Multilevel Indexing to Access Parts of Cells” on page 8-17
- “Comma-Separated Lists” on page 2-100

Add Cells to a Cell Array

This example shows how to add cells to a cell array.

Create a 1-by-3 cell array:

```
C = {1, 2, 3};
```

Assign data to a cell outside the current dimensions:

```
C{4,4} = 44
```

MATLAB expands the cell array to a rectangle that includes the specified subscripts. Any intervening cells contain empty arrays:

```
C =
     [1]     [2]     [3]     []
     []     []     []     []
     []     []     []     []
     []     []     []    [44]
```

Add cells without specifying a value by assigning an empty array as the contents of a cell:

```
C{5,5} = []
```

C is now a 5-by-5 cell array:

```
C =
     [1]     [2]     [3]     []     []
     []     []     []     []     []
     []     []     []     []     []
     []     []     []    [44]    []
     []     []     []     []     []
```

For related examples, see:

- “Access Data in a Cell Array” on page 8-5
- “Combine Cell Arrays” on page 8-10
- “Delete Data from a Cell Array” on page 8-9

Delete Data from a Cell Array

This example shows how to remove data from individual cells, and how to delete entire cells from a cell array. To run the code in this example, create a 3-by-3 cell array:

```
C = {1, 2, 3; 4, 5, 6; 7, 8, 9};
```

Delete the contents of a particular cell by assigning an empty array to the cell, using curly braces for content indexing, {}:

```
C{2,2} = []
```

This code returns

```
C =  
    [1]    [2]    [3]  
    [4]     []    [6]  
    [7]    [8]    [9]
```

Delete sets of cells using standard array indexing with smooth parentheses, (). For example, this command

```
C(2,:) = []
```

removes the second row of C:

```
C =  
    [1]    [2]    [3]  
    [7]    [8]    [9]
```

For related examples, see:

- “Add Cells to a Cell Array” on page 8-8
- “Access Data in a Cell Array” on page 8-5

Combine Cell Arrays

This example shows how to combine cell arrays by concatenation or nesting. To run the code in this example, create several cell arrays with the same number of columns:

```
C1 = {1, 2, 3};  
C2 = {'A', 'B', 'C'};  
C3 = {10, 20, 30};
```

Concatenate cell arrays with the array concatenation operator, `[]`. In this example, vertically concatenate the cell arrays by separating them with semicolons:

```
C4 = [C1; C2; C3]
```

C4 is a 3-by-3 cell array:

```
C4 =  
    [ 1]    [ 2]    [ 3]  
    'A'    'B'    'C'  
   [10]   [20]   [30]
```

Create a nested cell array with the cell array construction operator, `{}`:

```
C5 = {C1; C2; C3}
```

C5 is a 3-by-1 cell array, where each cell contains a cell array:

```
C5 =  
    {1x3 cell}  
    {1x3 cell}  
    {1x3 cell}
```

For more information, see “Concatenating Matrices”.

Pass Contents of Cell Arrays to Functions

These examples show several ways to pass data from a cell array to a MATLAB function that does not recognize cell arrays as inputs.

- Pass the contents of a single cell by indexing with curly braces, {}. Access part of an array within the cell by indexing further into the content (multilevel indexing).

This example creates a cell array that contains text and a 20-by-2 array of random numbers. Plot all of the data, and then plot only the first column of data.

```
randCell = {'Random Data', rand(20,2)};
plot(randCell{1,2})
title(randCell{1,1})
```

```
figure
plot(randCell{1,2}(:,1))
title('First Column of Data')
```

- Combine numeric data from multiple cells using the `cell2mat` function.

This example creates a 5-by-2 cell array that stores temperature data for three cities, and plots the temperatures for each city by date.

```
temperature(1,:) = {'01-Jan-2010', [45, 49, 0]};
temperature(2,:) = {'03-Apr-2010', [54, 68, 21]};
temperature(3,:) = {'20-Jun-2010', [72, 85, 53]};
temperature(4,:) = {'15-Sep-2010', [63, 81, 56]};
temperature(5,:) = {'31-Dec-2010', [38, 54, 18]};
```

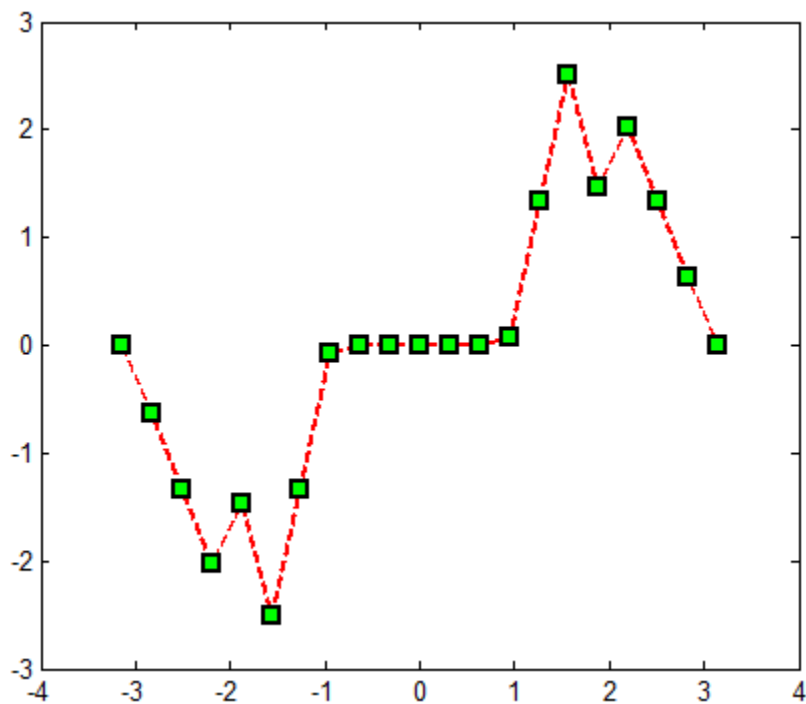
```
allTemps = cell2mat(temperature(:,2));
dates = datenum(temperature(:,1), 'dd-mmm-yyyy');
```

```
plot(dates, allTemps)
datetick('x', 'mmm')
```

- Pass the contents of multiple cells as a comma-separated list to functions that accept multiple inputs.

This example plots X against Y , and applies line styles from a 2-by-3 cell array C .

```
X = -pi:pi/10:pi;  
Y = tan(sin(X)) - sin(tan(X));  
  
C(:,1) = {'LineWidth'; 2};  
C(:,2) = {'MarkerEdgeColor'; 'k'};  
C(:,3) = {'MarkerFaceColor'; 'g'};  
  
plot(X, Y, '--rs', C{:})
```



For more information, see:

- “Access Data in a Cell Array” on page 8-5
- “Multilevel Indexing to Access Parts of Cells” on page 8-17

- “Comma-Separated Lists” on page 2-100

Preallocate Memory for a Cell Array

This example shows how to initialize and allocate memory for a cell array.

Cell arrays do not require completely contiguous memory. However, each cell requires contiguous memory, as does the cell array header that MATLAB creates to describe the array. For very large arrays, incrementally increasing the number of cells or the number of elements in a cell results in **Out of Memory** errors.

Initialize a cell array by calling the `cell` function, or by assigning to the last element. For example, these statements are equivalent:

```
C = cell(25,50);  
C{25,50} = [];
```

MATLAB creates the header for a 25-by-50 cell array. However, MATLAB does not allocate any memory for the contents of each cell.

For more information, see:

- “Preallocating Memory”
- “Memory Allocation” on page 21-2

Cell vs. Struct Arrays

This example compares cell and structure arrays, and shows how to store data in each type of array. Both cell and structure arrays allow you to store data of different types and sizes.

Structure Arrays

Structure arrays contain data in fields that you access by name.

For example, store patient records in a structure array.

```
patient(1).name = 'John Doe';  
patient(1).billing = 127.00;  
patient(1).test = [79, 75, 73; 180, 178, 177.5; 220, 210, 205];
```

```
patient(2).name = 'Ann Lane';  
patient(2).billing = 28.50;  
patient(2).test = [68, 70, 68; 118, 118, 119; 172, 170, 169];
```

Create a bar graph of the test results for each patient.

```
numPatients = numel(patient);  
for p = 1:numPatients  
    figure  
    bar(patient(p).test)  
    title(patient(p).name)  
end
```

Cell Arrays

Cell arrays contain data in cells that you access by numeric indexing. Common applications of cell arrays include storing lists of text strings and storing heterogeneous data from spreadsheets.

For example, store temperature data for three cities over time in a cell array.

```
temperature(1,:) = {'01-Jan-2010', [45, 49, 0]};  
temperature(2,:) = {'03-Apr-2010', [54, 68, 21]};  
temperature(3,:) = {'20-Jun-2010', [72, 85, 53]};  
temperature(4,:) = {'15-Sep-2010', [63, 81, 56]};
```

```
temperature(5,:) = {'31-Dec-2010', [38, 54, 18]};
```

Plot the temperatures for each city by date.

```
allTemps = cell2mat(temperature(:,2));  
dates = datenum(temperature(:,1), 'dd-mmm-yyyy');
```

```
plot(dates,allTemps)  
datetick('x','mmm')
```

Other Container Arrays

Struct and cell arrays are the most commonly used containers for storing heterogeneous data. If you have installed the Statistics Toolbox™, you can also use dataset arrays. Alternatively, use map containers, or create your own class.

Related Topics

- “Access Data in a Cell Array” on page 8-5
- “Access Data in a Structure Array” on page 7-6
- Chapter 10, “Map Containers”
- “Classes in the MATLAB Language”

Multilevel Indexing to Access Parts of Cells

This example explains techniques for accessing data in arrays stored within cells of cell arrays. To run the code in this example, create a sample cell array:

```
myNum = [1, 2, 3];
myCell = {'one', 'two'};
myStruct.Field1 = ones(3);
myStruct.Field2 = 5*ones(5);
```

```
C = {myNum, 100*myNum;
     myCell, myStruct};
```

Access the complete contents of a particular cell using curly braces, {}. For example,

```
C{1,2}
```

returns the numeric vector from that cell:

```
ans =
    100    200    300
```

Access part of the contents of a cell by appending indices, using syntax that matches the data type of the contents. For example:

- Enclose numeric indices in smooth parentheses. For example, `C{1,1}` returns the 1-by-3 numeric vector, `[1, 2, 3]`. Access the second element of that vector with the syntax

```
C{1,1}(1,2)
```

which returns

```
ans =
     2
```

- Enclose cell array indices in curly braces. For example, `C{2,1}` returns the cell array `{'one', 'two'}`. Access the contents of the second cell within that cell array with the syntax

```
C{2,1}{1,2}
```

which returns

```
ans =  
    two
```

- Refer to fields of a struct array with dot notation, and index into the array as described for numeric and cell arrays. For example, `C{2,2}` returns a structure array, where `Field2` contains a 5-by-5 numeric array of fives. Access the element in the fifth row and first column of that field with the syntax

```
C{2,2}.Field2(5,1)
```

which returns

```
ans =  
    5
```

You can nest any number of cell and structure arrays. For example, add nested cells and structures to `C`.

```
C{2,1}{2,2} = {pi, eps};  
C{2,2}.Field3 = struct('NestedField1', rand(3), ...  
                      'NestedField2', magic(4), ...  
                      'NestedField3', {'text'; 'more text'} );
```

These assignment statements access parts of the new data:

```
copy_pi = C{2,1}{2,2}{1,1}  
  
part_magic = C{2,2}.Field3.NestedField2(1:2,1:2)  
  
nested_cell = C{2,2}.Field3.NestedField3{2,1}
```

MATLAB displays:

```
copy_pi =  
    3.1416  
  
part_magic =  
    16     2  
     5    11
```



```
nested_cell =  
    more text
```


Function Handles

- “What Is a Function Handle?” on page 9-2
- “Creating a Function Handle” on page 9-3
- “Calling a Function By Means of Its Handle” on page 9-7
- “Preserving Data from the Workspace” on page 9-10
- “Applications of Function Handles” on page 9-13
- “Saving and Loading Function Handles” on page 9-19
- “Advanced Operations on Function Handles” on page 9-20
- “Functions That Operate on Function Handles” on page 9-27

What Is a Function Handle?

A *function handle* is a callable association to a MATLAB function. It contains an association to that function that enables you to invoke the function regardless of where you call it from. This means that, even if you are outside the normal scope of a function, you can still call it if you use its handle.

With function handles, you can:

- Pass a function to another function
- Capture data values for later use by a function
- Call functions outside of their normal scope
- Save the handle in a MAT-file to be used in a later MATLAB session

See “Applications of Function Handles” on page 9-13 for an explanation of each of these applications.

Creating a Function Handle

In this section...

“Maximum Length of a Function Name” on page 9-4

“The Role of Scope, Precedence, and Overloading When Creating a Function Handle” on page 9-4

“Obtaining Permissions from Class Methods” on page 9-5

“Using Function Handles for Anonymous Functions” on page 9-6

“Arrays of Function Handles” on page 9-6

You construct a handle for a specific function by preceding the function name with an @ sign. The syntax is:

```
h = @functionname
```

where *h* is the variable to which the returned function handle is assigned.

Use only the function *name*, with no path information, after the @ sign. If there is more than one function with this name, MATLAB associates with the handle the one function source it would dispatch to if you were actually calling the function.

Create a handle *h* for a function `plot` that is on your MATLAB path:

```
h = @plot;
```

Once you create a handle for a function, you can invoke the function by means of the handle instead of using the function name. Because the handle contains the absolute path to its function, you can invoke the function from any location that MATLAB is able to reach, as long as the program file for the function still exists at this location. This means that functions in one file can call functions that are not on the MATLAB path, subfunctions in a separate file, or even functions that are private to another folder, and thus not normally accessible to that caller.

Maximum Length of a Function Name

Function names used in handles are unique up to N characters, where N is the number returned by the function `namelengthmax`. If the function name exceeds that length, MATLAB truncates the latter part of the name.

For function handles created for Sun™ Java constructors, the length of any segment of the package name or class name must not exceed `namelengthmax` characters. (The term *segment* refers to any portion of the name that lies before, between, or after a dot. For example, `java.lang.String` has three segments). The overall length of the string specifying the package and class has no limit.

The Role of Scope, Precedence, and Overloading When Creating a Function Handle

At the time you create a function handle, MATLAB must decide exactly which function it is to associate the handle to. In doing so, MATLAB uses the same rules used to determine which file to invoke when you make a function call. To make this determination, MATLAB considers the following:

- **Scope** — The function named must be on the MATLAB path at the time the handle is constructed.
- **Precedence** — MATLAB selects which function(s) to associate the handle with, according to the function precedence rules described under *Determining Which Function Gets Called*.
- **Overloading** — If additional files on the path overload the function for any of the standard MATLAB classes, such as `double` or `char`, then MATLAB associates the handle with these files, as well.

Program files that overload a function for classes other than the standard MATLAB classes are not associated with the function handle at the time it is constructed. Function handles do operate on these types of overloaded functions, but MATLAB determines which implementation to call at the time of evaluation in this case.

Obtaining Permissions from Class Methods

When creating a function handle inside a method of a class, the function is resolved using the permissions of that method. When MATLAB invokes the function handle, it does so using the permissions of the class. This gives MATLAB the same access as the location where the function handle was created, including access to private and protected methods accessible to that class.

Example

This example defines two methods. One, `updateObj`, defines a listener for an event called `Update`, and the other, `callbackfcn`, responds to this event whenever it should occur. The latter function is a private function and thus would not normally be within the scope of the `notify` function. However, because `@callbackfcn` is actually a function handle, it retains the permissions of the context that created the function handle:

```
classdef updateObj < handle
    events
        Update
    end

    methods
        function obj = updateObj(varargin)
            addlistener(obj, 'Update', @callbackfcn);
            notify(obj, 'Update');
        end
    end

    methods (Access = private)
        function obj = callbackfcn(obj, varargin)
            disp('Object Updated')
            disp(obj);
        end
    end
end
```

To run this function, invoke `updateObj` at the MATLAB command line.

Using Function Handles for Anonymous Functions

Function handles also serve as the means of invoking anonymous functions. An anonymous function is a one-line expression-based MATLAB function that does not require a program file.

For example, the statement

```
sqr = @(x) x.^2;
```

creates an anonymous function that computes the square of its input argument x . The `@` operator makes `sqr` a function handle, giving you a means of calling the function:

```
sqr(20)
ans =
    400
```

Like nested functions, a handle to an anonymous function also stores all data that will be needed to resolve the handle when calling the function. Shares same issues as nested functions do.

See the documentation on “Anonymous Functions” on page 15-3 for more information.

Arrays of Function Handles

To create an array of function handles, you must use a cell array:

```
trigFun = {@sin, @cos, @tan};
```

For example, to plot the cosine of the range $-\pi$ to π at 0.01 intervals, use

```
plot(trigFun{2}(-pi:0.01:pi))
```


Calling a Function By Means of Its Handle

In this section...

“Calling Syntax” on page 9-7

“Calling a Function with Multiple Outputs” on page 9-8

“Returning a Handle for Use Outside of a Function File” on page 9-8

“Example — Using Function Handles in Optimization” on page 9-9

Function handles can give you access to functions you might not be able to execute. For instance, with a handle you can call a function even if it is no longer on your MATLAB path. You can also call a subfunction from outside of the file that defines that function.

Calling Syntax

The syntax for calling a function using a function handle is the same used when calling the function directly. For example, if you call function `myFun` like this:

```
[out1, out2, ...] = myFun(in1, in2, ...);
```

then you would call it using a handle in the same way, but using the handle name instead:

```
fHandle = @myFun;
[out1, out2, ...] = fHandle(in1, in2, ...);
```

There is one small difference. If the function being called takes no input arguments, then you must call the function with empty parentheses placed after the handle name. If you use only the handle name, MATLAB just identifies the name of the function:

<pre>% This identifies the handle. fHandle = @computer; fHandle ans = @computer</pre>	<pre>% This invokes the function. fHandle = @computer; fHandle() ans = PCWIN</pre>
--	---

Calling a Function with Multiple Outputs

The example below returns multiple values from a call to an anonymous function. Create anonymous function `f` that locates the nonzero elements of an array, and returns the row, column, and value of each element in variables `row`, `col`, and `val`:

```
f = @(X)find(X);
```

Call the function on matrix `m` using the function handle `f`. Because the function uses the MATLAB `find` function which returns up to three outputs, you can specify from 0 to 3 outputs in the call:

```
m = [3 2 0; -5 0 7; 0 0 1]
m =
     3     2     0
    -5     0     7
     0     0     1
```

```
[row col val] = f(m);
```

```
val
val =
     3
    -5
     2
     7
     1
```

Returning a Handle for Use Outside of a Function File

As stated previously, you can use function handles to call a function that may otherwise be hidden or out of scope. This example function `getHandle` returns a function handle `fHandle` to a caller that is outside of the file:

```
function fHandle = getHandle
fHandle = @subFun;

function res = subFun(t1, t2, varargin);
...
```

Call `getHandle` to obtain a function handle with which to invoke the subfunction. You can now call the subfunction as you would any function that is in scope:

```
f1 = getHandle;  
result = f1(startTime, endTime, procedure);
```

Example — Using Function Handles in Optimization

Function handles can be particularly useful in optimization work. If you have the MathWorks Optimization Toolbox™ installed, click on any of the links below to find information and examples on this topic:

- “Passing Extra Parameters” — Calling objective or constraint functions that have parameters in addition to the independent variable.
- “Anonymous Function Objectives” — Use function handles in writing simple objective functions.
- “Example: Nonlinear Curve Fitting with `lsqcurvefit`” — An example using `lsqcurvefit`, which takes two inputs for the objective.

Preserving Data from the Workspace

In this section...

“Preserving Data with Anonymous Functions” on page 9-10

“Preserving Data with Nested Functions” on page 9-11

Both anonymous functions and nested functions make use of variable data that is stored outside the body of the function itself. For example, the anonymous function shown here uses two variables: X and K . You pass the X variable into the anonymous function whenever you invoke the function. The value for K however is taken from the currently active workspace:

```
K = 200;  
fAnon = @(X)K * X;  
  
fAnon([2.54 1.43 0.68 1.90 1.02 2.13]);
```

What would happen if you tried to invoke this function after you cleared K from the workspace? Or if you saved the anonymous function to a `.mat` file and then loaded it into an entirely separate computing environment where K is not defined?

The answer is that MATLAB stores any values needed by an anonymous (or nested) function within the handle itself. It does this at the time you construct the handle. This does not include values from the argument list as these values get passed in whenever you call the function.

Preserving Data with Anonymous Functions

If you create an anonymous function at the MATLAB command window, that function has access to the workspace of your current MATLAB session. If you create the function inside of another function, then it has access to the outer function’s workspace. Either way, if your anonymous function depends upon variables from an outside workspace, then MATLAB stores the variables and their values within the function handle at the time the handle is created.

This example puts a 3-by-5 matrix into the base workspace, and then creates a function handle to an anonymous function that requires access to the matrix.

Create matrix A and anonymous function testAnon:

```
A = magic(5); A(4:5,:) = []
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
```

```
testAnon = @(x)x * A;           % Anonymous function
```

Call the anonymous function via its handle, passing in a multiplier value. This multiplies all elements by 5.2:

```
testAnon(5.2)
ans =
    88.4000   124.8000     5.2000    41.6000    78.0000
   119.6000    26.0000    36.4000    72.8000    83.2000
    20.8000    31.2000    67.6000   104.0000   114.4000
```

Clear variable A from the base workspace and verify that this action has no effect on the output of the anonymous function:

```
clear A

testAnon(5.2)
ans =
    88.4000   124.8000     5.2000    41.6000    78.0000
   119.6000    26.0000    36.4000    72.8000    83.2000
    20.8000    31.2000    67.6000   104.0000   114.4000
```

This works because the variable A and its value at the time the anonymous function is created is preserved within the function handle that also provides access to the anonymous function. See “Variables Used in the Expression” on page 15-8 for more information.

Preserving Data with Nested Functions

Nested functions are similar to anonymous functions in their ability to preserve workspace data needed by a function handle. In this case however, the workspace belongs to one of the functions inside of which the handle is being created. See “Variable Scope in Nested Functions” on page 15-19 and

“Using Function Handles with Nested Functions” on page 15-21 for more information on this subject.

This example shows a function `getHandles` that returns a handle to nested function `getApproxVal_V4`. The nested function uses two variables, `const` and `adjust`, from the workspace of the outer function. Calling `getHandles` creates a function handle to the nested function and also stores these two variables within that handle so that they will be available whenever the nested function is invoked:

```
function handle = getHandles(adjust)
const = 16.3;
handle = @getApproxVal_V4;

    function vOut = getApproxVal_V4(vectIn)
        vOut = ((vectIn+adjust)*const) + ((vectIn-adjust)*const);
    end
end
```

Call the `getHandles` function to obtain a handle to the nested function:

```
adjustVal = 0.023;
getApproxValue = getHandles(adjustVal);

getApproxValue([0.67 -0.09 1.01 0.33 -0.14 -0.23])
ans =
    21.8420    -2.9340    32.9260    10.7580    -4.5640    -7.4980
```

The documentation on “Examining a Function Handle” on page 9-20 explains how to see which variables are stored within a particular function handle. Another helpful resource is “Using Function Handles with Nested Functions” on page 15-21.

Loading a Saved Handle to a Nested Function

If you save a function handle to a nested function and, at some later date, modify the function and then reload the handle, you may observe unexpected behavior from the restored handle. when you invoke the function from the reloaded handle.

Applications of Function Handles

In this section...

“Example of Passing a Function Handle” on page 9-13

“Pass a Function to Another Function” on page 9-13

“Capture Data Values For Later Use By a Function” on page 9-15

“Call Functions Outside of Their Normal Scope” on page 9-18

“Save the Handle in a MAT-File for Use in a Later MATLAB Session” on page 9-18

Example of Passing a Function Handle

The following example creates a handle for a function supplied by MATLAB called `humps` and assigns it to the variable `h`. (The `humps` function returns a strong maxima near $x = 0.3$ and $x = 0.9$).

```
h = @humps;
```

After constructing the handle, you can pass it in the argument list of a call to some other function, as shown here. This example passes the function handle `h` that was just created as the first argument in a call to `fminbnd`. This function then minimizes over the interval `[0.3, 1]`.

```
x = fminbnd(h, 0.3, 1)
x =
    0.6370
```

Using a function handle enables you to pass different functions for `fminbnd` to use in determining its final result.

Pass a Function to Another Function

The ability to pass variables to a function enables you to run the function on different values. In the same way, you can pass function handles as input arguments to a function, thus allowing the called function to change the operations it runs on the input data.

Example 1 — Run quad on Varying Functions

Run the quadrature function on varying input functions:

```
a = 0; b = 5;

quad(@log, a, b)
ans =
    3.0472

quad(@sin, a, b)
ans =
    0.7163

quad(@humps, a, b)
ans =
    12.3566
```

Example 2 — Run quad on Anonymous Functions

Run quad on a MATLAB built-in function or an anonymous function:

```
n = quad(@log, 0, 3);

n = quad(@(x)x.^2, 0, 3);
```

Change the parameters of the function you pass to quad with a simple modification of the anonymous function that is associated with the function handle input:

```
a = 3.7;
z = quad(@(x)x.^a, 0, 3);
```

Example 3 — Compare quad Results on Different Functions

Compare the integral of the cosine function over the interval [a, b]:

```
a = 0; b = 10;
int1 = quad(@cos,a,b)

int1 =
```



```
-0.5440
```

with the integral over the same interval of the piecewise polynomial `pp` that approximates the cosine function by interpolating the computed values `x` and `y`:

```
x = a:b;  
y = cos(x);  
pp = spline(x,y);  
int2 = quad(@(x)ppval(pp,x), a, b)
```

```
int2 =  
-0.5485
```

Capture Data Values For Later Use By a Function

You can do more with a function handle than just create an association to a certain function. By using anonymous functions, you can also capture certain variables and their values from the function workspace and store them in the handle. These data values are stored in the handle at the time of its construction, and are contained within the handle for as long as it exists. Whenever you then invoke the function by means of its handle, MATLAB supplies the function with all variable inputs specified in the argument list of the function call, and also any constant inputs that were stored in the function handle at the time of its construction.

Storing some or all input data in a function handle enables you to reliably use the same set of data with that function regardless of where or when you invoke the handle. You can also interrupt your use of a function and resume it with the same data at a later time simply by saving the function handle to a MAT-file.

Example 1 – Constructing a Function Handle that Preserves Its Variables

Compare the following two ways of implementing a simple plotting function called `draw_plot`. The first case creates the function as one that you would call by name and that accepts seven inputs specifying coordinate and property information:

```
function draw_plot(x, y, lnSpec, lnWidth, mkEdge, mkFace, mkSize)
```

```
plot(x, y, lnSpec, ...
     'LineWidth', lnWidth, ...
     'MarkerEdgeColor', mkEdge, ...
     'MarkerFaceColor', mkFace, ...
     'MarkerSize', mkSize)
```

The second case implements `draw_plot` as an anonymous function to be called by a function handle, `h`. The `draw_plot` function has only two inputs now; the remaining five are specified only on a call to the handle constructor function, `get_plot_handle`:

```
function h = get_plot_handle(lnSpec, lnWidth, mkEdge, ...
                             mkFace, mkSize)
h = @draw_plot;
    function draw_plot(x, y)
        plot(x, y, lnSpec, ...
             'LineWidth', lnWidth, ...
             'MarkerEdgeColor', mkEdge, ...
             'MarkerFaceColor', mkFace, ...
             'MarkerSize', mkSize)
    end
end
```

Because these input values are required by the `draw_plot` function but are not made available in its argument list, MATLAB supplies them by storing them in the function handle for `draw_plot` at the time it is constructed. Construct the function handle `h`, also supplying the values to be stored in handle:

```
h = get_plot_handle('--rs', 2, 'k', 'g', 10);
```

Now call the function, specifying only the `x` and `y` inputs:

```
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
h(x, y) % Draw the plot
```

The later section on “Examining a Function Handle” on page 9-20 continues this example by showing how you can examine the contents of the function and workspace contents of this function handle.

Example 2 – Varying Data Values Stored in a Function Handle

Values stored within a handle to a nested function do not have to remain constant. The following function constructs and returns a function handle `h` to the anonymous function `addOne`. In addition to associating the handle with `addOne`, MATLAB also stores the initial value of `x` in the function handle:

```
function h = counter
x = 0;
h = @addOne;
    function y = addOne;
        x = x + 1;
        y = x;
    end
end
```

The `addOne` function that is associated with handle `h` increments variable `x` each time you call it. This modifies the value of the variable stored in the function handle:

```
h = counter;
h()
ans =
     1
h()
ans =
     2
```

Example 3 – You Cannot Vary Data in a Handle to an Anonymous Function

Unlike the example above, values stored within a handle to an anonymous function do remain constant. Construct a handle to an anonymous function that just returns the value of `x`, and initialize `x` to 300. The value of `x` within the function handle remains constant regardless of how you modify `x` external to the handle:

```
x = 300;
h = @( )x;

x = 50;
h()
```

```
ans =  
    300  
  
clear x  
h()  
ans =  
    300
```

Call Functions Outside of Their Normal Scope

By design, only functions within a program file are permitted to access subfunctions defined within that file. However, if, in this same file, you were to construct a function handle for one of the internal subfunctions, and then pass that handle to a variable that exists outside of the file, access to that subfunction would be essentially unlimited. By capturing the access to the subfunction in a function handle, and then making that handle available to functions external to the file (or to the command line), the example extends that scope. An example of this is shown in the preceding section, “Capture Data Values For Later Use By a Function” on page 9-15.

Private functions also have specific access rules that limit their availability with the MATLAB environment. But, as with subfunctions, MATLAB allows you to construct a handle for a private function. Therefore, you can call it by means of that handle from any location or even from the MATLAB command line, should it be necessary.

Save the Handle in a MAT-File for Use in a Later MATLAB Session

If you have one or more function handles that you would like to reuse in a later MATLAB session, you can store them in a MAT-file using the `save` function and then use `load` later on to restore them to your MATLAB workspace.

Saving and Loading Function Handles

You can save and load function handles in a MAT-file using the MATLAB `save` and `load` functions. If you load a function handle that you saved in an earlier MATLAB session, the following conditions could cause unexpected behavior:

- Any of the 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 does not exist or is located in a different folder than on the system on which the handle was saved.

In both of these cases, the function handle is now invalid because it is no longer associated with any existing function code. Although the handle is invalid, MATLAB still performs the load successfully and without displaying a warning. Attempting to invoke the handle, however, results in an error.

Invalid or Obsolete Function Handles

If you create a handle to a function that is not on the MATLAB path, or if you load a handle to a function that is no longer on the path, MATLAB catches the error only when the handle is invoked. You can assign an invalid handle and use it in such operations as `func2str`. MATLAB catches and reports an error only when you attempt to use it in a runtime operation.

Advanced Operations on Function Handles

In this section...

“Examining a Function Handle” on page 9-20

“Converting to and from a String” on page 9-21

“Comparing Function Handles” on page 9-23

Examining a Function Handle

Use the `functions` function to examine the contents of a function handle.

Caution MATLAB provides the `functions` function for querying and debugging purposes only. Because its behavior may change in subsequent releases, you should not rely upon it for programming purposes.

The following example is a continuation of an example in an earlier section of the Function Handles documentation. See Example 1 in the section “Capture Data Values For Later Use By a Function” on page 9-15 for the complete example.

Construct a function handle that contains both a function association, and data required by that function to execute. The following function constructs the function handle, `h`:

```
function h = get_plot_handle(lnSpec, lnWidth, mkEdge, ...
    mkFace, mkSize)
h = @draw_plot;
    function draw_plot(x, y)
        plot(x, y, lnSpec, ...
            'LineWidth', lnWidth, ...
            'MarkerEdgeColor', mkEdge, ...
            'MarkerFaceColor', mkFace, ...
            'MarkerSize', mkSize)
    end
end
```

Use functions to examine the contents of the returned handle:

```
f = functions(h)
f =
    function: 'get_plot_handle/draw_plot'
    type: 'nested'
    file: 'D:\matlab\work\get_plot_handle.m'
    workspace: {[1x1 struct]}
```

The call to `functions` returns a structure with four fields:

- `function` — Name of the function or subfunction to which the handle is associated. (Function names that follow a slash character (/) are implemented in the program code as subfunctions.)
- `type` — Type of function (e.g., simple, nested, anonymous)
- `file` — Filename and path to the file. (For built-in functions, this is the string 'MATLAB built-in function')
- `workspace` — Variables in the function workspace at the time the handle was constructed, along with their values

Examine the workspace variables that you saved in the function handle:

```
f.workspace{:}
ans =
    h: @get_plot_handle/draw_plot
    lnSpec: '--rs'
    lnWidth: 2
    mrkrEdge: 'k'
    mrkrFace: 'g'
    mrkrSize: 10
```

Converting to and from a String

Two functions, `str2func` and `func2str` enable you to convert between a string containing a function name and a function handle that is associated with that function name.

Converting a String to a Function Handle

Another means of creating a function handle is to convert a string that holds a function name to a handle for the named function. You can do this using the `str2func` function:

```
handle = str2func('functionname');
```

The example below takes the name of a function as the first argument. It compares part of the name to see if this is a polynomial function, converts the function string to a function handle if it is not, and then calls the function by means of its handle:

```
function run_function(funcname, arg1, arg2)
if strcmp(funcname, 'poly', 4)
    disp 'You cannot run polynomial functions on this data.'
    return
else
    h = str2func(funcname);
    h(arg1, arg2);
end
```

Note Nested functions are not accessible to `str2func`. To construct a function handle for a nested function, you must use the function handle constructor, `@`.

Converting a Function Handle to a String

You can also convert a function handle back into a string using the `func2str` function:

```
functionname = func2str(handle);
```

This example converts the function handle `h` to a string containing the function name, and then uses the function name in a message displayed to the user:

```
function call_h(h, arg1, arg2)
sprintf('Calling function %s ...\n', func2str(h))
h(arg1, arg2)
```


Comparing Function Handles

This section describes how MATLAB determines whether or not separate function handles are equal to each other:

- “Comparing Handles Constructed from a Named Function” on page 9-23
- “Comparing Handles to Anonymous Functions” on page 9-23
- “Comparing Handles to Nested Functions” on page 9-24
- “Comparing Handles Saved to a MAT-File” on page 9-25

Comparing Handles Constructed from a Named Function

MATLAB considers function handles that you construct from the same named function (e.g., `handle = @sin`) to be equal. The `isequal` function returns a value of `true` when comparing these types of handles:

```
func1 = @sin;
func2 = @sin;
isequal(func1, func2)
ans =
     1
```

If you save these handles to a MAT-file, and then load them back into the workspace later on, they are still equal.

Comparing Handles to Anonymous Functions

Unlike handles to named functions, any two function handles that represent the same anonymous function (i.e., handles to anonymous functions that contain the same text) are not equal. This is because MATLAB cannot guarantee that the frozen values of non-argument variables (such as `A`, below) are the same.

```
A = 5;
h1 = @(x)A * x.^2;
h2 = @(x)A * x.^2;

isequal(h1, h2)
ans =
     0
```

Note In general, MATLAB may underestimate the equality of function handles. That is, a test for equality may return `false` even when the functions happen to behave the same. But in cases where MATLAB does indicate equality, the functions are guaranteed to behave in an identical manner.

If you make a copy of an anonymous function handle, the copy and the original are equal:

```
h1 = @(x)A * x.^2;   h2 = h1;
isequal(h1, h2)
ans =
     1
```

Comparing Handles to Nested Functions

MATLAB considers function handles to the same nested function to be equal only if your code constructs these handles on the same call to the function containing the nested functions. Given this function that constructs two handles to the same nested function:

```
function [h1, h2] = test_eq(a, b, c)
h1 = @findZ;
h2 = @findZ;

    function z = findZ
        z = a.^3 + b.^2 + c';
    end
end
```

function handles constructed from the same nested function and on the same call to the parent function are considered equal:

```
[h1 h2] = test_eq(4, 19, -7);

isequal(h1, h2),
ans =
     1
```

while those constructed from different calls are not considered equal:

```
[q1 q2] = test_eq(3, -1, 2);  
  
isequal(h1, q1)  
ans =  
    0
```

Comparing Handles Saved to a MAT-File

If you save equivalent anonymous or nested function handles to separate MAT-files, and then load them back into the MATLAB workspace, they are no longer equal. This is because saving the function handle loses track of the original circumstances under which the function handle was created. Reloading it results in a function handle that compares as being unequal to the original function handle.

Create two equivalent anonymous function handles:

```
h1 = @(x) sin(x);  
h2 = h1;  
  
isequal(h1, h2)  
ans =  
    1
```

Save each to a different MAT-file:

```
save fname1 h1;  
save fname2 h2;
```

Clear the MATLAB workspace, and then load the function handles back into the workspace:

```
clear all  
load fname1  
load fname2
```

The function handles are no longer considered equal:

```
isequal(h1, h2)  
ans =
```

0

Note, however, that equal anonymous and nested function handles that you save to the same MAT-file are equal when loaded back into MATLAB.

Functions That Operate on Function Handles

MATLAB provides the following functions for working with function handles. See the reference pages for these functions for more information.

Function	Description
<code>functions</code>	Return information describing a function handle.
<code>func2str</code>	Construct a function name string from a function handle.
<code>str2func</code>	Construct a function handle from a function name string.
<code>save</code>	Save a function handle from the current workspace to a MAT-file.
<code>load</code>	Load a function handle from a MAT-file into the current workspace.
<code>isa</code>	Determine if a variable contains a function handle.
<code>isequal</code>	Determine if two function handles are handles to the same function.

Map Containers

- “Overview of the Map Data Structure” on page 10-2
- “Description of the Map Class” on page 10-4
- “Creating a Map Object” on page 10-6
- “Examining the Contents of the Map” on page 10-9
- “Reading and Writing Using a Key Index” on page 10-11
- “Modifying Keys and Values in the Map” on page 10-15
- “Mapping to Different Value Types” on page 10-18

Overview of the Map Data Structure

A *Map* is a type of fast key lookup data structure that offers a flexible means of indexing into its individual elements. Unlike most array data structures in the MATLAB software that only allow access to the elements by means of integer indices, the indices for a Map can be nearly any scalar numeric value or a character string.

Indices into the elements of a Map are called *keys*. These keys, along with the data *values* associated with them, are stored within the Map. Each entry of a Map contains exactly one unique key and its corresponding value. Indexing into the Map of rainfall statistics shown below with a string representing the month of August yields the value internally associated with that month, 37.3.

	KEYS	VALUES	
	Jan	327.2	
	Feb	368.2	
	Mar	197.6	
	Apr	178.4	
	May	100.0	
	Jun	69.9	
	Jul	32.3	
Aug →	Aug	37.3	→ 37.3
	Sep	19.0	
	Oct	37.0	
	Nov	73.2	
	Dec	110.9	
	Annual	1551.0	

Mean monthly rainfall statistics (mm)

Keys are not restricted to integers as they are with other arrays. Specifically, a key may be any of the following types:

- 1-by-N character array
- Scalar real double or single

- Signed or unsigned scalar integer

The values stored in a Map can be of any type. This includes arrays of numeric values, structures, cells, strings, objects, or other Maps.

Note A Map is most memory efficient when the data stored in it is a scalar number or a character array.

Description of the Map Class

In this section...

“Properties of the Map Class” on page 10-4

“Methods of the Map Class” on page 10-5

A Map is actually an object, or instance, of a MATLAB class called Map. It is also a handle object and, as such, it behaves like any other MATLAB handle object. This section gives a brief overview of the Map class. For more details, see the `containers.Map` reference page.

Properties of the Map Class

All objects of the Map class have three properties. You cannot write directly to any of these properties; you can change them only by means of the methods of the Map class.

Property	Description	Default
Count	Unsigned 64-bit integer that represents the total number of key/value pairs contained in the Map object.	0
KeyType	String that indicates the type of all keys contained in the Map object. KeyType can be any of the following: <code>double</code> , <code>single</code> , <code>char</code> , and signed or unsigned 32-bit or 64-bit integer. If you attempt to add keys of an unsupported type, <code>int8</code> for example, MATLAB makes them <code>double</code> .	<code>char</code>
ValueType	String that indicates the type of values contained in the Map object. If the values in a Map are all scalar numbers of the same type, ValueType is set to that type. If the values are all character arrays, ValueType is <code>'char'</code> . Otherwise, ValueType is <code>'any'</code> .	<code>any</code>

To examine one of these properties, follow the name of the Map object with a dot and then the property name. For example, to see what type of keys are used in Map `mapObj`, use

```
mapObj.KeyType
```

A Map is a handle object. As such, if you make a copy of the object, MATLAB does not create a new Map; it creates a new handle for the existing Map that you specify. If you alter the Map's contents in reference to this new handle, MATLAB applies the changes you make to the original Map as well. You can, however, delete the new handle without affecting the original Map.

Methods of the Map Class

The Map class implements the following methods. Their use is explained in the later sections of this documentation and also in the function reference pages.

Method	Description
<code>isKey</code>	Check if Map contains specified key
<code>keys</code>	Names of all keys in Map
<code>length</code>	Length of Map
<code>remove</code>	Remove key and its value from Map
<code>size</code>	Dimensions of Map
<code>values</code>	Values contained in Map

Creating a Map Object

In this section...

“Constructing an Empty Map Object” on page 10-6

“Constructing An Initialized Map Object” on page 10-7

“Combining Map Objects” on page 10-8

A Map is an object of the Map class. It is defined within a MATLAB package called `containers`. As with any class, you use its constructor function to create any new instances of it. You must include the package name when calling the constructor:

```
newMap = containers.Map(optional_keys_and_values)
```

Constructing an Empty Map Object

When you call the Map constructor with no input arguments, MATLAB constructs an empty Map object. When you do not end the command with a semicolon, MATLAB displays the following information about the object you have constructed:

```
newMap = containers.Map()
newMap =
  containers.Map handle
  Package: containers

Properties:
  Count: 0
  KeyType: 'char'
  ValueType: 'any'
Methods, Events, Superclasses
```

The properties of an empty Map object are set to their default values:

- Count = 0
- KeyType = 'char'
- ValueType = 'any'

Once you construct the empty Map object, you can use the `keys` and `values` methods to populate it. For a summary of MATLAB functions you can use with a Map object, see “Methods of the Map Class” on page 10-5

Constructing An Initialized Map Object

Most of the time, you will want to initialize the Map with at least some keys and values at the time you construct it. You can enter one or more sets of keys and values using the syntax shown here. The brace operators (`{}`) are not required if you enter only one key/value pair:

```
mapObj = containers.Map({key1, key2, ...}, {val1, val2, ...});
```

For those keys and values that are character strings, be sure that you specify them enclosed within single quotation marks. For example, when constructing a Map that has character string keys, use

```
mapObj = containers.Map(...
    {'keystr1', 'keystr2', ...}, {val1, val2, ...});
```

As an example of constructing an initialized Map object, create a new Map for the following key/value pairs taken from the monthly rainfall map shown earlier in this section.

KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

```
k = {'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', ...
     'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec', 'Annual'};

v = {327.2, 368.2, 197.6, 178.4, 100.0, 69.9, ...
     32.3, 37.3, 19.0, 37.0, 73.2, 110.9, 1551.0};

rainfallMap = containers.Map(k, v)
rainfallMap =
  containers.Map handle
  Package: containers

Properties:
  Count: 13
  KeyType: 'char'
  ValueType: 'double'
  Methods, Events, Superclasses
```

The Count property is now set to the number of key/value pairs in the Map, 13, the KeyType is char, and the ValueType is double.

Combining Map Objects

You can combine Map objects vertically using concatenation. However, the result is not a vector of Maps, but rather a single Map object containing all key/value pairs of the contributing Maps. Horizontal vectors of Maps are not allowed. See “Building a Map with Concatenation” on page 10-13, below.

Examining the Contents of the Map

Each entry in a Map consists of two parts: a unique key and its corresponding value. To find all the keys in a Map, use the `keys` method. To find all of the values, use the `values` method.

Create a new Map called `tickets` that maps airline ticket numbers to the holders of those tickets. Construct the Map with four key/value pairs:

```
ticketMap = containers.Map(...
    {'2R175', 'B7398', 'A479GY', 'NZ1452'}, ...
    {'James Enright', 'Carl Haynes', 'Sarah Latham', ...
     'Bradley Reid'});
```

Use the `keys` method to display all keys in the Map. MATLAB lists keys of type `char` in alphabetical order, and keys of any numeric type in numerical order:

```
keys(ticketMap)
ans =
    '2R175'    'A479GY'    'B7398'    'NZ1452'
```

Next, display the values that are associated with those keys in the Map. The order of the values is determined by the order of the keys associated with them.

This table shows the keys listed in alphabetical order:

keys	values
2R175	James Enright
A479GY	Sarah Latham
B7398	Carl Haynes
NZ1452	Bradley Reid

The `values` method uses the same ordering of values:

```
values(ticketMap)
ans =
```

'James Enright' 'Sarah Latham' 'Carl Haynes' 'Bradley Reid'

Reading and Writing Using a Key Index

In this section...

“Reading From the Map” on page 10-11

“Adding Key/Value Pairs” on page 10-12

“Building a Map with Concatenation” on page 10-13

When reading from the Map, use the same keys that you have defined and associated with particular values. Writing new entries to the Map requires that you supply the values to store with a key for each one .

Note For a large Map, the keys and value methods use a lot of memory as their outputs are cell arrays.

Reading From the Map

After you have constructed and populated your Map, you can begin to use it to store and retrieve data. You use a Map in the same manner that you would an array, except that you are not restricted to using integer indices. The general syntax for looking up a value (`valueN`) for a given key (`keyN`) is shown here. If the key is a character string, enclose it in single quotation marks:

```
valueN = mapObj(keyN);
```

You can find any single value by indexing into the map with the appropriate key:

```
passenger = ticketMap('2R175')
passenger =
    James Enright
```

Find the person who holds ticket A479GY:

```
sprintf('    Would passenger %s please come to the desk?\n', ...
    ticketMap('A479GY'))
ans =
    Would passenger Sarah Latham please come to the desk?
```

To access the values of multiple keys, use the `values` method, specifying the keys in a cell array:

```
values(ticketMap, {'2R175', 'B7398'})
ans =
    'James Enright'    'Carl Haynes'
```

Map containers support scalar indexing only. You cannot use the colon operator to access a range of keys as you can with other MATLAB classes. For example, the following statements throw an error:

```
ticketMap('2R175':'B7398')
ticketMap(:)
```

Adding Key/Value Pairs

Unlike other array types, each entry in a Map consists of two items: the value and its key. When you write a new value to a Map, you must supply its key as well. This key must be consistent in type with any other keys in the Map.

Use the following syntax to insert additional elements into a Map:

```
existingMapObj(newKeyName) = newValue;
```

Add two more entries to the `ticketMap` used in the above examples, Verify that the Map now has five key/value pairs:

```
ticketMap('947F4') = 'Susan Spera';
ticketMap('417R93') = 'Patricia Hughes';
```

```
ticketMap.Count
ans =
    6
```

List all of the keys and values in Map `ticketMap`:

```
keys(ticketMap), values(ticketMap)
ans =
    '2R175'    '417R93'    '947F4'    'A479GY'    'B7398'    'NZ1452'
ans =
    Columns 1 through 3
```

```
'James Enright'   'Patricia Hughes'   'Susan Spera'
Columns 4 through 6
'Sarah Latham'   'Carl Haynes'   'Bradley Reid'
```

Building a Map with Concatenation

You can add key/value pairs to a Map in groups using concatenation. The concatenation of Map objects is different from other classes. Instead of building a vector of s, MATLAB returns a single Map containing the key/value pairs from each of the contributing Map objects.

Rules for the concatenation of Map objects are:

- Only vertical vectors of Map objects are allowed. You cannot create an m-by-n array or a horizontal vector of s. For this reason, `vertcat` is supported for Map objects, but not `horzcat`.
- All keys in each map being concatenated must be of the same class.
- You can combine Maps with different numbers of key/value pairs. The result is a single Map object containing key/value pairs from each of the contributing maps:

```
tMap1 = containers.Map({'2R175', 'B7398', 'A479GY'}, ...
    {'James Enright', 'Carl Haynes', 'Sarah Latham'});

tMap2 = containers.Map({'417R93', 'NZ1452', '947F4'}, ...
    {'Patricia Hughes', 'Bradley Reid', 'Susan Spera'});

% Concatenate the two maps:
ticketMap = [tMap1; tMap2];
```

The result of this concatenation is the same 6-element map that was constructed in the previous section:

```
ticketMap.Count
ans =
    6

keys(ticketMap), values(ticketMap)
ans =
    '2R175'   '417R93'   '947F4'   'A479GY'   'B7398'   'NZ1452'
```

```
ans =  
  Columns 1 through 3  
    'James Enright'    'Patricia Hughes'    'Susan Spera'  
  Columns 4 through 6  
    'Sarah Latham'    'Carl Haynes'    'Bradley Reid'
```

- Concatenation does not include duplicate keys or their values in the resulting Map object.

In the following example, both objects `m1` and `m2` use a key of 8. In Map `m1`, 8 is a key to value C; in `m2`, it is a key to value X:

```
m1 = containers.Map({1, 5, 8}, {'A', 'B', 'C'});  
m2 = containers.Map({8, 9, 6}, {'X', 'Y', 'Z'});
```

Combine `m1` and `m2` to form a new Map object, `m`:

```
m = [m1; m2];
```

The resulting Map object `m` has only five key/value pairs. The value C was dropped from the concatenation because its key was not unique:

```
keys(m), values(m)  
ans =  
    [1]    [5]    [6]    [8]    [9]  
ans =  
    'A'    'B'    'Z'    'X'    'Y'
```

Modifying Keys and Values in the Map

In this section...

“Removing Keys and Values from the Map” on page 10-15

“Modifying Values” on page 10-15

“Modifying Keys” on page 10-16

“Modifying a Copy of the Map” on page 10-16

Note Keep in mind that if you have more than one handle to a Map, modifying the handle also makes changes to the original Map. See “Modifying a Copy of the Map” on page 10-16, below.

Removing Keys and Values from the Map

Use the `remove` method to delete any entries from a Map. When calling this method, specify the Map object name and the key name to remove. MATLAB deletes the key and its associated value from the Map.

The syntax for the `remove` method is

```
remove('mapName', 'keyname');
```

Remove one entry (the specified key and its value) from the Map object:

```
remove(ticketMap, 'NZ1452');
values(ticketMap)
ans =
    Columns 1 through 3
    'James Enright'    'Patricia Hughes'    'Susan Spera'
    Columns 4 through 5
    'Sarah Latham'    'Carl Haynes'
```

Modifying Values

You can modify any value in a Map simply by overwriting the current value. The passenger holding ticket A479GY is identified as Sarah Latham:

```
ticketMap('A479GY')
ans =
    Sarah Latham
```

Change the passenger's first name to Anna Latham by overwriting the original value for the A479GY key:

```
ticketMap('A479GY') = 'Anna Latham';
```

Verify the change:

```
ticketMap('A479GY')
ans =
    'Anna Latham';
```

Modifying Keys

To modify an existing key while keeping the value the same, first remove both the key and its value from the Map. Then create a new entry, this time with the corrected key name.

Modify the ticket number belonging to passenger James Enright:

```
remove(ticketMap, '2R175');
ticketMap('2S185') = 'James Enright';

k = keys(ticketMap); v = values(ticketMap);
str1 = '    '%s'' has been assigned a new\n';
str2 = '    ticket number: %s.\n';

fprintf(str1, v{1})
fprintf(str2, k{1})
```

```
'James Enright' has been assigned a new
    ticket number: 2S185.
```

Modifying a Copy of the Map

Because `ticketMap` is a handle object, you need to be careful when making copies of the Map. Keep in mind that by copying a Map object, you are really

just creating another handle to the same object. Any changes you make to this handle are also applied to the original Map.

Make a copy of Map `ticketMap`. Write to this copy, and notice that the change is applied to the original Map object itself:

```
copiedMap = ticketMap;

copiedMap('AZ12345') = 'unidentified person';
ticketMap('AZ12345')
ans =
    unidentified person
```

Clean up:

```
remove(ticketMap, 'AZ12345');
clear copiedMap;
```

Mapping to Different Value Types

In this section...

“Mapping to a Structure Array” on page 10-18

“Mapping to a Cell Array” on page 10-19

It is fairly common to store other classes, such as structures or cell arrays, in a Map structure. However, Maps are most memory efficient when the data stored in them belongs to one of the basic MATLAB types such as double, char, integers, and logicals.

Mapping to a Structure Array

The following example maps airline seat numbers to structures that contain information on who occupies the seat. To start out, create the following structure array:

```
s1.ticketNum = '2S185'; s1.destination = 'Barbados';
s1.reserved = '06-May-2008'; s1.origin = 'La Guardia';
s2.ticketNum = '947F4'; s2.destination = 'St. John';
s2.reserved = '14-Apr-2008'; s2.origin = 'Oakland';
s3.ticketNum = 'A479GY'; s3.destination = 'St. Lucia';
s3.reserved = '28-Mar-2008'; s3.origin = 'JFK';
s4.ticketNum = 'B7398'; s4.destination = 'Granada';
s4.reserved = '30-Apr-2008'; s4.origin = 'JFK';
s5.ticketNum = 'NZ1452'; s5.destination = 'Aruba';
s5.reserved = '01-May-2008'; s5.origin = 'Denver';
```

Map five of the seats to one of these structures:

```
seatingMap = containers.Map( ...
    {'23F', '15C', '15B', '09C', '12D'}, ...
    {s5, s1, s3, s4, s2});
```

Using this Map object, find information about the passenger, who has reserved seat 09C:

```
seatingMap('09C')
ans =
```



```

        ticketNum: 'B7398'
    destination: 'Granada'
        reserved: '30-Apr-2008'
        origin: 'JFK'

```

```

seatingMap('15B').ticketNum
ans =
    A479GY

```

Using two Maps together, you can find out the name of the person who has reserved the seat:

```

passenger = ticketMap(seatingMap('15B').ticketNum)
passenger =
    Anna Latham

```

Mapping to a Cell Array

As with structures, you can also map to a cell array in a Map object. Continuing with the airline example of the previous sections, some of the passengers on the flight have “frequent flyer” accounts with the airline. Map the names of these passengers to records of the number of miles they have used and the number of miles they still have available:

```

accountMap = containers.Map( ...
    {'Susan Spera', 'Carl Haynes', 'Anna Latham'}, ...
    {{247.5, 56.1}, {0, 1342.9}, {24.6, 314.7}});

```

Use the Map to retrieve account information on the passengers:

```

name = 'Carl Haynes';
acct = accountMap(name);

fprintf('%s has used %.1f miles on his/her account,\n', ...
    name, acct{1})
fprintf(' and has %.1f miles remaining.\n', acct{2})

```

```

Carl Haynes has used 0.0 miles on his/her account,
    and has 1342.9 miles remaining.

```


Combining Unlike Classes

- “Valid Combinations of Unlike Classes” on page 11-2
- “Combining Unlike Integer Types” on page 11-3
- “Combining Integer and Noninteger Data” on page 11-6
- “Combining Cell Arrays with Non-Cell Arrays” on page 11-7
- “Empty Matrices” on page 11-8
- “Concatenation Examples” on page 11-9

Valid Combinations of Unlike Classes

Matrices and arrays can be composed of elements of most any MATLAB data type as long as all elements in the matrix are of the same type. If you do include elements of unlike classes when constructing a matrix, MATLAB converts some elements so that all elements of the resulting matrix are of the same type. (See Chapter 3, “Overview of MATLAB Classes” for information on any of the MATLAB classes discussed here.)

Data type conversion is done with respect to a preset precedence of classes. The following table shows the five classes you can concatenate with an unlike type without generating an error (that is, with the exception of character and logical).

TYPE	character	integer	single	double	logical
character	character	character	character	character	invalid
integer	character	integer	integer	integer	integer
single	character	integer	single	single	single
double	character	integer	single	double	double
logical	invalid	integer	single	double	logical

For example, concatenating a double and single matrix always yields a matrix of type single. MATLAB converts the double element to single to accomplish this.

More About

- “Combining Unlike Integer Types” on page 11-3
- “Combining Integer and Noninteger Data” on page 11-6
- “Combining Cell Arrays with Non-Cell Arrays” on page 11-7
- “Concatenation Examples” on page 11-9

Combining Unlike Integer Types

In this section...

“Overview” on page 11-3

“Example of Combining Unlike Integer Sizes” on page 11-4

“Example of Combining Signed with Unsigned” on page 11-4

Overview

If you combine different integer types in a matrix (e.g., signed with unsigned, or 8-bit integers with 16-bit integers), MATLAB returns a matrix in which all elements are of one common type. MATLAB sets all elements of the resulting matrix to the data type of the left-most element in the input matrix. For example, the result of the following concatenation is a vector of three 16-bit signed integers:

```
A = [int16(450) uint8(250) int32(1000000)]
```

MATLAB also displays a warning to inform you that the result may not be what you had expected:

```
A = [int16(450) uint8(250) int32(1000000)];  
Warning: Concatenation with dominant (left-most) integer class  
may overflow other operands on conversion to return class.
```

You can disable this warning by entering the following two commands directly after the operation that caused the warning. The first command retrieves the message identifier associated with the most recent warning issued by MATLAB. The second command uses this identifier to disable any further warnings of that type from being issued:

```
[msg, intcat_msgid] = lastwarn;  
warning('off', intcat_msgid);
```

To reenble the warning so that it will now be displayed, use

```
warning('on', intcat_msgid);
```

You can use these commands to disable or enable the display of any MATLAB warning.

Example of Combining Unlike Integer Sizes

After disabling the integer concatenation warnings as shown above, concatenate the following two numbers once, and then switch their order. The return value depends on the order in which the integers are concatenated. The left-most type determines the data type for all elements in the vector:

```
A = [int16(5000) int8(50)]
A =
    5000    50
```

```
B = [int8(50) int16(5000)]
B =
    50   127
```

The first operation returns a vector of 16-bit integers. The second returns a vector of 8-bit integers. The element `int16(5000)` is set to 127, the maximum value for an 8-bit signed integer.

The same rules apply to vertical concatenation:

```
C = [int8(50); int16(5000)]
C =
    50
   127
```

Note You can find the maximum or minimum values for any MATLAB integer type using the `intmax` and `intmin` functions. For floating-point types, use `realmax` and `realmin`.

Example of Combining Signed with Unsigned

Now do the same exercise with signed and unsigned integers. Again, the left-most element determines the data type for all elements in the resulting matrix:

```
A = [int8(-100) uint8(100)]
A =
   -100    100
```

```
B = [uint8(100) int8(-100)]
B =
    100     0
```

The element `int8(-100)` is set to zero because it is no longer signed.

MATLAB evaluates each element *prior to* concatenating them into a combined array. In other words, the following statement evaluates to an 8-bit signed integer (equal to 50) and an 8-bit unsigned integer (unsigned -50 is set to zero) before the two elements are combined. Following the concatenation, the second element retains its zero value but takes on the unsigned `int8` type:

```
A = [int8(50), uint8(-50)]
A =
    50     0
```

Combining Integer and Noninteger Data

If you combine integers with `double`, `single`, or `logical` classes, all elements of the resulting matrix are given the data type of the left-most integer. For example, all elements of the following vector are set to `int32`:

```
A = [true pi int32(1000000) single(17.32) uint8(250)]
```


Combining Cell Arrays with Non-Cell Arrays

Combining a number of arrays in which one or more is a cell array returns a new cell array. Each of the original arrays occupies a cell in the new array:

```
A = [100, {uint8(200), 300}, 'MATLAB'];  
whos A
```

Name	Size	Bytes	Class	Attributes
A	1x4	477	cell	

Each element of the combined array maintains its original class:

```
fprintf('Classes: %s %s %s %s\n',...  
       class(A{1}),class(A{2}),class(A{3}),class(A{4}))
```

```
Classes: double uint8 double char
```

Empty Matrices

If you construct a matrix using empty matrix elements, the empty matrices are ignored in the resulting matrix:

```
A = [5.36; 7.01; []; 9.44]
```

```
A =
```

```
5.3600
```

```
7.0100
```

```
9.4400
```

Concatenation Examples

In this section...

“Combining Single and Double Types” on page 11-9

“Combining Integer and Double Types” on page 11-9

“Combining Character and Double Types” on page 11-10

“Combining Logical and Double Types” on page 11-10

Combining Single and Double Types

Combining `single` values with `double` values yields a `single` matrix. Note that 5.73×10^{300} is too big to be stored as a `single`, thus the conversion from `double` to `single` sets it to infinity. (The `class` function used in this example returns the data type for the input value).

```
x = [single(4.5) single(-2.8) pi 5.73*10^300]
x =
    4.5000    -2.8000    3.1416         Inf

class(x)           % Display the data type of x
ans =
    single
```

Combining Integer and Double Types

Combining integer values with `double` values yields an integer matrix. Note that the fractional part of `pi` is rounded to the nearest integer. (The `int8` function used in this example converts its numeric argument to an 8-bit integer).

```
x = [int8(21) int8(-22) int8(23) pi 45/6]
x =
    21    -22    23     3     7

class(x)
ans =
    int8
```

Combining Character and Double Types

Combining character values with double values yields a character matrix. MATLAB converts the double elements in this example to their character equivalents:

```
x = ['A' 'B' 'C' 68 69 70]
x =
  ABCDEF
```

```
class(x)
ans =
  char
```

Combining Logical and Double Types

Combining logical values with double values yields a double matrix. MATLAB converts the logical true and false elements in this example to double:

```
x = [true false false pi sqrt(7)]
x =
  1.0000         0         0   3.1416   2.6458
```

```
class(x)
ans =
  double
```

Using Objects

- “MATLAB Objects” on page 12-2
- “General Purpose Vs. Specialized Arrays” on page 12-5
- “Key Object Concepts” on page 12-8
- “Creating Objects” on page 12-11
- “Accessing Object Data” on page 12-14
- “Calling Object Methods” on page 12-16
- “Desktop Tools Are Object Aware” on page 12-19
- “Getting Information About Objects” on page 12-21
- “Copying Objects” on page 12-26
- “Destroying Objects” on page 12-33

MATLAB Objects

In this section...
“Getting Oriented” on page 12-2
“Getting Comfortable with Objects” on page 12-2
“What Are Objects and Why Use Them?” on page 12-2
“Accessing Objects” on page 12-3
“Objects In the MATLAB Language” on page 12-3
“Other Kinds of Objects Used by MATLAB” on page 12-4

Getting Oriented

This chapter provides information for people using objects. It does not provide a thorough treatment of object-oriented concepts, but instead focuses on what you need to know to use the objects provided with MATLAB.

If you are interested in object-oriented programming in the MATLAB language, see *Object-Oriented Programming*. For background information on objects, see object-oriented design.

Getting Comfortable with Objects

MATLAB uses objects because they are a convenient way to package data. Working with objects in MATLAB is like working with any variables and is often more convenient because objects are optimized for specific purposes. Think of an object as a neatly packaged collection of data that includes functions that operate on the data. The documentation for any particular object describes how to use it.

What Are Objects and Why Use Them?

In the simplest sense, objects are special-purpose variables that have a specific set of operations that you can perform on the data they contain. You do not need to know how the operations are implemented or how the data is stored. This makes objects modular and easy to pass within application

programs. It also isolates your code from changes to the object's design and implementation.

In a more general sense, objects are organized collections of data and functions that have been designed for specific purposes. For example, an object might be designed to contain time series data that consists of value/time-sample pairs and associated information like units, sample uniformity, and so on. This object could have a set of specific operations designed to perform analysis, such as filtering, interpolating, and plotting. The following sections provide examples of such objects.

Accessing Objects

You access an object with its variable name. Interacting with objects variables in MATLAB software is really no different from interacting with any other variables. Basically, you can perform the same common operations on variables whether they hold numbers or specialized objects. For example, you can do the following things with objects:

- Create it and assign a variable name so you can reference it again
- Assign or reassign data to it (see “Accessing Object Data” on page 12-14)
- Operate on its data (see “Calling Object Methods” on page 12-16)
- Convert it to another class (if this operation is supported by the object's class)
- Save it to a MAT-file so you can reload it later (see `save`)
- Copy it (see “Copying Objects” on page 12-26)
- Clear it from the workspace (`clear`)

Any object can have restrictions on how you create it, access its data, or what operations you can perform on it. Refer to the documentation for the particular MATLAB object for a description of what you can do with that object.

Objects In the MATLAB Language

The MATLAB language uses many specialized objects. For example, `MException` objects capture information when errors occur, `timer` objects execute code at a certain time interval, the `serial` object enables you to

communicate with devices connected to your computer's serial port, and so on. MATLAB toolboxes often define objects to manage the specific data and analyses performed by the toolbox.

All of these objects are designed to provide specific functionality that is not as conveniently available from general purpose language components.

Other Kinds of Objects Used by MATLAB

The MATLAB language enables you to use other kinds of objects in your MATLAB programs. The following objects are different from the MATLAB objects described in this documentation. See the individual sections referenced below for information on using these objects.

- Handle Graphics® objects represent objects used to create graphs and GUIs. These objects provide a `set/get` interface to property values, but are not extensible by subclassing. See “Handle Graphics Objects” for more information.
- Sun Java objects can be used in MATLAB code enabling you to access the capabilities of Java classes. See “Using Java Libraries from MATLAB” for more information.
- Microsoft COM objects enable you to integrate these software components into your application. See “Using COM Objects from MATLAB ” for more information.
- Microsoft .NET objects enable you to integrate .NET assemblies into your application. See “Using .NET Libraries from MATLAB” for more information.
- User-defined MATLAB objects created prior to Version 7.6 used different syntax for class definition (no `classdef` block) and exhibit other differences. See “Compatibility with Previous Versions ” for more information.

General Purpose Vs. Specialized Arrays

In this section...

“How They Differ” on page 12-5

“Using General-Purpose Data Structures” on page 12-5

“Using Specialized Objects” on page 12-6

How They Differ

The MATLAB language enables you to use both general-purpose and specialized arrays. For example, numeric multidimensional arrays, `struct`, and `cell` arrays provide general-purpose data storage. You typically extract data from the array and pass this data to functions (e.g., to perform mathematical analysis). Then, you store the data back in general-purpose arrays.

When using a specialized object, you typically pass the object’s data to a function that creates the object. Once you have created the object, you use specially defined functions to operate on the data. These functions are unique to the object and are designed specifically for the type and structure of the data contained by the object.

Using General-Purpose Data Structures

A commonly used general-purpose data structure references data via fieldnames. For example, these statements create a MATLAB `struct` (a MATLAB structure array):

```
s.Data = rand(10,1);  
s.Time = .01:.01:.1;  
s.Name = 'Data1';  
s.Units = 'seconds';
```

The structure `s` contains two arrays of numbers. However, `s` is a generic type in the sense that MATLAB does not define special functions to operate on the data in this particular structure. For example, while `s` contains two fields that would be useful to plot, `Data` and `Time`, you cannot pass `s` to the `plot` function:

```
plot(s)
Error using plot
Not enough input arguments.
```

While `s` has enough information to create a plot of `Data` versus `Time`, `plot` cannot access this data because structures like `s` can contain any values in its fields and the fields can have any name. Just because one field is named `Data` does not force you to assign data to that field.

To plot the data in `s`, you have to extract the data from the fields, pass them as numeric arrays in the desired order to the plot function, add a title, labels, and so on:

```
plot(s.Time,s.Data)
title(['Time Series Plot: ' s.Name])
xlabel(['Time (' s.Units ')'])
ylabel(s.Name)
```

You could create a function to perform these steps for you. Other programs using the structure `s` would need to create their own functions or access the one you created.

Using Specialized Objects

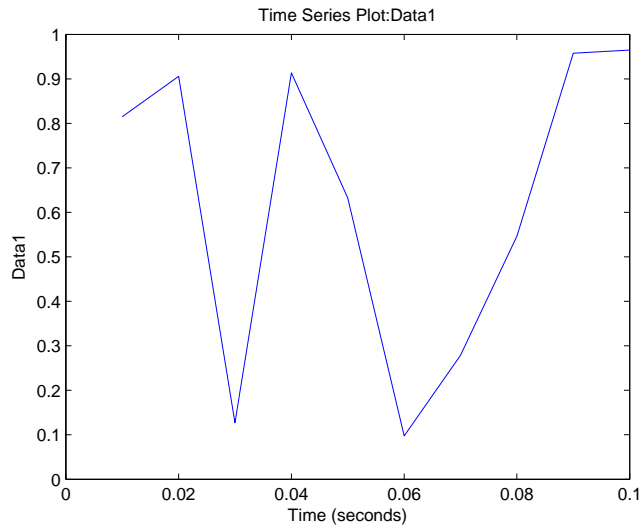
Compare the array `s` above to an object that you have designed specifically to contain and manipulate time series data. For example, the following statement creates a MATLAB `timeseries` object. It is initialized to store the same data as the structure `s` above:

```
tsobj = timeseries(rand(10,1),.01:.01:.1,'Name','Data1');
```

The function that creates the object `tsobj`, accepts sample data, sample times, a property name/property value pair (`Name/Data1`), and uses a default value of `Units` (which is seconds).

The designer of this object created a special version of the `plot` function that works directly with this object. For example:

```
plot(tsobj)
```



Notice how the object's `plot` function creates a graph that is plotted and labeled with the data from the `tsobj` object. As a user of this object, you do not need write your own code to produce this graph. The class design specifies the standard way to present graphs of `timeseries` data and all clients of this object use the same code for plotting.

See "Time Series Objects" for more on using MATLAB `timeseries` objects.

Key Object Concepts

In this section...
“Basic Concepts” on page 12-8
“Classes Describe How to Create Objects” on page 12-8
“Properties Contain Data” on page 12-9
“Methods Implement Operations” on page 12-9
“Events are Notices Broadcast to Listening Objects” on page 12-10

Basic Concepts

There are some basic concepts that are fundamental to objects. Objects represent something in the real world, like an error condition or the set of data you collected in a product test. Objects enable you to do something useful, like provide an error report or analyze and present the results of tests.

There are basic components that MATLAB uses to realize the design of an object. These components include:

- Classes
- Properties
- Methods
- Events

Classes Describe How to Create Objects

A *class* defines a set of similar objects. It is a description from which MATLAB creates a particular instance of the class, and it is the instance (that is, the object) that contains actual data. Therefore, while there is a `timeseries` class, you work with `timeseries` objects.

Classes are defined in code files — either as separate `.m` files or built-in to the MATLAB executable. Objects are specific representations of a class that you access through workspace variables.

Properties Contain Data

Objects store data in *properties*. Consider a `timeseries` object as an example. `timeseries` object properties contain time series data, corresponding time values, and related information, such as units, events, data quality, and interpolation method. MATLAB objects enable you to access property data directly (see “Accessing Object Data” on page 12-14 for information on property syntax).

Properties are sometimes called fields in other programming languages and are similar to the fields of MATLAB structures. Properties have descriptive names, such as `Data` and `DataInfo`, in the case of `timeseries` objects, and can contain any kind of MATLAB data, including other objects.

An object, then, is a container for a predefined set of data. Unlike a cell array or structure, you cannot add new properties or delete defined properties from an object. Doing so would compromise the object’s intended purpose and violate the class design.

The class design can restrict the values you can assign to a property. For example, a `Length` property might restrict possible values to positive integers or might be read only and determine its own value when queried.

Methods Implement Operations

Class *methods* are functions designed to work with objects of a particular class. Methods enable the class designer to implement specific operations that are optimized for the data contained in the object. You do not have to extract the data from the object, modify its format, and pass it to a general-purpose MATLAB function because the class defines methods with an awareness of the object’s structure.

Methods can define operations that are unique to a particular class of object, such as adding a data sample to an existing set of time series data, or can *overload* common operations in a way that makes sense for the particular object. For example, `timeseries` objects have an `addsample` method to add a new data sample to an existing `timeseries` object. Also, `timeseries` overloads the MATLAB `plot` function to work with `timeseries` objects.

MATLAB software determines which overloaded version of a method to call based on the class of the object passed as an argument. If you execute a MATLAB statement like:

```
tsobjnew = tsobj1 + tsobj2;
```

where `tsobj1` and `tsobj2` are `timeseries` objects, MATLAB calls the `timeseries` version of the `+` operation (if defined) and returns a new `timeseries` object.

Because the `timeseries` class defines the operation, you can add a `timeseries` object to a scalar number:

```
tsobjnew = tsobj1 + 4;
```

The class definition determines what happens when you add a scalar double to a `timeseries` object (the scalar is added to each `Data` value).

Methods make working with objects convenient for the user, but also provide advantages to the class designer. Methods hide implementation details from users—you do not need to create your own functions to access and manipulate data, as you would when using general-purpose data structures like `structs` and cell arrays. This provides the flexibility to change the internal design of an object without affecting object clients (i.e., application programs that use the objects).

Events are Notices Broadcast to Listening Objects

Classes can define names for specific actions and trigger the broadcast of notices when those actions occur. Listeners respond to the broadcast of an event notice by executing a predefined function.

For example, objects can listen for the change of the value of a property and execute a function when that change occurs. If an object defines an event for which you can define a listening object, the object's documentation describes that event. See “Events — Sending and Responding to Messages” for information on how class designers use events.

Creating Objects

In this section...

“Class Constructor” on page 12-11

“When to Use Package Names” on page 12-11

Class Constructor

Usually, you create an object by calling a function designed for the purpose of creating that specific class of object. For example, the following code creates a `timeseries` object and assigns it to the variable `tsobj`:

```
load count.dat % Load some data
tsobj = timeseries(count(:,1),1:24, 'Name', 'Data1');
```

The `timeseries` constructor creates an object and initializes its data with the values specified as arguments. Classes that create objects define a special method whose purpose is to create objects of the class. This method has the same name as the class and is called the *class constructor*.

However, in some cases, you might create objects by calling other functions or even using a GUI. For example, a `try-catch` block can return an `MException` object that contains information about a specific error condition. In this case, you do not explicitly create the object, rather it is returned by the `catch` statement (see “Accessing Object Data” on page 12-14 for an example).

When to Use Package Names

A package is a container that provides a logical grouping for class and function definitions. The class and function names within a given package must be unique, but can be reused in other packages. Packages are folders that begin with the `+` character.

If a package folder contains a class definition, then you must use the package name when calling the class constructor. For example, this statement creates a `Map` object, whose class definition file is in a folder in the `containers` package:

```
mapobj = containers.Map({'rose', 'bicycle'}, {'flower', 'machine'});
```

You need to use the package name to refer to:

- Class constructors (e.g., `containers.Map`), which you call to create an object
- Static methods (methods that do not require an object of the class as an argument)
- Package functions (functions defined in the package)

However, because MATLAB uses the class of an object to determine which ordinary method to call, you do not need to use the package name in conjunction with object references. For example, suppose you have the following folder structure:

```
pathfolder/+packagename/@ClassName/ClassName.m
pathfolder/+packagename/@ClassName/staticMethodName.m
pathfolder/+packagename/functionName.m
```

In the following examples, `obj` is the object you are creating.

```
% Create object of ClassName
obj = packagename.ClassName(...);

% Call methodName
obj.methodName(...);

% Set or get the value of property PropertyName
obj.PropertyName = x;
x = obj.PropertyName;

% Call static method staticMethodName
packagename.ClassName.staticMethodName(...);

% Call package function functionName
packagename.functionName(...)
```

If a package folder contains a class definition file, then consider the package name as part of the class name. Wherever you need to use the class name, include the package name. For example, `containers.Map` is the full class name of the `Map` class.

See the object's user documentation for the syntax you need to use to create objects.

See “Organizing Classes in Folders” and “Create a Namespace with Packages” for more information on the use of packages.

See “Importing Classes” for information on importing packages into functions.

Accessing Object Data

In this section...

“Listing Public Properties” on page 12-14

“Getting Property Values” on page 12-14

“Setting Property Values” on page 12-15

Listing Public Properties

Note Always use the correct case when referring to properties by name.

Display the names of all public object properties using the `properties` function with the object’s class name or with an actual object. For example:

```
>> properties('MException')
```

```
Properties for class MException:
```

```
    identifier  
    message  
    cause  
    stack
```

Getting Property Values

After creating an object, you can access the values of its properties:

```
try  
    a = rand(4);  
    a(17) = 7;  
catch me % catch creates an MException object named me  
    disp(['Current error identifier: ' me.identifier])  
end  
Current error identifier: MATLAB:indexed_matrix_cannot_be_resized
```

Access the data in properties using dot notation:

```
object.PropertyName
```

For example, you can access the `message` property of the `MException` object, `me`, with this syntax:

```
me.message
ans =
    In an assignment A(I) = B, a matrix A cannot be resized.
```

See “Capturing Information About the Error” on page 18-5 for more information on using `MException` objects.

Setting Property Values

Objects often restrict what values you can assign to them. For example, the following `timeseries` object has 10 data values, each corresponding to a sample time:

```
tsobj = timeseries(rand(10,1),1:10,'Name','Random Sample');
```

Now suppose you attempt to set the `Data` property to a three-element vector:

```
tsobj.Data = [1 2 3];
Error using timeseries.timeseries>timeseries.utreshape
Size of the data array is incompatible with the time vector.
```

The `timeseries` class design ensures that the number of data samples matches the number of time samples. This illustrates one of the advantages a specialized object has over a general purpose-data structure like a MATLAB `struct`.

Calling Object Methods

In this section...

“What Operations Can You Perform” on page 12-16

“Method Syntax” on page 12-16

“Class of Objects Returned by Methods” on page 12-18

What Operations Can You Perform

Methods define an object’s behavior. Consequently, classes implement methods that an object user is unlikely to call directly. The user documentation for the object you are using describes the operations you can perform on any particular object.

You can list the methods defined by a class with the `methods` or `methodsview` functions:

```
methods('timeseries')
```

Methods for class timeseries:

<code>addevent</code>	<code>gettsbetweenevents</code>	<code>set</code>
<code>addsample</code>	<code>horzcat</code>	<code>setabstime</code>
<code>createTstoolNode</code>	<code>idealfilter</code>	<code>setinterpmethod</code>
<code>ctranspose</code>	<code>init</code>	<code>setprop</code>
<code>...</code>		

Method Syntax

Call an object’s method using dot notation:

```
returnValue = object.methodName(args,...)
```

You also can call a method using function syntax, passing the object as the first (left-most) argument.

```
returnValue = methodName(object,args,...)
```

For example, MException objects have a `getReport` method that returns information about the error.

```
try
    surf
catch me
    disp(me.getReport)
end
```

Error using ==> surf at 50
Not enough input arguments.

Dot and function notation are usually equivalent. That is, both of the following statements return the MException report:

```
rpt = getReport(me); % Call getReport using function notation
rpt = me.getReport; % Call getReport using dot notation
```

Calling the Correct Method

It is possible for the function syntax to call an unexpected method if there is more than one object in the argument list. Suppose there are two classes, `ClassA` and `ClassB`, that define a method called `addData`. Suppose further that `ClassA` is defined as being inferior to `ClassB` in precedence (something that the class designer can do in the class definition). In this situation, given `objA` is of `ClassA` and `objB` is of `ClassB`, the following two statements call different methods:

```
addData(objA, objB) % Calls objB.addData
objA.addData(objB) % Calls objA.addData
```

If `ClassA` and `ClassB` had equal precedence, then the left-most argument determines which method MATLAB calls (i.e., `objA.addData` in both statements).

It is unlikely that you will encounter this particular scenario, however, if you are calling a method that accepts more than one object as arguments, using dot notation removes any ambiguity about which object's method MATLAB calls.

Class of Objects Returned by Methods

While methods sometimes return objects of the same class, this is not always the case. For example, the MException object's getReport returns a character string:

```
try
  surf
catch me
  rpt = me.getReport;
end
```

```
whos
  Name          Size          Bytes  Class          Attributes

  me             1x1             1118  MException
  rpt            1x126           252   char
```

Methods can return any type of value and properties can contain any type of value. However, class constructor methods always return an object or array of objects of the same type as the class.

Desktop Tools Are Object Aware

In this section...

“Tab Completion Works with Objects” on page 12-19

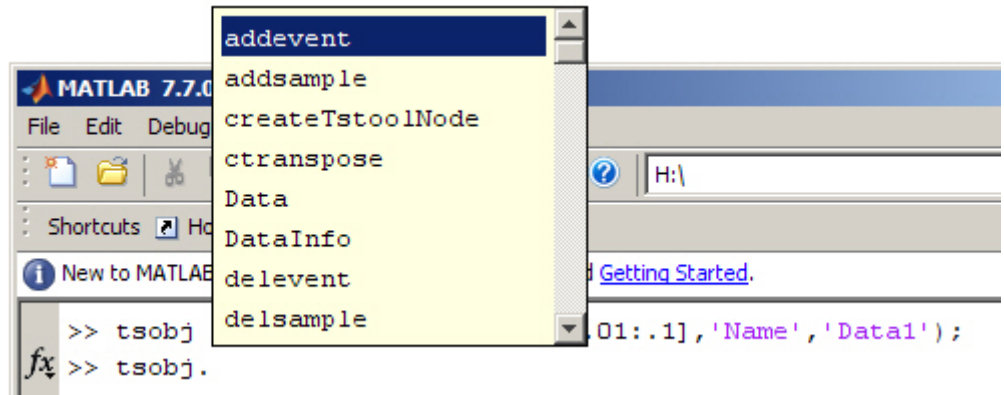
“Editing Objects with the Variable Editor” on page 12-19

Tab Completion Works with Objects

MATLAB tab completion works with objects. For example, if you enter an object name followed by a dot:

```
tsobj.
```

and then press the tab key, MATLAB pops up a selection box with a list of properties and methods:



The more letters you complete after the dot, the more specific is the list. See “Complete Names in the Command Window Using the Tab Key” for more information.

Editing Objects with the Variable Editor

You can use the MATLAB Variable Editor to edit object properties. To open an object in the Variable Editor, you can double-click the object name in the Workspace browser or use the `openvar` command:

```
tsobj = timeseries(rand(10,1),.01:.01:.1,'Name','Data1');  
openvar tsobj
```

See “Viewing and Editing Workspace Variables with the Variable Editor” for more information.

Getting Information About Objects

In this section...

“The Class of Workspace Variables” on page 12-21

“Information About Class Members” on page 12-23

“Logical Tests for Objects” on page 12-23

“Displaying Objects” on page 12-24

“Getting Help for MATLAB Objects” on page 12-25

The Class of Workspace Variables

All workspace variables are of a specific class. For example, consider the following variable created in your workspace:

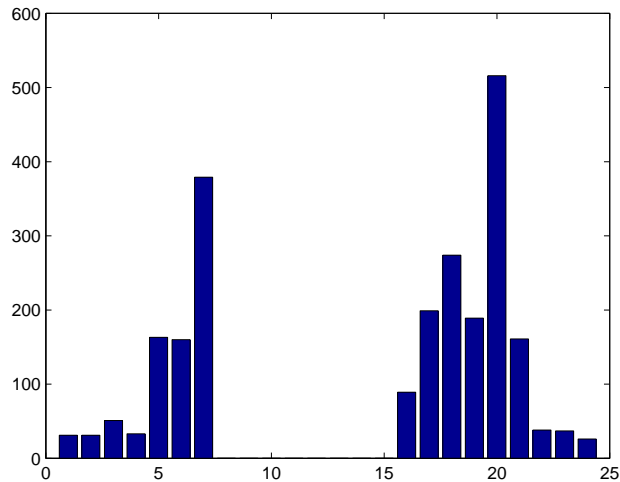
```
load count.dat % Load some data
tsobj = timeseries(count(:,1),1:24,'Name','Data1');
whos
```

Name	Size	Bytes	Class	Attributes
count	24x3	576	double	
tsobj	1x1	1261	timeseries	

The `whos` command lists information about your workspace variables. Notice that the variable loaded from the `count.dat` file (`count`) is an array of doubles. You know, therefore, that you can perform indexing and arithmetic operations on this array. For example:

```
newcount = sum(count,2);
newcount(8:15) = NaN;
bar(newcount)
```

Indexed assignment and the `bar` function work with inputs of class `double`.



However, the `timeseries` class does not define a `bar` method for `timeseries` objects. The `timeseries` class defines a `plot` method for graphing because the class design specified a line plot as the best way to represent time series data.

Extracting Data From Object Properties

Suppose you have a `timeseries` object and you want to work directly with the numeric values of the `timeseries` data. You can extract data from the object properties and assign these values to an array. For example

```
load count.dat
tsoj = timeseries(sum(count,2),1:24, 'Name', 'DataSum');
d = tsoj.Data;
t = tsoj.Time;
n = tsoj.Name;
d(8:15) = NaN;
bar(t,d); title(n)
```

Testing for the Class of an Object

Suppose you create a function that operates on more than one class of object. If you have a `timeseries` object, you call the `timeseries` plot method, but

if the object is of class `double`, you can call the `bar` function (which isn't supported by `timeseries` objects). You could use `isa` as in the following code to make this determination:

```
obj = tsobj.Data; % Define an input variable
function myPlotter(obj)
    if isa(obj, 'timeseries')
        plot(obj)
    elseif isa(obj, 'double')
        bar(obj)
    end
end
```

Information About Class Members

These functions provide information about the object.

Function	Purpose
<code>class</code>	Return class of object
<code>events</code>	List of event names defined by the class
<code>methods</code>	List of methods implemented by the class
<code>methodsvew</code>	Information on class methods in separate window
<code>properties</code>	List of class property names

Logical Tests for Objects

In functions, you might need conditional statements to determine the status of an object before performing certain actions. For example, you might perform different actions based on the class of an object (see “Testing for the Class of an Object” on page 12-22). The following functions provide logical tests for objects:

Function	Purpose
<code>isa</code>	Determine whether argument belongs to a particular class. True for object's class and all of object's superclasses.
<code>isequal</code>	Determine if two objects are equal.
<code>isobject</code>	Determine whether the input is a MATLAB object.

Testing for Object Equality

`isequal` finds two objects to be equal if all the following conditions are met:

- Both objects are of the same class
- Both objects are of the same size
- All corresponding property values are equal

`isequal` tests the value of every array element in every property and every property of every object contained in the objects being tested. As contained objects are tested for equality, MATLAB calls each object's own version of `isequal` (if such versions exist).

If objects contain large amounts of data stored in other objects, then testing for equality can be a time-consuming process.

Identifying MATLAB Objects

The `isobject` function returns `true` only for MATLAB objects. For Sun Java objects, use `isjava`. For Handle Graphics objects, use `ishandle`.

Note `ishandle` returns `false` for MATLAB handle objects. See “Testing for Handle or Value Class” on page 12-31 for more information.

Displaying Objects

When you issue commands that return objects and do not terminate those commands with a semicolon, or when you pass an object to the `disp` function, MATLAB displays information about the object. For example:

```
hobj = containers.Map({'Red Sox', 'Yankees'},
{'Boston', 'New York'})
hobj =
  containers.Map handle
  Package: containers

  Properties:
    Count: 2
    KeyType: 'char'
    ValueType: 'char'
  Methods, Events, Superclasses
```

This information includes links (shown in blue) to documentation on the object's class and superclasses, and lists of methods, events, and superclasses. Properties and their current values are also listed.

Some classes (`timeseries`, for example) redefine how they display objects to provide more useful information for this particular class.

Getting Help for MATLAB Objects

You can get documentation for MATLAB objects using the `doc` command with the class name. To see the reference pages for the objects used in this chapter, use the following commands:

```
doc timeseries
doc MException
doc containers.Map % Include the package name
```

Copying Objects

In this section...

“Two Copy Behaviors” on page 12-26

“Value Object Copy Behavior” on page 12-26

“Handle Object Copy Behavior” on page 12-27

“Testing for Handle or Value Class” on page 12-31

Two Copy Behaviors

There are two fundamental kinds of MATLAB classes—handles and values.

Value classes create objects that behave like ordinary MATLAB variables with respect to copy operations. Copies are independent values. Operations that you perform on one object do not affect copies of that object.

Handle classes create objects that behave as references. This is because a handle, and all copies of this handle, refer to the same underlying object. When you create a handle object, you can copy the handle, but not the data referenced by the object’s properties. Any operations you perform on a handle object are visible from all handles that reference that object.

Value Object Copy Behavior

MATLAB numeric variables are of value objects. For example, when you copy `a` to the variable `b`, both variables are independent of each other. Changing the value of `a` does not change the value of `b`:

```
a = 8;  
b = a;
```

Now reassign `a` and `b` is unchanged:

```
a = 6;  
b  
b =  
    8
```

Clearing a does not affect b:

```
clear a
b
b =
    8
```

Value Object Properties

The copy behavior of values stored as properties in value objects is the same. For example, suppose vobj1 is a value object with property a:

```
vobj1.a = 8; % Property is set to a value
```

If you copy vobj1 to vobj2, and then change the value of vobj1 property a, you can see that the value of the copied object's property vobj2.a is unaffected:

```
vobj2 =vobj1;
vobj1.a = 5;

vobj2.a
ans =
    8
```

Handle Object Copy Behavior

Here is a handle class called HdClass that defines a property called Data.

```
classdef HdClass < handle
    properties
        Data
    end
    methods
        function obj = HdClass(val)
            if nargin > 0
                obj.Data = val;
            end
        end
    end
end
```

Create an object of this class with the following statement:

```
hobj1 = HdClass(8)
```

Because this statement is not terminated with a semicolon, MATLAB displays information about the object:

```
hobj1 =  
  
    HdClass handle  
  
    Properties:  
        Data: 8  
  
    Methods, Events, Superclasses
```

The variable `hobj1` is a handle that references the object created. Copying `hobj1` to `hobj2` results in another handle (the variable `hobj2`) referring to the same object:

```
hobj2 = hobj1  
hobj2 =  
  
    HdClass handle  
  
    Properties:  
        Data: 8  
  
    Methods, Events, Superclasses
```

Because handle objects reference the data contained in their properties, copying an object copies the handle to a new variable name, but the properties still refer to the same data. For example, given that `hobj1` is a handle object with property `Data`:

```
hobj1.Data  
  
ans =  
  
     8
```

Change the value of `hobj1`'s `Data` property and the value of the copied object's `Data` property also changes:


```
hobj1.Data = 5;
```

```
hobj2.Data
```

```
ans =
```

```
5
```

Because `hobj2` and `hobj1` are handles to the same object, changing the copy, `hobj2`, also changes the data you access through handle `hobj1`:

```
hobj2.Data = 17;
```

```
hobj1.Data
```

```
ans =
```

```
17
```

Copy Method for Handle Classes

Handle classes can derive copy functionality from the `matlab.mixin.Copyable` class. Class designers can investigate the use of this class in their class design.

Reassigning Handle Variables

Reassigning a handle variable produces the same result as reassigning any MATLAB variable. When you create a new object and assign it to `hobj1`:

```
hobj1 = HdClass(3.14);
```

`hobj1` references the new object, not the same object referenced previously (and still referenced by `hobj2`).

Clearing Handle Variables

When you clear a handle from the workspace, MATLAB removes the variable, but does not removed the object referenced by the handle. Therefore, given `hobj1` and `hobj2`, which both reference the same object, you can clear either handle without affecting the object:

```
hobj1.Data = 2^8;
```

```
clear hobj1
hobj2
hobj2 =

    HdClass handle
```

```
Properties:
    Data: 256
```

[Methods](#), [Events](#), [Superclasses](#)

If you clear both `hobj1` and `hobj2`, then there are no references to the object and MATLAB deletes the object and frees the memory used by that object.

Deleting Handle Objects

To remove an object referenced by any number of handles, use `delete`. Given `hobj1` and `hobj2`, which both reference the same object, if you delete either handle, MATLAB deletes the object:

```
hobj1 = HdClass(8);
hobj2 = hobj1;
delete(hobj1)
hobj2
```

```
hobj2 =
```

```
deleted HdClass handle
```

[Methods](#), [Events](#), [Superclasses](#)

See “Destroying Objects” on page 12-33 for more information about object lifecycle.

Modifying Objects

When you pass an object to a function, MATLAB follows *pass by value* semantics. This means that MATLAB passes a copy of the object to the function. If you modify the object in the function, MATLAB modifies only the copy of the object. The differences in copy behavior between handle and value classes are important in such cases:

- Value class — The function must return the modified copy of the object to the caller.
- Handle class — The copy refers to the same data as the original object. Therefore, the function does not need to return the modified copy.

See “Passing Objects to Functions” for more information.

More Information About Handle and Value Classes

For information about handle and value classes for class designers, see “Value or Handle Class — Which to Use” in the Object-Oriented Programming documentation.

Testing for Handle or Value Class

If you are writing MATLAB programs that copy objects, you might need to determine if any given object is a handle or a value. To determine if an object is a handle object, use the `isa` function:

```
isa(obj, 'handle')
```

For example, the `containers.Map` class creates a handle object:

```
hobj = containers.Map({'Red Sox', 'Yankees'}, {'Boston', 'New York'});  
isa(hobj, 'handle')
```

```
ans =
```

```
1
```

`hobj` is also a `containers.Map` object:

```
isa(hobj, 'containers.Map')
```

```
ans =
```

```
1
```

If you query the class of `hobj`, you see that it is a `containers.Map` object:

```
class(hobj)
```

```
ans =
```

```
containers.Map
```

The `class` function returns the specific class of an object, whereas `isa` returns `true` for any of the object's superclasses as well. This behavior is consistent with the object-oriented concept that an object is a member of all its superclasses. Therefore, it is true that a `containers.Map` object is a `handle` object and a `containers.Map` object.

There is no equivalent test for value classes because there is no value base class. If an object is a value object, `isa(object, 'handle')` returns `false` (i.e., logical 0).

See Chapter 10, “Map Containers” for more information on the `containers.Map` class.

Destroying Objects

In this section...

“Object Lifecycle” on page 12-33

“Difference Between clear and delete” on page 12-33

Object Lifecycle

An object’s lifecycle ends when:

- You reassign a new value to that variable.
- The object is no longer used in a function.
- Function execution ends.

MATLAB handle classes have a special method called `delete` that MATLAB calls when a handle object lifecycle ends.

Calling `delete` on an object explicitly makes all copies of a handle object invalid because it destroys the data associated with the object and frees memory used by deleted objects. MATLAB calls `delete` automatically so it is not necessary for you to do so. Classes can redefine the handle class `delete` method to perform other cleanup operations, like closing files or saving data.

Deleting a handle object renders all copies invalid:

```
hobj1 = HdClass(8);  
hobj2 = hobj1;  
delete(hobj1)  
hobj2.Data  
Invalid or deleted object.
```

Difference Between clear and delete

The handle class `delete` method removes the handle object, but does not clear the variable name. The `clear` function removes a variable name, but does not remove the values to which the variable refers. For example, if you have two variables that refer to the same handle object, you can clear either one without affecting the actual object:

```
hobj = containers.Map({'Red Sox', 'Yankees'}, {'Boston', 'New York'});  
hobj_copy = hobj;  
clear hobj  
city = hobj_copy('Red Sox')  
  
city =  
  
Boston
```

If you call `clear` on all handle variables that refer to the same handle object, then you have lost access to the object and MATLAB destroys the object. That is, when there are no references to an object, the object ceases to exist.

On value objects, you can call `clear` to remove the variable. However, MATLAB does not automatically call a value class `delete` method, if one exists, when you clear the variable.

Defining Your Own Classes

All MATLAB data types are implemented as object-oriented classes. You can add data types of your own to your MATLAB environment by creating additional classes. These user-defined classes define the structure of your new data type, and the functions, or *methods*, that you write for each class define the behavior for that data type.

These methods can also define the way various MATLAB operators, including arithmetic operations, subscript referencing, and concatenation, apply to the new data types. For example, a class called `polynomial` might redefine the addition operator (+) so that it correctly performs the operation of addition on polynomials.

With MATLAB classes you can

- Create methods that overload existing MATLAB functionality
- Restrict the operations that are allowed on an object of a class
- Enforce common behavior among related classes by inheriting from the same parent class
- Significantly increase the reuse of your code

For more information, see “Classes in the MATLAB Language”.

Scripts and Functions

- Chapter 14, “Program Files”
- Chapter 15, “Types of Functions”
- Chapter 16, “Function Arguments”
- Chapter 17, “Programming Tips”

Program Files

- “Program Development” on page 14-2
- “Working with Functions in Files” on page 14-9
- “Scripts and Functions” on page 14-24
- “Base and Function Workspaces” on page 14-29
- “Share Data Between Workspaces” on page 14-30
- “Calling Functions” on page 14-35
- “Functions Provided By MATLAB” on page 14-44

Program Development

In this section...

“Overview” on page 14-2

“Creating a Program” on page 14-2

“Getting the Bugs Out” on page 14-3

“Cleaning Up the Program” on page 14-4

“Improving Performance” on page 14-5

“Checking It In” on page 14-6

“Protecting Your Source Code” on page 14-6

Overview

As you write a MATLAB function or script, you save it to a file that has a `.m` file extension. There are two types of these files you can write: scripts and functions. This section covers basic program development, describes how to write and call scripts and functions, and shows how to pass different types of data when calling a function.

For more ideas on good programming style, see “Program Development” on page 17-18 in the MATLAB Programming Tips documentation. The Programming Tips section is a compilation of useful pieces of information that can show you alternate and often more efficient ways to accomplish common programming tasks while also expanding your knowledge of MATLAB.

Creating a Program

You can type in your program code using any text editor. This section focuses on using the MATLAB Editor/Debugger for this purpose.

The first step in creating a program is to open an editing window. To create a file for a new function, type the word `edit` at the MATLAB command prompt. To edit an existing file, type `edit` followed by the file name:

```
edit drawPlot.m
```

MATLAB opens a new window for entering your program code. As you type in your program, MATLAB keeps track of the line numbers in the left column.

For more information on the Editor/Debugger, see “Create, Open, Save, and Close Files”.

Saving the Program

It is usually a good idea to save your program periodically while you are in the development process. To do this, click **File > Save** in the Editor/Debugger. Enter a file name with a `.m` extension in the **Save file as** dialog box that appears and click **OK**. It is customary and less confusing if you give the file the same name as the first function in the file.

Running the Program

Before trying to run your program, make sure that its file is on the MATLAB path. The MATLAB path defines those folders that you want MATLAB to know about when executing files. The path includes all the folders that contain functions provided with MATLAB. It should also include any folders that you use for your own functions.

Use the `which` function to see if your program is on the path:

```
which drawPlot
  D:\user5\matlab\mywork\drawPlot.m
```

If not, add its folder to the path using the `addpath` function:

```
addpath('D:\user5\matlab\mywork')
```

Now you can run the program just by typing the name of the file at the MATLAB command prompt:

```
drawPlot(xdata, ydata)
```

Getting the Bugs Out

In all but the simplest programs, you are likely to encounter some type of unexpected behavior when you run the program for the first time. Program defects can show up in the form of warning or error messages displayed in the

command window, programs that hang (never terminate), inaccurate results, or some number of other symptoms. This is where the second functionality of the MATLAB Editor/Debugger becomes useful.

The MATLAB Debugger enables you to examine the inner workings of your program while you run it. You can stop the execution of your program at any point and then continue from that point, stepping through the code line by line and examining the results of each operation performed. You have the choice of operating the debugger from the Editor window that displays your program, from the MATLAB command line, or both.

The Debugging Process

You can step through the program right from the start if you want. For longer programs, you will probably save time by stopping the program somewhere in the middle and stepping through from there. You can do this by approximating where the program code breaks and setting a stopping point (or *breakpoint*) at that line. Once a breakpoint has been set, start your program from the MATLAB command prompt. MATLAB opens an Editor/Debugger window (if it is not already open) showing a green arrow pointing to the next line to execute.

From this point, you can examine any values passed into the program, or the results of each operation performed. You can step through the program line by line to see which path is taken and why. You can step into any functions that your program calls, or choose to step over them and just see the end results. You can also modify the values assigned to a variable and see how that affects the outcome.

To learn about using the MATLAB Debugger, see “Debugging Process and Features”. Type `help debug` for a listing of all MATLAB debug functions.

For programming tips on how to debug, see “Debugging” on page 17-21 in the Programming Tips documentation.

Cleaning Up the Program

Even after your program is bug-free, there are still some steps you can take to improve its performance and readability. The MATLAB Code Analyzer utility generates a report that can highlight potential problems in your code. For

example, you might be using the element-wise AND operator (&) where the short-circuit AND (&&) is more appropriate. You might be using the `find` function in a context where logical subscripting would be faster.

MATLAB offers the Code Analyzer and several other reporting utilities to help you make the finishing touches to your program code. These tools are described in “Using MATLAB Reports”.

Improving Performance

The MATLAB Profiler generates a report that shows how your program spends its processing time. For details about using the MATLAB Profiler, see [Profiling for Improving Performance](#). For tips on other ways to improve the performance of your programs, see “Techniques for Improving Performance” on page 20-4.

Three types of reports are available:

- “Summary Report” on page 14-5
- “Detail Report” on page 14-5
- “File Listing” on page 14-6

Summary Report

The summary report provides performance information on your main program and on every function it calls. This includes how many times each function is called, the total time spent in that function, along with a bar graph showing the relative time spent by each function.

Detail Report

When you click a function name in the summary report, MATLAB displays a detailed report on that function. This report shows the lines of that function that take up the most time, the time spent executing that line, the percentage of total time for that function that is spent on that line, and a bar graph showing the relative time spent on the line.

File Listing

The detail report for a function also displays all code for that function. This listing enables you to view the time-consuming code in the context of the entire function body. For every line of code that takes any significant time, additional performance information is provided by the statistics and by the color and degree of highlighting of the program code.

Checking It In

Source control systems offer a way to manage large numbers of files while they are under development. They keep track of the work done on these files as your project progresses, and also ensure that changes are made in a secure and orderly fashion.

If you have a source control system available, you will probably want to check your files into the system once they are complete. If further work is required on one of those files, you just check it back out, make the necessary modifications, and then check it back in again.

MATLAB provides an interface to external source control systems so that you can check files in and out directly from your MATLAB session. For more information, see:

- “Checking Files Into and Out of Source Control from the MATLAB Desktop on Microsoft Windows”
- “Checking Files Into the Source Control System on UNIX Platforms”

Protecting Your Source Code

Although MATLAB source (.m) code is executable by itself, the contents of MATLAB source files are easily accessed, revealing design and implementation details. If you do not want to distribute your proprietary application code in this format, you can use one of these more secure options instead:

- Deploy as P-code — Convert some or all of your source code files to a content-obscured form called a *P-code* file (from its .p file extension), and distribute your application code in this format.

- Compile into binary format — Compile your source code files using the MATLAB Compiler to produce a standalone application. Distribute the latter to end users of your application.

In general, if you want to run the code as a standalone application outside of MATLAB, it is best to use the MATLAB Compiler™ to make your code secure. If you plan to run the code within the MATLAB environment, there is no need to run the Compiler. Instead, convert to P-code those modules of your source code that need to be secure.

Building a Content Obscured Format with P-Code

A P-code file behaves the same as the MATLAB source from which it was produced. The P-code file also runs at the same speed as the source file. Because the contents of P-code files are purposely obscured, they offer a secure means of distribution outside of your organization.

Note Because users of P-code files cannot view the MATLAB code, consider providing diagnostics to enable a user to proceed in the event of an error.

Building the P-Code File. To generate a P-code file, enter the following command in the MATLAB Command Window:

```
pcode file1 file2, ...
```

The command produces the files, *file1.p*, *file2.p*, and so on. To convert *all* *.m* source files residing in your current folder to P-code files, use the command:

```
pcode *.m
```

See the `pcode` function reference page for a description of all syntaxes for generating P-code files.

Invoking the P-Code File. You invoke the resulting P-code file in the same way you invoke the MATLAB *.m* source file from which it was derived. For example, to invoke file *myfun.p*, type

```
[out, out2, ...] = myfun(in1, in2, ...);
```

To invoke script `myscript.p`, type

```
myscript;
```

When you call a P-code file, MATLAB gives it execution precedence over its corresponding `.m` source file. This is true even if you happen to change the source code at some point after generating the P-code file. Remember to remove the `.m` source file before distributing your code.

Running Older P-Code Files on Later Versions of MATLAB. P-Code files are designed to be independent of the release under which they were created and the release in which they are used (backward and forward compatibility). New and deprecated MATLAB features can be a problem, but it is the same problem that would exist if you used the original MATLAB input file. To fix errors of this kind in a P-code file, fix the corresponding MATLAB input file and create a new P-code file.

P-code files built using MATLAB Version 7.4 and earlier have a different format than those built with more recent versions of MATLAB. You still can use these older P-code files when you run MATLAB 7.4 and later, but this capability could be removed in a future release. MathWorks recommends that you rebuild any P-code files that were built with MATLAB 7.4 or earlier using a more recent version of MATLAB, and then redistribute them as necessary.

Building a Standalone Executable

Another way to protect your source code is to build it into a standalone executable and distribute the executable, along with any other necessary files, to external customers. You must have the MATLAB Compiler and a supported C or C++ compiler installed to prepare files for deployment. The end user, however, does not need MATLAB.

To build a standalone application for your MATLAB application, develop and debug your application following the usual procedure for MATLAB program files. Then, generate the executable file or files following the instructions in “Steps by the Developer to Deploy to End Users” in the MATLAB Compiler documentation.

Working with Functions in Files

In this section...

“Overview” on page 14-9

“Types of Program Files” on page 14-9

“Basic Parts of a Program File” on page 14-10

“Creating a Program File” on page 14-15

“Providing Help for Your Program” on page 14-17

“Cleaning Up When the Function Completes” on page 14-17

Overview

The MATLAB software 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 of `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 program file.

Types of Program Files

Program files can be *scripts* that simply execute a series of MATLAB statements, or they can be *functions* that also accept input arguments and produce output.

MATLAB scripts:

- Are useful for automating a series of steps you need to perform many times.
- Do not accept input arguments or return output arguments.
- Store variables in a workspace that is shared with other scripts and with the MATLAB command line interface.

MATLAB functions:

- Are useful for extending the MATLAB language for your application.

- Can accept input arguments and return output arguments.
- Store variables in a workspace internal to the function.

Basic Parts of a Program File

This simple function shows the basic parts of a program file. Any line that begins with % is not executable:

```
function f = fact(n)                Function definition line
% Compute a factorial value.       H1 line
% FACT(N) returns the factorial of N, Help text
% usually denoted by N!

% Put simply, FACT(N) is PROD(1:N). Comment
f = prod(1:n);                     Function body
```

This table briefly describes each of these program file parts. Both functions and scripts can have all of these parts, except for the function definition line which applies to functions only. The sections that follow the table describe these parts in greater detail.

File Element	Description
Function definition line (functions only)	Defines the function name, and the number and order of input and output arguments
H1 line	A one line summary description of the program, displayed when you request help on an entire folder, or when you use lookfor
Help text	A more detailed description of the program, displayed together with the H1 line when you request help on a specific function
Function or script body	Program code that performs the actual computations and assigns values to any output arguments
Comments	Text in the body of the program that explains the internal workings of the program

Function Definition Line

The function definition line informs MATLAB that the file contains a function, and specifies the argument calling sequence of the function. This line contains the `function` keyword and must always be the first line of the file, except for lines that are nonexecutable comments. The function definition line for the `fact` function is

```
function y = fact(x)
```

Diagram illustrating the components of the function definition line `function y = fact(x)`:

- `function`: keyword
- `y`: output argument
- `fact`: function name
- `x`: input argument

All MATLAB functions have a function definition line that follows this pattern.

Function Name. Function names must begin with a letter, can contain any alphanumeric characters or underscores, and must be no longer than the maximum allowed length (returned by the function `namelengthmax`). Because variables must obey similar rules, you can use the `isvarname` function to check whether a function name is valid:

```
isvarname myfun
```

Function names also cannot be the same as any MATLAB keyword. Use the `iskeyword` function with no inputs to display a list of all keywords.

Although function names can be of any length, MATLAB uses only the first `N` characters of the name (where `N` is the number returned by the function `namelengthmax`) and ignores the rest. Hence, it is important to make each function name unique in the first `N` characters:

```
N = namelengthmax
```

```
N =
```

```
63
```

Note Some operating systems might restrict file 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 file name and the function definition line name are different, MATLAB ignores the internal (function) name. Thus, if `average.m` is the file that defines a function named `computeAverage`, you would invoke the function by typing

```
average
```

Note While the function name specified on the function definition line does not have to be the same as the file name, it is best to use the same name for both to avoid confusion.

Function Arguments. If the function has multiple output values, enclose the output argument list in square brackets. Input arguments, if present, are enclosed in parentheses following the function name. Use commas to separate multiple input or output arguments. Here is the declaration for a function named `sphere` that has three inputs and three outputs:

```
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 the variables 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 functionname` 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 file, it is important to make it as descriptive as possible.

Help Text

You can create online help for your program files by entering help text on one or more consecutive comment lines at the start of your program. MATLAB considers the first group of consecutive lines immediately following the H1 line that begin with `%` to be the online help text for the function. The first line without `%` as the left-most character ends the help.

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 functionname` at the command prompt, MATLAB displays the H1 line followed by the online help text for that function. The help system ignores any comment lines that appear after this help block.

Note Help text in a program file can be viewed at the MATLAB command prompt only (using `help functionname`). You cannot display this text using the MATLAB Help browser. You can, however, use the Help browser to get help on MATLAB functions and also to read the documentation on any MathWorks products.

The Function or Script 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 a program 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.
```

In addition to comment lines, you can insert blank lines anywhere in the file. Blank lines are ignored. However, a blank line can indicate the end of the help text entry for a program file.

Block Comments. To write comments that require more than one line, use the block comment operators, %{ and %}:

```
{
This next block of code checks the number of inputs
passed in, makes sure that each input is a valid data
type, and then branches to start processing the data.
}
```

Note The `%{` and `%}` operators must appear alone on the lines that immediately precede and follow the block of help text. Do not include any other text on these lines.

Creating a Program File


You create files for your programs using a text editor. MATLAB provides a built-in editor, but you can use any text editor you like. Once you have written and saved the file, you can run the program 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
```

Using Text Editors

Program files are ordinary text files that you create using a text editor. If you use the MATLAB Editor/Debugger, open a new file by selecting **New > File** from the **File** menu at the top of the MATLAB Command Window.

Another way to edit a program file is from the MATLAB command line using the `edit` function. For example,

```
edit foo
```

opens the editor on the file `foo.m`. Omitting a file name opens the editor on an untitled file.

You can create the `fact` function shown in “Basic Parts of a Program File” on page 14-10 by opening your text editor, entering the lines shown, and saving the text in a file called `fact.m` in your current folder.

Once you have created this file, here are some things you can:

- List the names of the files in your current folder:

```
what
```

- List the contents of file `fact.m`:

```
type fact
```

- Call the `fact` function:

```
fact(5)
ans =
    120
```

A Word of Caution on Saving Program Files

Save any files you create and any MathWorks supplied files that you edit in folders outside of the folder tree in which the MATLAB software is installed. If you keep your files in any of the installed folders, your files can be overwritten when you install a new version of MATLAB.

MATLAB installs its software into folders under `matlabroot/toolbox`. To see what `matlabroot` is on your system, type `matlabroot` at the MATLAB command prompt.

Also note that locations of files in the `matlabroot/toolbox` folder tree are loaded and cached in memory at the beginning of each MATLAB session to

improve performance. If you save files to *matlabroot*/toolbox folders using an external editor, or if you add or remove files from these folders using file system operations, enter the commands `clear functionname` and `rehash toolbox` before you use the files in the current session.

For more information, see the `rehash` function reference page or the section `Toolbox Path Caching`.

Providing Help for Your Program

You can provide user information for the programs you write by including a help text section at the beginning of your program file. (See “Help Text” on page 14-13).

You can also make help entries for an entire folder by creating a file with the special name `Contents.m` that resides in the folder. This file must contain only comment lines; that is, every line must begin with a percent sign. MATLAB displays the lines in a `Contents.m` file whenever you type

```
help foldername
```

`Contents.m` files are optional. If you have a folder that is on the path that does not contain a `Contents.m` file, MATLAB displays the first comment line (the “H1” line) of each `.m` file in response to typing `help foldername`. If you do not want to display any help summaries at all, create an empty `Contents.m` file in that folder. When an empty `Contents.m` file exists, typing `help foldername` causes MATLAB to respond with `No help found for foldername..`

To get help in creating and validating your `Contents.m` files, you can use the `Contents Report` tool in the `Current Folder` browser. See “`Displaying and Updating a Report on the Contents of a Folder`” for more information.

Cleaning Up When the Function Completes

When you have programmed all that you set out to do in your file, there is one last step to consider before it is complete. Make sure that you leave your program environment in a clean state that does not interfere with any other program code. For example, you might want to

- Close any files that you opened for import or export.
- Restore the MATLAB path.
- Lock or unlock memory to prevent or allow erasing MATLAB function or MEX-files.
- Set your working folder back to its default if you have changed it.
- Make sure global and persistent variables are in the correct state.

MATLAB provides the `onCleanup` function for this purpose. This function, when used within any program, establishes a cleanup routine for that function. When the function terminates, whether normally or in the event of an error or **Ctrl+C**, MATLAB automatically executes the cleanup routine.

The following statement establishes a cleanup routine `cleanupFun` for the currently running program:

```
cleanupObj = onCleanup(@cleanupFun);
```

When your program exits, MATLAB finds any instances of the `onCleanup` class and executes the associated function handles. The process of generating and activating function cleanup involves the following steps:

- 1** Write one or more cleanup routines for the program under development. Assume for now that it takes only one such routine.
- 2** Create a function handle for the cleanup routine.
- 3** At some point, generally early in your program code, insert a call to the `onCleanup` function, passing the function handle.
- 4** When the program is run, the call to `onCleanup` constructs a cleanup object that contains a handle to the cleanup routine created in step 1.
- 5** When the program ends, MATLAB implicitly clears all objects that are local variables. This invokes the destructor method for each local object in your program, including the cleanup object constructed in step 4.
- 6** The destructor method for this object invokes this routine if it exists. This perform the tasks needed to restore your programming environment.

You can declare any number of cleanup routines for a program file. Each call to `onCleanup` establishes a separate cleanup routine for each cleanup object returned.

If, for some reason, the object returned by `onCleanup` persists beyond the life of your program, then the cleanup routine associated with that object is not run when your function terminates. Instead, it will run whenever the object is destroyed (e.g., by clearing the object variable).

Your cleanup routine should never rely on variables that are defined outside of that routine. For example, the nested function shown here on the left executes with no error, whereas the very similar one on the right fails with the error, `Undefined function or variable 'k'`. This results from the cleanup routine's reliance on variable `k` which is defined outside of the nested cleanup routine:

```
function testCleanup
k = 3;
myFun
    function myFun
        fprintf('k is %d\n', k)
    end
end

function testCleanup
k = 3;
obj = onCleanup(@myFun);
    function myFun
        fprintf('k is %d\n', k)
    end
end
```

Examples of Cleaning Up a Program Upon Exit

Example 1 – Close Open Files on Exit. MATLAB closes the file with identifier `fid` when function `openFileSafely` terminates:

```
function openFileSafely(fileName)
fid = fopen(fileName, 'r');
c = onCleanup(@( )fclose(fid));

s = fread(fid);
    .
    .
    .
end
```

Example 2 – Maintain the Selected Folder. This example preserves the current folder whether `functionThatMayError` returns an error or not:

```
function changeFolderSafely(fileName)
    currentFolder = pwd;
    c = onCleanup(@()cd(currentFolder));

    functionThatMayError;
end % c executes cd(currentFolder) here.
```

Example 3 – Close Figure and Restore MATLAB Path. This example extends the MATLAB path to include files in the `toolbox\images` folders, and then displays a figure from one of these folders. After the figure displays, the cleanup routine `restore_env` closes the figure and restores the path to its original state:

```
function showImageOutsidePath(imageFile)
    fig1 = figure;
    imgpath = genpath([matlabroot '\toolbox\images']);

    % Define the cleanup routine.
    cleanupObj = onCleanup(@()restore_env(fig1, imgpath));

    % Modify the path to gain access to the image file,
    % and display the image.
    addpath(imgpath);
    rgb = imread(imageFile);
    fprintf('\n  Opening the figure %s\n', imageFile);
    image(rgb);
    pause(2);

    % This is the cleanup routine.
    function restore_env(fighandle, newpath)
        disp '  Closing the figure'
        close(fighandle);
        pause(2)

        disp '  Restoring the path'
        rmpath(newpath);
    end
end
```

Run the function as shown here. You can verify that the path has been restored by comparing the length of the path before and after running the function:

```
origLen = length(path);

showImageOutsidePath('greens.jpg')
    Opening the figure greens.jpg
    Closing the figure
    Restoring the path

currLen = length(path);
currLen == origLen
ans =
     1
```

Retrieving Information About the Cleanup Routine

In Example 3 shown above, the cleanup routine and data needed to call it are contained in a handle to an anonymous function:

```
@()restore_env(fig1, imgpath)
```

The details of that handle are then contained within the object returned by the `onCleanup` function:

```
cleanupObj = onCleanup(@()restore_env(fig1, imgpath));
```

You can access these details using the `task` property of the cleanup object as shown here. (Modify the `showImageOutsidePath` function by adding the following code just before the comment line that says, “% This is the cleanup routine.”)

```
disp '    Displaying information from the function handle:'
task = cleanupObj.task;
fun = functions(task)
wsp = fun.workspace{2,1}
fprintf('\n');
pause(2);
```

Run the modified function to see the output of the `functions` command and the contents of one of the workspace cells:

```
showImageOutsidePath('greens.jpg')
```

Opening the figure `greens.jpg`

Displaying information from the function handle:

```
fun =  
    function: '@()restore_env(fig1,imgpath)'  
    type: 'anonymous'  
    file: 'c:\work\g6.m'  
    workspace: {2x1 cell}  
wsp =  
    imageFile: 'greens.jpg'  
    fig1: 1  
    imgpath: [1x3957 char]  
    cleanupObj: [1x1 onCleanup]  
    rgb: [300x500x3 uint8]  
    task: @()restore_env(fig1,imgpath)
```

Closing the figure

Restoring the path

Using onCleanup Versus try-catch

Another way to run a cleanup routine when a function terminates unexpectedly is to use a `try-catch` statement. There are limitations to using this technique however. If the user ends the program by typing **Ctrl+C**, MATLAB immediately exits the `try` block, and the cleanup routine never executes. The cleanup routine also does not run when you exit the function normally.

The following program cleans up if an error occurs, but not in response to **Ctrl+C**:

```
function cleanupByCatch  
try  
    pause(10);  
catch  
    disp(' Collecting information about the error')
```



```
        disp('    Executing cleanup tasks')
    end
```

Unlike the try-catch statement, the `onCleanup` function responds not only to a normal exit from your program and any error that might be thrown, but also to **Ctrl+C**. This next example replaces the try-catch with `onCleanup`:

```
function cleanupByFunc
obj = onCleanup(@(())...
    disp('    Executing cleanup tasks'));
pause(10);
```

onCleanup in Scripts

`onCleanup` does not work in scripts as it does in functions. In functions, the cleanup object is stored in the function workspace. When the function exits, this workspace is cleared thus executing the associated cleanup routine. In scripts, the cleanup object is stored in the base workspace (that is, the workspace used in interactive work done at the command prompt). Because exiting a script has no effect on the base workspace, the cleanup object is not cleared and the routine associated with that object does not execute. To use this type of cleanup mechanism in a script, you would have to explicitly clear the object from the command line or another script when the first script terminates.

Scripts and Functions

In this section...

“Scripts” on page 14-24

“Functions” on page 14-25

“Types of Functions” on page 14-26

“Organizing Your Functions” on page 14-27

“Identifying Dependencies” on page 14-27

Scripts

Scripts are the simplest kind of program file because they have no input or output arguments. They are useful for automating series of MATLAB commands, such as computations that you have to perform repeatedly from the command line.

For example, these statements calculate rho for several trigonometric functions of theta, then create a series of polar plots:

```
% A script to produce          % Comment lines
% "flower petal" plots
theta = -pi:0.01:pi;          % Computations
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,:))      % Graphics output
    pause
end
```

Try entering these commands in a 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 **Enter** or **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 (`k`, `theta`, and `rho`) remain in the workspace. To see a listing of them, enter `whos` at the command prompt.

Scripts share the base workspace with your interactive MATLAB session and with other scripts. For more information, see “Base and Function Workspaces” on page 14-29.

Functions

The main difference between a script and a function is that a function accepts input from and returns output to its caller, whereas scripts do not. You define MATLAB functions in a file that begins with a line containing the `function` key word. You cannot define a function within a script file or at the MATLAB command line.

Functions always begin with a function definition line and end either with the first matching end statement, the occurrence of another function definition line, or the end of the file, whichever comes first. Using `end` to mark the end of a function definition is required only when the function being defined contains one or more nested functions.

The average function is a simple 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. Nonvector 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
```

Try entering these commands in a 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
```

Each function in a file has an area of memory, separate from the MATLAB base workspace, in which it operates. This area, called the function workspace, gives each function its own workspace context. For more information, see “Base and Function Workspaces” on page 14-29.

Types of Functions

MATLAB provides the following types of functions. Each function type is described in more detail in a later section of this documentation:

- The primary function is the first function in a program file and typically contains the main program.
- Subfunctions act as subroutines to the main function. You can also use them to define multiple functions within a single file.
- Nested functions are functions defined within another function. They can help to improve the readability of your program and also give you more flexible access to variables in the file.
- Anonymous functions provide a quick way of making a function from any MATLAB expression. You can compose anonymous functions either from within another function or at the MATLAB command prompt.
- Overloaded functions are useful when you need to create a function that responds to different types of inputs accordingly. They are similar to overloaded functions in any object-oriented language.
- Private functions give you a way to restrict access to a function. You can call them only from a function in the parent folder.

You might also see the term function functions in the documentation. This is not really a separate function type. The term function functions refers to any functions that accept another function as an input argument. You can pass a function to another function using a function handle.

Organizing Your Functions

When writing and saving your functions, you have several options on how to organize the functions within the file, and also where in your folder structure you want to save them. Be sure to place your function files either in the folder in which you plan to run MATLAB, or in some other folder that is on the MATLAB path.

Use this table as a general guide when creating and saving your files:

If your program or routine . . .	then . . .
Requires only one function	Make it a single (primary) function in the file.
Also requires subroutines	Make each subroutine a subfunction within same file as the primary.
Is for use only in the context of a certain function	Nest it within the other function. Nested functions also offer wider access to variables within the function.
Is a constructor or method of a MATLAB class	Put the file in a MATLAB class folder.
Is to have limited access	Put the file in a private subfolder.
Is part of a group of similar functions or classes	Put the file in a package subfolder.

If necessary, you can work around some of the constraints regarding function access by using function handles. You might find this useful when debugging your functions.

Identifying Dependencies

If you need to know what other functions and scripts your program is dependent upon, use one of the techniques described below.

Simple Display of Program File Dependencies

For a simple display of all program files referenced by a particular function, follow these steps:

- 1 Type `clear functions` to clear all functions from memory (see Note below).

Note `clear functions` does not clear functions locked by `mlock`. If you have locked functions (which you can check using `inmem`) unlock them with `munlock`, and then repeat step 1.

- 2 Execute the function you want to check. Note that the function arguments you choose to use in this step are important, because you can get different results when calling the same function with different arguments.
- 3 Type `inmem` to display all program files that were used when the function ran. If you want to see what MEX-files were used as well, specify an additional output:

```
[mfiles, mexfiles] = inmem
```

Detailed Display of Program File Dependencies

For a much more detailed display of dependent function information, use the `depfun` function. In addition to program files, `depfun` shows which built-ins and classes a particular function depends on:

```
[list, builtins, classes] = depfun('strtok.m');  
  
list  
list =  
    'D:\matlabR14\toolbox\matlab\strfun\strtok.m'  
    'D:\matlabR14\toolbox\distcomp\toChar.m'  
    'D:\matlabR14\toolbox\matlab\datafun\prod.m'  
    'D:\matlabR14\toolbox\matlab\datatypes\@opaque\char.m'  
    .  
    .  
    .
```

Base and Function Workspaces

This topic explains the differences between the base workspace and function workspaces, including workspaces for local functions (sometimes called subfunctions), nested functions, and scripts.

The *base workspace* stores variables that you create at the command line. This includes any variables that scripts create, assuming that you run the script from the command line or from the Editor. Variables in the base workspace exist until you clear them or end your MATLAB session.

Functions do not use the base workspace. Every function has its own *function workspace*. Each function workspace is separate from the base workspace and all other workspaces to protect the integrity of the data. Even local functions in a common file have their own workspaces. Variables specific to a function workspace are called *local* variables. Typically, local variables do not remain in memory from one function call to the next.

When you call a script from a function, the script uses the function workspace.

Like local functions, nested functions have their own workspaces. However, these workspaces are unique in two significant ways:

- Nested functions can access and modify variables in the workspaces of the functions that contain them.
 - All of the variables in nested functions or the functions that contain them must be explicitly defined. That is, you cannot call a function or script that assigns values to variables unless those variables already exist in the function workspace.
- “Share Data Between Workspaces” on page 14-30
- “Nested Functions” on page 15-16

Related Examples

More About

Share Data Between Workspaces

This topic shows how to share variables between workspaces or allow them to persist between function executions.

In most cases, variables created within a function are *local* variables known only within that function. Local variables are not available at the command line or to any other function. However, there are several ways to share data between functions or workspaces.

In this section...

“Best Practice: Passing Arguments” on page 14-30

“Nested Functions” on page 14-31

“Persistent Variables” on page 14-32

“Global Variables” on page 14-32

“Evaluating in Another Workspace” on page 14-33

Best Practice: Passing Arguments

The most secure way to extend the scope of a function variable is to use function input and output arguments, which allow you to pass values of variables.

For example, create two functions, `update1` and `update2`, that share and modify an input value. `update2` can be a local function in the file `update1.m`, or can be a function in its own file, `update2.m`.

```
function y1 = update1(x1)
    y1 = 1 + update2(x1);
```

```
function y2 = update2(x2)
    y2 = 2 * x2;
```

Call the `update1` function from the command line and assign to variable `Y` in the base workspace:

```
X = [1,2,3];
Y = update1(X)
```



```
Y =
    3    5    7
```

Nested Functions

A nested function has access to the workspaces of all functions in which it is nested. So, for example, a nested function can use a variable (in this case, `x`) that is defined in its parent function:

```
function primaryFx
    x = 1;
    nestedFx;

    function nestedFx
        x = x + 1;
    end
end
```

When parent functions do not use a given variable, the variable remains local to the nested function. For example, in this version of `primaryFx`, the two nested functions have their own versions of `x` that cannot interact with each other.

```
function primaryFx
    nestedFx1;
    nestedFx2;

    function nestedFx1
        x = 1;
    end

    function nestedFx2
        x = 2;
    end
end
```

For more information, see “Variable Scope in Nested Functions” on page 15-19.

Persistent Variables

When you declare a variable within a function as persistent, the variable retains its value from one function call to the next. Other local variables retain their value only during the current execution of a function. Persistent variables are equivalent to static variables in other programming languages.

Declare variables using the `persistent` keyword before you use them. MATLAB initializes persistent variables to an empty matrix, `[]`.

For example, define a function in a file named `findSum.m` that initializes a sum to 0, and then adds to the value on each iteration.

```
function findSum(inputvalue)
persistent SUM_X

if isempty(SUM_X)
    SUM_X = 0;
end
SUM_X = SUM_X + inputvalue;
```

When you call the function, the value of `SUM_X` persists between subsequent executions.

These operations clear the persistent variables for a function:

- `clear all`
- `clear functionname`
- Editing the function file

To prevent clearing persistent variables, lock the function file using `mlock`.

Global Variables

Global variables are variables that you can access from functions or from the command line. They have their own workspace, which is separate from the base and function workspaces.

However, global variables carry notable risks. For example:

- Any function can access and update a global variable. Other functions that use the variable might return unexpected results.
- If you unintentionally give a “new” global variable the same name as an existing global variable, one function can overwrite the values expected by another. This error is difficult to diagnose.

Use global variables sparingly, if at all.

If you use global variables, declare them using the `global` keyword before you access them within any particular location (function or command line). For example, create a function in a file called `falling.m`:

```
function h = falling(t)
    global GRAVITY
    h = 1/2*GRAVITY*t.^2;
```

Then, enter these commands at the prompt:

```
global GRAVITY
GRAVITY = 32;
y = falling((0:.1:5)');
```

The two global statements make the value assigned to `GRAVITY` at the command prompt available inside the function. However, as a more robust alternative, redefine the function to accept the value as an input:

```
function h = falling(t,gravity)
    h = 1/2*gravity*t.^2;
```

Then, enter these commands at the prompt:

```
GRAVITY = 32;
y = falling((0:.1:5)',GRAVITY);
```

Evaluating in Another Workspace

The `evalin` and `assignin` functions allow you to evaluate commands or variable names from strings and specify whether to use the current or base workspace.

Like global variables, these functions carry risks of overwriting existing data. Use them sparingly.

`evalin` and `assignin` are sometimes useful for callback functions in graphical user interfaces to evaluate against the base workspace. For example, create a list box of variable names from the base workspace:

```
function listBox
figure;
lb = uicontrol('Style','listbox','Position',[10 10 100 100],...
              'Callback',@update_listBox);
update_listBox(lb)

function update_listBox(src,~)
vars = evalin('base','who');
set(src,'String',vars)
```

For other programming applications, consider argument passing and the techniques described in “Alternatives to the `eval` Function” on page 2-108.

More About

- “Base and Function Workspaces” on page 14-29

Calling Functions

In this section...
“What Happens When You Call a Function” on page 14-35
“Function Precedence Order” on page 14-35
“Resolving Difficulties In Calling Functions” on page 14-38
“Calling External Functions” on page 14-43
“Running External Programs” on page 14-43

What Happens When You Call a Function

When you call a function from either the command line or from within another program 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 the `clear` function, or until you quit MATLAB.

Clearing Functions from Memory

You can use `clear` in any of the following ways to remove functions from the MATLAB workspace.

Syntax	Description
<code>clear functionname</code>	Remove specified function from workspace.
<code>clear functions</code>	Remove all compiled functions.
<code>clear all</code>	Remove all variables and functions.

Function Precedence Order

This topic explains how MATLAB determines which function to call when multiple functions in the current scope have the same name. The current scope includes the current file, an optional private subfolder relative to the currently running function, the current folder, and the MATLAB path.

MATLAB uses this precedence order:

1 Variables

Before assuming that a name matches a function, MATLAB checks for a variable with that name in the current workspace.

Note If you create a variable with the same name as a function, MATLAB cannot run that function until you clear the variable from memory.

2 Nested functions within the current function

3 Local functions within the current file

4 Private functions

Private functions are functions in a subfolder named `private` that is immediately below the folder of the currently running file.

5 Class constructors in @ folders

MATLAB uses class constructors to create a variety of objects (such as `timeseries` or `audioplayer`), and you can define your own classes using object-oriented programming. For example, if you create a class folder `@polynom` and a constructor function `@polynom/polynom.m`, the constructor takes precedence over other functions named `polynom.m` anywhere on the path.

6 Overloaded methods

Overloading refers to intentionally reusing the same name. Overloaded methods typically implement similar functionality for different data types or classes. MATLAB checks the classes of the input arguments to determine which method to use.

7 Functions in the current folder

8 Functions elsewhere on the path, in order of appearance

When determining the precedence of functions within the same folder, MATLAB considers the file type, in this order:

- 1** Built-in function
- 2** MEX-function
- 3** Simulink® model, with file types in this order:
 - a** SLX file
 - b** MDL file
- 4** P-file (that is, an encoded program file with a .p extension)
- 5** Program file with a .m extension

For example, if MATLAB finds a .m file and a P-file with the same name in the same folder, it uses the P-file. Because P-files are not automatically regenerated, make sure that you regenerate the P-file whenever you edit the program file.

To determine the function MATLAB calls for a particular input, include the function name and the input in a call to the `which` function. For example, determine the location of the `max` method that MATLAB calls for double and `int8` values:

```
testval = 10;  
which max(testval)
```

```
built-in (matlabroot\toolbox\matlab\datafun\@double\max)  
% double method
```

```
testval = int8(10);  
which max(testval)
```

```
built-in (matlabroot\toolbox\matlab\datafun\@int8\max)  
% int8 method
```

For more information, see:

- “What Is the Search Path?”
- Variables
- “Nested Functions” on page 15-16

- “Subfunctions” on page 15-33
- “Private Functions” on page 15-35

Resolving Difficulties In Calling Functions

The two most common problems related to invoking functions in MATLAB are:

- “Conflicting Function and Variable Names” on page 14-38
- “Undefined Functions or Variables” on page 14-38

Conflicting Function and Variable Names

MATLAB throws an error if a variable and function have been given the same name and there is insufficient information available for MATLAB to resolve the conflict. You may see an error message something like the following:

```
Error: <functionName> was previously used as a variable,  
      conflicting with its use here as the name of a function  
      or command.
```

where <functionName> is the name of the function.

Certain uses of the `eval` and `load` functions can also result in a similar conflict between variable and function names. For more information, see:

- “Troubleshooting: Loading Variables within a Function”
- “Alternatives to the `eval` Function” on page 2-108

Undefined Functions or Variables

You may encounter the following error message, or something similar, while working with functions or variables in MATLAB:

```
??? Undefined function or variable 'x'.
```

These errors usually indicate that MATLAB cannot find a particular variable or MATLAB program file in the current directory or on the search path. The root cause is likely to be one of the following:

- The name of the function has been misspelled.

- The function name and name of the file containing the function are not the same.
- The toolbox to which the function belongs is not installed.
- The search path to the function has been changed.
- The function is part of a toolbox that you do not have a license for.

Follow the steps described in this section to resolve this situation.

Verify that You Have the Correct Spelling of the Function Name. One of the first things to check when you are unable to invoke a function is the spelling of the function name. Especially with longer function names or names containing similar characters (e.g., letter l and numeral one), it is easy to make an error that is not easily detected.

An unrecognized function name may also be the result of case mismatch. For example, the name of the MATLAB function `accumarray` contains lowercase letters only. The following command fails because it includes an uppercase letter in the function name:

```
accumArray
??? Undefined function or variable 'accumArray'.
```

Using the alphabetical or categorized function lists in the MATLAB Help Browser can help you find the correct spelling.

Make Sure the Function Name Matches the File Name. You establish the name for a function when you write its function definition line. This name should always match the name of the file you save it to. For example, if you create a function named `curveplot`,

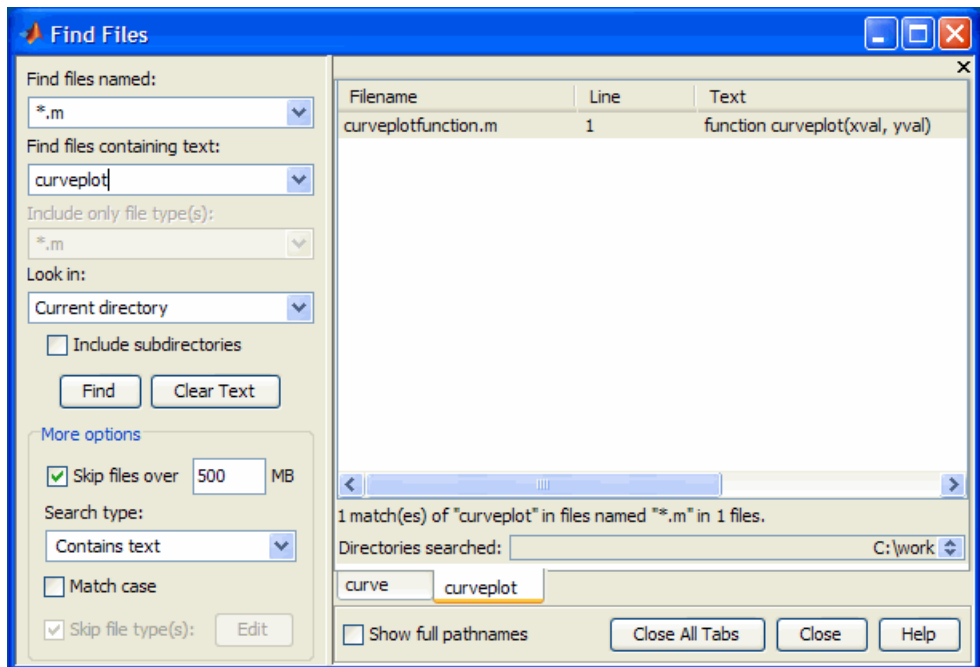
```
function curveplot(xVal, yVal)
    - program code -
```

then you should name the file containing that function `curveplot.m`. If you create a `pcode` file for the function, then name that file `curveplot.p`. In the case of conflicting function and file names, the file name overrides the name given to the function. In this example, if you save the `curveplot` function to a file named `curveplotfunction.m`, then attempts to invoke the function using the function name will fail:

```
curveplot
??? Undefined function or variable 'curveplot'.
```

If you encounter this problem, change either the function name or file name so that they are the same. If you have difficulty locating the file that uses this function, use the MATLAB **Find Files** utility as follows:

- 1 Open the **Find Files** dialog box by clicking **Edit > Find Files** in MATLAB.
- 2 Under **Find files named:** enter `*.m`
- 3 Under **Find files containing text:** enter the function name.
- 4 Click the **Find** button



Make Sure the Toolbox Is Installed. If you are unable to use a built-in function from MATLAB or its toolboxes, make sure that the function is installed. If you do not know which toolbox supports the function you need, reference the following list:

http://www.mathworks.com/support/functions/alpha_list.html

Once you know which toolbox the function belongs to, use the `ver` function to see which toolboxes are installed on the system from which you run MATLAB. The `ver` function displays a list of all currently installed MathWorks products. If you can locate the toolbox you need in the output displayed by `ver`, then the toolbox is installed. For help with installing MathWorks products, see the Installation Guide documentation.

If you do not see the toolbox and you believe that it is installed, then perhaps the MATLAB path has been set incorrectly. Go on to the next section.

Verify the Path Used to Access the Function. This step resets the path to the default. Because MATLAB stores the toolbox information in a cache file, you will need to first update this cache and then reset the path. To do this,

- 1** Go to the **File** menu and select **Preferences...**
- 2** Go to the **General** heading. Click the button **Update Toolbox Path Cache** and press **OK**.
- 3** Go to the **File** menu and select **Set Path...**
- 4** Click **Default**, and a small dialog box opens warning that you will lose your current path settings if you proceed. Click **Yes** if you decide to proceed, and then click **OK** and then **Save** to finish.

(If you have added any custom paths to MATLAB, you will need to restore those later)

Run `ver` again to see if the toolbox is installed. If not, you may need to reinstall this toolbox to use this function. See the Related Solution 1-1CBD3, "How do I install additional toolboxes into my existing MATLAB" for more information about installing a toolbox.

Once `ver` shows your toolbox, run the following command to see if you can find the function:

```
which -all <functionname>
```

replacing <functionname> with the name of the function. You should be presented with the path(s) of the function file. If you get a message indicating that the function name was not found, you may need to reinstall that toolbox to make the function active.

Verify that Your License Covers The Toolbox. If you receive the error message “Has no license available”, there is a licensing related issue preventing you from using the function. To find the error that is occurring, you can use the following command:

```
license checkout <toolbox_license_key_name>
```

replacing <toolbox_license_key_name> with the proper key name for the toolbox that contains your function. To find the license key name, look at the INCREMENT lines in your license file. For information on how to find your license file see the related solution: 1-63ZIR6, “Where are the license files for MATLAB located?”

The license key names of all the toolboxes are located after each INCREMENT tag in the license.dat file. For example:

```
INCREMENT MATLAB MLM 17 00-jan-0000 0 k  
B454554BADECED4258 \HOSTID=123456 SN=123456
```

If your license.dat file has no INCREMENT lines, refer to your license administrator for them. For example, to test the licensing for Symbolic Math Toolbox, you would run the following command:

```
license checkout Symbolic_Toolbox
```

A correct testing gives the result "ANS=1". An incorrect testing results in an error from the license manager. You can either troubleshoot the error by looking up the license manager error here:

<http://www.mathworks.com/support/install.html>

or you can contact the Installation Support Team with the error here:

http://www.mathworks.com/support/contact_us/index.html

When contacting support, provide your license number, your MATLAB version, the function you are using, and the license manager error (if applicable).

Calling External Functions

The MATLAB external interface offers a number of ways to run external functions from MATLAB. This includes programs written in C or Fortran, methods invoked on Sun Java or COM (Component Object Model) objects, functions that interface with serial port hardware, and functions stored in shared libraries.

Running External Programs

For information on how to invoke operating systems commands or execute programs that are external to MATLAB, see [Running External Programs](#).

Functions Provided By MATLAB

In this section...

“Overview” on page 14-44

“Functions” on page 14-44

“Built-In Functions” on page 14-45

“Overloaded MATLAB Functions” on page 14-46

“Internal Utility Functions” on page 14-47

Overview

Many of the functions provided with the MATLAB software are implemented as program files just like the files you create with MATLAB. Other MATLAB functions are precompiled executable programs called *built-ins* that run much more efficiently. Many MATLAB functions are also overloaded so that they handle different classes appropriately.

Functions

If you look in the subfolders of the `toolbox\matlab` folder, you can find the sources to many of the functions supplied with MATLAB. Locate your `toolbox\matlab` folder by typing

```
dir([matlabroot '\toolbox\matlab\'])
```

Any MATLAB functions that you write are just like any other functions coded with MATLAB. When one of these functions is called, MATLAB parses and executes each line of code in the file. It saves the parsed version of the function in memory, eliminating parsing time on any further calls to this function.

Identifying Functions

To find out if a function is implemented with a program file, use the `exist` function. The `exist` function searches for the name you enter on the MATLAB path and returns a number identifying the source. If the source is a file with a `.m` file extension, then `exist` returns the number 2. This example identifies the source for the `repmat` function as a program file:

```
exist repmat
ans =
     2
```

The `exist` function also returns 2 for files that have a file type unknown to MATLAB. However, if you invoke `exist` on a MATLAB function name, the file type is known to MATLAB and returns 2 only on program files.

Viewing the Source Code

One advantage of functions implemented as files is that you can look at the source code. This can help when you need to understand why the function returns a value you did not expect, if you need to figure out how to code something in MATLAB that is already coded in a function, or perhaps to help you create a function that overloads one of the MATLAB functions.

To find the source code for any MATLAB function, use `which`. For example,

```
which repmat
```

returns the full path for the program file:

```
D:\MATLAB\toolbox\matlab\elmat\repmat.m
```

Built-In Functions

Functions that are frequently used or that can take more time to execute are often implemented as executable files. These functions are called *built-ins*.

Unlike MATLAB program file functions, you cannot see the source code for built-ins. Although most built-in functions do have a program file associated with them, this file is there mainly to supply the help documentation for the function. For example, view the program file for the `reshape` function:

```
type reshape.m
```

The file contains only help text.

Identifying Built-In Functions

As with program file functions, you can identify which functions are built-ins using the `exist` function. This function identifies built-ins by returning the number 5:

```
exist reshape
ans =
     5
```

Forcing a Built-In Call

If you overload any of the MATLAB built-in functions to handle a specific class, then MATLAB always calls the overloaded function on that type. If, for some reason, you need to call the built-in version, you can override the usual calling mechanism using the `builtin` function. The expression

```
builtin('reshape', arg1, arg2, ..., argN);
```

forces a call to the MATLAB built-in function, `reshape`, passing the arguments shown even though an overload exists for the class in this argument list.

Note With the exception of overloading, you should not create a MATLAB program file that has the same name as a MATLAB built-in. Because built-in functions have a higher precedence than most other types of program files (with the exception of private and subfunctions), MATLAB does not recognize functions that share the same name with a built-in.

Overloaded MATLAB Functions

An *overloaded function* is an additional implementation of an existing function that is designed specifically to handle a certain class. When you pass an argument of this type in a call to the function, MATLAB looks for the function implementation that handles that type and executes that function code.

Each overloaded MATLAB function has a file on the MATLAB path. The files for a certain class reside in a folder named with an @ sign followed by the class name. For example, if you need to plot expressions of class `polynom` in a manner that is unique to that class, you can overload the MATLAB `plot`

function. To do this, create your own plotting function `plot.m` specifically for use with objects of the `polynom` class. Then, create a folder called `@polynom`, and store your own version of `plot.m` in that folder.

You can add your own overloads to any function. Just create a class folder for the class you want to support for that function, and create a file that handles the type in a manner different from the default. See [Defining Classes — Syntax and Developing Classes — Typical Workflow](#).

When you use the `which` command with the `-all` option, MATLAB returns all occurrences of the file you are looking for. This is an easy way to find functions that are overloaded:

```
which -all set           % Show all implementations for 'set'
```

Internal Utility Functions

MathWorks reserves the use of packages named `internal` for utility functions used by internal MATLAB code. Functions that belong to an `internal` package are intended for MathWorks use only. Using functions that belong to an `internal` package is strongly discouraged. These functions are not guaranteed to work in a consistent manner from one release to the next. In fact, any of these functions and classes could be removed from the MATLAB software in any subsequent release without notice and without documentation in the product release notes.

Any function called with a syntax that begins with the package name `internal` is an internal function. For example,

```
internal.matlab.functionname
```

Any function on the MATLAB path that resides at any level under a folder named `+internal` is an internal function. For example,

```
matlab\toolbox\matlab\+internal\functionname
```


Types of Functions

- “Overview of MATLAB Function Types” on page 15-2
- “Anonymous Functions” on page 15-3
- “Primary Functions” on page 15-15
- “Nested Functions” on page 15-16
- “Subfunctions” on page 15-33
- “Private Functions” on page 15-35
- “Overloaded Functions” on page 15-37

Overview of MATLAB Function Types

There are essentially two ways to create a new function for your MATLAB application: in a command entered at run-time, or in a file saved to permanent storage.

The command-oriented function, called an *anonymous function*, is relatively brief in its content. It consists of a single MATLAB statement that can interact with multiple input and output arguments. The benefit of using anonymous functions is that you do not have to edit and maintain a file for functions that require only a brief definition.

There are several types of functions that you write and execute as a file. The most basic of these are *primary functions* and *subfunctions*. Primary functions are visible to other functions outside of the file they are defined in, while subfunctions, generally speaking, are not. That is, you can call a primary function from an anonymous function or from a function defined in a separate file, but you can call a subfunction only from functions within the same file. (See the Description section of the `function_handle` reference page for information on making a subfunction externally visible.)

Two specific types of primary functions are the *private* and *overloaded function*. Private functions are visible only to a limited group of other functions. This type of function can be useful if you want to limit access to a function, or when you choose not to expose the implementation of a function. Overloaded functions act the same way as overloaded functions in most computer languages. You can create multiple implementations of a function so that each responds accordingly to different types of inputs.

The last type of MATLAB function is the *nested function*. Nested functions are not an independent function type; they exist within the body of one of the other types of functions discussed here (with the exception of anonymous functions), and also within other nested functions.

Anonymous Functions

In this section...

“Constructing an Anonymous Function” on page 15-3

“Arrays of Anonymous Functions” on page 15-6

“Outputs from Anonymous Functions” on page 15-7

“Variables Used in the Expression” on page 15-8

“Examples of Anonymous Functions” on page 15-11

Constructing an Anonymous Function

Anonymous functions give you a quick means of creating simple functions without having to store your function to a file each time. You can construct an anonymous function either at the MATLAB command line or in any function or script.

The syntax for creating an anonymous function from an expression is

```
fhandle = @(arglist) expr
```

Starting from the right of this syntax statement, the term `expr` represents the body of the function: the code that performs the main task your function is to accomplish. This consists of any single, valid MATLAB expression. Next is `arglist`, which is a comma-separated list of input arguments to be passed to the function. These two components are similar to the body and argument list components of any function.

Leading off the entire right side of this statement is an @ sign. The @ sign is the MATLAB operator that constructs a function handle. Creating a function handle for an anonymous function gives you a means of invoking the function. It is also useful when you want to pass your anonymous function in a call to some other function. The @ sign is a required part of an anonymous function definition.

Note Function handles not only provide access to anonymous functions. You can create a function handle to any MATLAB function. The constructor uses a different syntax: `fhandle = @functionname` (e.g., `fhandle = @sin`). To find out more about function handles, see Chapter 9, “Function Handles”.

The syntax statement shown above constructs the anonymous function, returns a handle to this function, and stores the value of the handle in variable `fhandle`. You can use this function handle in the same way as any other MATLAB function handle.

Simple Example

The statement below creates an anonymous function that finds the square of a number. When you call this function, MATLAB assigns the value you pass in to variable `x`, and then uses `x` in the equation `x.^2`:

```
sqr = @(x) x.^2;
```

The `@` operator constructs a function handle for this function, and assigns the handle to the output variable `sqr`. As with any function handle, you execute the function associated with it by specifying the variable that contains the handle, followed by a comma-separated argument list in parentheses. The syntax is

```
fhandle(arg1, arg2, ..., argN)
```

To execute the `sqr` function defined above, type

```
a = sqr(5)
a =
    25
```

Because `sqr` is a function handle, you can pass it in an argument list to other functions. The code shown here passes the `sqr` anonymous function to the MATLAB `quad` function to compute its integral from zero to one:

```
quad(sqr, 0, 1)
ans =
    0.3333
```

A Two-Input Example

As another example, you could create the following anonymous function that uses two input arguments, x and y . Variables A and B are already defined:

```
A = [2 3 4];    B = [5 6 7];
sumAxBy = @(x, y) (A*x + B*y);
```

```
whos sumAxBy
Name          Size          Bytes  Class

sumAxBy      1x1              16  function_handle
```

To call this function, assigning 5 to x and 7 to y , type

```
sumAxBy(5, 7)
```

Evaluating With No Input Arguments

For anonymous functions that do not take any input arguments, construct the function using empty parentheses for the input argument list:

```
t = @() datestr(now);
```

Also use empty parentheses when invoking the function:

```
t()

ans =
04-Sep-2003 10:17:59
```

You must include the parentheses. If you type the function handle name with no parentheses, MATLAB just identifies the handle; it does not execute the related function:

```
t

t =
    @() datestr(now)
```

Arrays of Anonymous Functions

To store multiple anonymous functions in an array, use a cell array. The example shown here stores three simple anonymous functions in cell array A:

```
A = {@(x)x.^2, @(y)y+10, @(x,y)x.^2+y+10}
A =
    @(x)x.^2    @(y)y+10    @(x,y)x.^2+y+10
```

Execute the first two functions in the cell array by referring to them with the usual cell array syntax, A{1} and A{2}:

```
A{1}(4) + A{2}(7)
ans =
    33
```

Do the same with the third anonymous function that takes two input arguments:

```
A{3}(4, 7)
ans =
    33
```

Space Characters in Anonymous Function Elements

Note that while using space characters in the definition of any function can make your code easier to read, spaces in the body of an anonymous function that is defined in a cell array can sometimes be ambiguous to MATLAB. To ensure accurate interpretation of anonymous functions in cell arrays, you can do any of the following:

- Remove all spaces from at least the body (not necessarily the argument list) of each anonymous function:

```
A = {@(x)x.^2, @(y)y+10, @(x, y)x.^2+y+10};
```

- Enclose in parentheses any anonymous functions that include spaces:

```
A = {(@(x)x .^ 2), (@(y) y +10), (@(x, y) x.^2 + y+10)};
```

- Assign each anonymous function to a variable, and use these variable names in creating the cell array:


```
A1 = @(x)x.^2; A2 = @(y) y +10; A3 = @(x, y)x.^2 + y+10;  
A = {A1, A2, A3};
```

Outputs from Anonymous Functions

As with other MATLAB functions, the number of outputs returned by an anonymous function depends mainly on how many variables you specify to the left of the equals (=) sign when you call the function.

For example, consider an anonymous function `getPersInfo` that returns a person's address, home phone, business phone, and date of birth, in that order. To get someone's address, you can call the function specifying just one output:

```
address = getPersInfo(name);
```

To get more information, specify more outputs:

```
[address, homePhone, busPhone] = getPersInfo(name);
```

Of course, you cannot specify more outputs than the maximum number generated by the function, which is four in this case.

Example

The anonymous `getXLSData` function shown here calls the MATLAB `xlsread` function with a preset spreadsheet filename (`records.xls`) and a variable worksheet name (`worksheet`):

```
getXLSData = @(worksheet) xlsread('records.xls', worksheet);
```

The `records.xls` worksheet used in this example contains both numeric and text data. The numeric data is taken from instrument readings, and the text data describes the category that each numeric reading belongs to.

Because the MATLAB `xlsread` function is defined to return up to three values (numeric, text, and raw data), `getXLSData` can also return this same number of values, depending on how many output variables you specify to the left of the equals sign in the call. Call `getXLSData` a first time, specifying only a single (numeric) output, `dNum`:

```
dNum = getXLSData('Week 12');
```

Display the data that is returned using a for loop. You have to use generic names (v1, v2, v3) for the categories due to the fact that the text of the real category names was not returned in the call:

```
for k = 1:length(dNum)
    disp(sprintf('%s    v1: %2.2f    v2: %d    v3: %d', ...
        datestr(clock, 'HH:MM'), dNum(k,1), dNum(k,2), ...
        dNum(k,3)));
end
```

Here is the output from the first call:

```
12:55    v1: 78.42    v2: 32    v3: 37
13:41    v1: 69.73    v2: 27    v3: 30
14:26    v1: 77.65    v2: 17    v3: 16
15:10    v1: 68.19    v2: 22    v3: 35
```

Now try this again, but this time specifying two outputs, numeric (dNum) and text (dTxt):

```
[dNum, dTxt] = getXLSData('Week 12');

for k = 1:length(dNum)
    disp(sprintf('%s    %s: %2.2f    %s: %d    %s: %d', ...
        datestr(clock, 'HH:MM'), dTxt{1}, dNum(k,1), ...
        dTxt{2}, dNum(k,2), dTxt{3}, dNum(k,3)));
end
```

This time, you can display the category names returned from the spreadsheet:

```
12:55    Temp: 78.42    HeatIndex: 32    WindChill: 37
13:41    Temp: 69.73    HeatIndex: 27    WindChill: 30
14:26    Temp: 77.65    HeatIndex: 17    WindChill: 16
15:10    Temp: 68.19    HeatIndex: 22    WindChill: 35
```

Variables Used in the Expression

Anonymous functions commonly include two types of variables:

- Variables specified in the argument list. These often vary with each function call.

- Variables specified in the body of the expression. MATLAB captures these variables and holds them constant throughout the lifetime of the function handle.

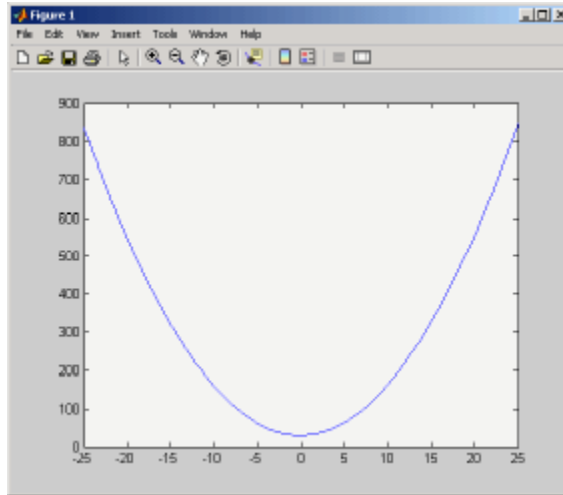
The latter variables must have a value assigned to them at the time you construct an anonymous function that uses them. Upon construction, MATLAB captures the current value for each variable specified in the body of that function. The function will continue to associate this value with the variable even if the value should change in the workspace or go out of scope.

The fact that MATLAB captures the values of these variables when the handle to the anonymous function is constructed enables you to execute an anonymous function from anywhere in the MATLAB environment, even outside the scope in which its variables were originally defined. But it also means that to supply new values for any variables specified within the expression, you must reconstruct the function handle.

Changing Variables Used in an Anonymous Function

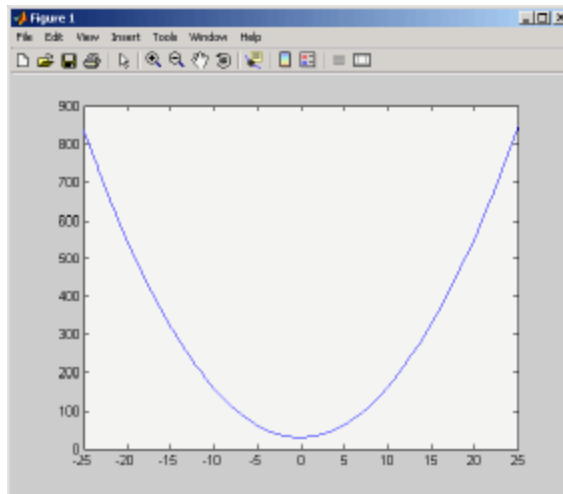
The second statement shown below constructs a function handle for an anonymous function called `parabola` that uses variables `a`, `b`, and `c` in the expression. Passing the function handle to the MATLAB `fplot` function plots it out using the initial values for these variables:

```
a = 1.3;    b = .2;    c = 30;
parabola = @(x) a*x.^2 + b*x + c;
fplot(parabola, [-25 25])
```



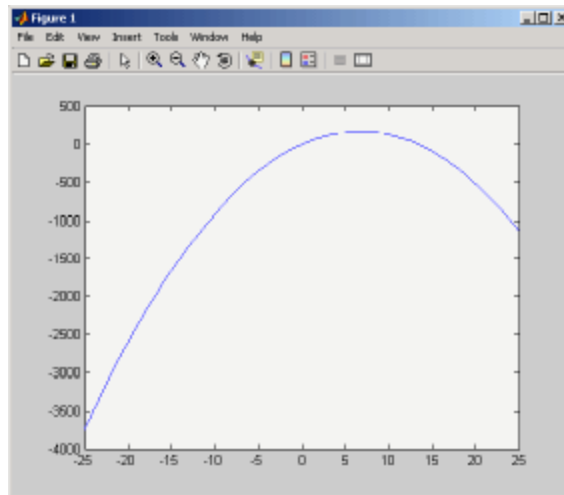
If you change the three variables in the workspace and replot the figure, the parabola remains unchanged because the parabola function is still using the initial values of a, b, and c:

```
a = -3.9;    b = 52;    c = 0;  
fplot(parabola, [-25 25])
```



To get the function to use the new values, you need to reconstruct the function handle, causing MATLAB to capture the updated variables. Replot using the new construct, and this time the parabola takes on the new values:

```
a = -3.9;    b = 52;    c = 0;
parabola = @(x) a*x.^2 + b*x + c;
fplot(parabola, [-25 25])
```



For the purposes of this example, there is no need to store the handle to the anonymous function in a variable (`parabola`, in this case). You can just construct and pass the handle right within the call to `fplot`. In this way, you update the values of `a`, `b`, and `c` on each call:

```
fplot(@(x) a*x.^2 + b*x + c, [-25 25])
```

Examples of Anonymous Functions

This section shows a few examples of how you can use anonymous functions. These examples are intended to show you how to program with this type of function.

The examples in this section include:

- “Example 1 — Passing a Function to `quad`” on page 15-12

- “Example 2 — Multiple Anonymous Functions” on page 15-13

Example 1 — Passing a Function to quad

The equation shown here has one variable t that can vary each time you call the function, and two additional variables, g and ω . Leaving these two variables flexible allows you to avoid having to hardcode values for them in the function definition:

$$x = g * \cos(\omega * t)$$

One way to program this equation is to write a function, and then create a function handle for it so that you can pass the function to other functions, such as the MATLAB `quad` function as shown here. However, this requires creating and maintaining a new file for a purpose that is likely to be temporary, using a more complex calling syntax when calling `quad`, and passing the g and ω parameters on every call. Here is the function:

```
function f = vOut(t, g, omega)
f = g * cos(omega * t);
```

This code has to specify g and ω on each call:

```
g = 2.5; omega = 10;

quad(@vOut, 0, 7, [], [], g, omega)
ans =
    0.1935

quad(@vOut, -5, 5, [], [], g, omega)
ans =
   -0.1312
```

You can simplify this procedure by setting the values for g and ω just once at the start, constructing a function handle to an anonymous function that only lasts the duration of your MATLAB session, and using a simpler syntax when calling `quad`:

```
g = 2.5; omega = 10;
f = @(t) (g * cos(omega * t));
```

```
quad(f, 0, 7)
ans =
    0.1935
```

```
quad(f, -5, 5)
ans =
   -0.1312
```

To preserve an anonymous function from one MATLAB session to the next, save the function handle to a MAT-file

```
save anon.mat f
```

and then load it into the MATLAB workspace in a later session:

```
load anon.mat f
```

Example 2 – Multiple Anonymous Functions

This example solves the following equation by combining two anonymous functions:

$$g(c) = \int_0^1 (x^2 + cx + 1) dx$$

The equivalent anonymous function for this expression is

```
g = @(c) (quad(@(x) (x.^2 + c*x + 1), 0, 1));
```

This was derived as follows. Take the parenthesized part of the equation (the integrand) and write it as an anonymous function. You do not need to assign the output to a variable as it will only be passed as input to the `quad` function:

```
@(x) (x.^2 + c*x + 1)
```

Next, evaluate this function from zero to one by passing the function handle, shown here as the entire anonymous function, to `quad`. You need to temporarily set `c` to some value to test this:

```
c = 2;
```

```
quad(@(x) (x.^2 + c*x + 1), 0, 1)
ans =
    2.3333
```

Supply the value for **c** by constructing an anonymous function for the entire equation and you are done:

```
g = @(c) (quad(@(x) (x.^2 + c*x + 1), 0, 1));

g(2)
ans =
    2.3333
```


Primary Functions

The first function in any MATLAB program file is called the *primary function*. Following the primary function can be any number of subfunctions, which can serve as subroutines to the primary function.

Under most circumstances, the primary function is the only function in the file that you can call from the MATLAB command line or from another function. You invoke this function using the name of the file in which it is defined.

For example, the average function shown here resides in the file `average.m`:

```
function y = average(x)
% AVERAGE Mean of vector elements.

y = sum(x)/length(x);      % Actual computation
```

You can invoke this function from the MATLAB command line with this command to find the average of three numbers:

```
average([12 60 42])
```

Note that it is customary to give the primary function the same name as the file in which it resides. If the function name differs from the filename, then you must use the filename to invoke the function.

Nested Functions

In this section...

“Writing Nested Functions” on page 15-16

“Calling Nested Functions” on page 15-18

“Variable Scope in Nested Functions” on page 15-19

“Using Function Handles with Nested Functions” on page 15-21

“Restrictions on Assigning to Variables” on page 15-26

“Examples of Nested Functions” on page 15-27

Writing Nested Functions

You can define one or more functions within another function in your MATLAB application. These inner functions are said to be *nested* within the function that contains them. You can also nest functions within other nested functions. You cannot however define a nested function inside any of the MATLAB program control statements. This includes any block of code that is controlled by an `if/elseif/else`, `switch`, `for`, `while`, or `try/catch` statement.

To write a nested function, simply define one function within the body of another function in your program. Like any function, a nested function contains any or all of the components described in “Basic Parts of a Program File” on page 14-10. In addition, you must always terminate a nested function with an `end` statement:

```
function x = A(p1, p2)
...
    function y = B(p3)
        ...
    end
...
end
```

Note Functions do not normally require a terminating `end` statement. This rule does not hold, however, when you nest functions. If a program file contains one or more nested functions, you must terminate *all* functions (including subfunctions) in the file with `end`, whether or not they contain nested functions.

Example – More Than One Nested Function

This example shows function A and two additional functions nested inside A at the same level:

```
function x = A(p1, p2)
...
    function y = B(p3)
        ...
    end

    function z = C(p4)
        ...
    end
...
end
```

Example – Multiply Nested Functions

This example shows multiply nested functions, C nested inside B, and B in A:

```
function x = A(p1, p2)
...
    function y = B(p3)
        ...
        function z = C(p4)
            ...
        end
    end
...
end
...
end
```

Calling Nested Functions

You can call a nested function

- From the level immediately above it. (In the following code, function A can call B or D, but not C or E.)
- From a function nested at the same level within the same parent function. (Function B can call D, and D can call B.)
- From a function at any lower level. (Function C can call B or D, but not E.)

```
function A(x, y)                % Primary function
B(x, y);
D(y);

    function B(x, y)            % Nested in A
C(x);
D(y);

        function C(x)          % Nested in B
D(x);
end
end

    function D(x)              % Nested in A
E(x);

        function E(x)          % Nested in D
...
end
end
end
```

You can also call a subfunction from any nested function in the same file.

You can pass variable numbers of arguments to and from nested functions, but you should be aware of how MATLAB interprets `varargin`, `varargout`, `nargin`, and `nargout` under those circumstances. See "Passing Optional Arguments to Nested Functions" for more information on this.

Note If you construct a function handle for a nested function, you can call the nested function from any MATLAB function that has access to the handle. See “Using Function Handles with Nested Functions” on page 15-21.

Nested functions are not accessible to the `str2func` or `feval` function. You cannot call a nested function using a handle that has been constructed with `str2func`. And, you cannot call a nested function by evaluating the function name with `feval`. To call a nested function, you must either call it directly by name, or construct a function handle for it using the `@` operator.

Variable Scope in Nested Functions

The scope of a variable is the range of functions that have direct access to the variable to set, modify, or acquire its value. When you define a local (i.e., nonglobal) variable within a function, its scope is normally restricted to that function alone. For example, subfunctions do not share variables with the primary function or with other subfunctions. This is because each function and subfunction stores its variables in its own separate workspace.

Like other functions, a nested function has its own workspace. But it also has access to the workspaces of all functions in which it is nested. So, for example, a variable that has a value assigned to it by the primary function can be read or overwritten by a function nested at any level within the primary. Similarly, a variable that is assigned in a nested function can be read or overwritten by any of the functions containing that function.

In the following two examples, variable `x` is stored in the workspace of the outer `varScope` function and can be read or written to by all functions nested within it.

<pre>function varScope1 x = 5; nestfun1 function nestfun1 nestfun2 function nestfun2 x = x + 1 end end end</pre>	<pre>function varScope2 nestfun1 function nestfun1 nestfun2 function nestfun2 x = 5; end end x = x + 1 end</pre>
---	---

As a rule, a variable used or defined within a nested function resides in the workspace of the outermost function that both contains the nested function and accesses that variable. The scope of this variable is then the function to which this workspace belongs, and all functions nested to any level within that function.

In the next example, the outer function, `varScope3`, does not access variable `x`. Following the rule just stated, `x` is unknown to the outer function and thus is not shared between the two nested functions. In fact, there are two separate `x` variables in this example: one in the function workspace of `nestfun1` and one in the function workspace of `nestfun2`. When `nestfun2` attempts to update `x`, it fails because `x` does not yet exist in this workspace:

```
function varScope3
nestfun1
nestfun2

    function nestfun1
        x = 5;
    end

    function nestfun2
        x = x + 1
    end
end
```

The Scope of Output Variables

Variables containing values returned by a nested function are not in the scope of outer functions. In the two examples shown here, the one on the left fails in the second to last line because, although the *value* of *y* is returned by the nested function, the *variable* *y* is local to the nested function, and unknown to the outer function. The example on the right assigns the return value to a variable, *z*, and then displays the value of *z* correctly.

Incorrect	Correct
<pre>function varScope4 x = 5; nestfun; function y = nestfun y = x + 1; end y end</pre>	<pre>function varScope5 x = 5; z = nestfun; function y = nestfun y = x + 1; end z end</pre>

Using Function Handles with Nested Functions

Every function has a certain *scope*, that is, a certain range of other functions to which it is visible. A function's scope determines which other functions can call it. You can call a function that is out of scope by providing an alternative means of access to it in the form of a function handle. (The function handle, however, must be within the scope of its related function when you construct the handle.) Any function that has access to a function handle can call the function with which the handle is associated.

Note Although you can call an out of scope function by means of a function handle, the handle itself must be within the scope of its related function at the time it is constructed.

The section on “Calling Nested Functions” on page 15-18 defines the scope of a nested function. As with other types of functions, you can make a nested function visible beyond its normal scope with a function handle. The following function `getCubeHandle` constructs a handle for nested function `findCube`

and returns its handle, `h`, to the caller. The `@` sign placed before a function name (e.g., `@findCube`) is the MATLAB operator that constructs a handle for that function:

```
function h = getCubeHandle
h = @findCube;           % Function handle constructor

    function cube = findCube(X) % Nested function
        cube = X .^ 3;
    end
end
```

Call `getCubeHandle` to obtain the function handle to the nested function `findCube`. Assign the function handle value returned by `getCubeHandle` to an output variable, `cubeIt` in this case:

```
cubeIt = getCubeHandle;
```

You can now use this variable as a means of calling `findCube` from outside of its program file:

```
cubeIt(8)
ans =
    512
```

Note When calling a function by means of its handle, use the same syntax as if you were calling a function directly. But instead of calling the function by its name (e.g., `strcmp(S1, S2)`), use the variable that holds the function handle (e.g., `fhandle(S1, S2)`).

Function Handles and Nested Function Variables

One characteristic of nested functions that makes them different from other MATLAB functions is that they can share nonglobal variables with certain other functions in the same file. A nested function `nFun` can share variables with any outer function that contains `nFun`, and with any function nested within `nFun`. This characteristic has an impact on how certain variables are stored when you construct a handle for a nested function.

Defining Variables When Calling Via Function Handle. The example below shows a primary function `getHandle` that returns a function handle for the nested function `nestFun`. The `nestFun` function uses three different types of variables. The `VLoc` variable is local to the nested function, `VInp` is passed in when the nested function is called, and `VExt` is defined by the outer function:

```
function h = getHandle(X)
h = @nestFun;
VExt = someFun(X);

    function nestFun(VInp)
        VLoc = 173.5;
        doSomeTask(VInp, VLoc, VExt);
    end
end
```

As with any function, when you call `nestFun`, you must ensure that you supply the values for any variables it uses. This is a straightforward matter when calling the nested function directly (that is, calling it from `getHandle`). `VLoc` has a value assigned to it within `nestFun`, `VInp` has its value passed in, and `VExt` acquires its value from the workspace it shares with `getHandle`.

However, when you call `nestFun` using a function handle, only the nested function executes; the outer function, `getHandle`, does not. It might seem at first that the variable `VExt`, otherwise given a value by `getHandle`, has no value assigned to it in the case. What in fact happens though is that MATLAB stores variables such as `VExt` inside the function handle itself when it is being constructed. These variables are available for as long as the handle exists.

The `VExt` variable in this example is considered to be *externally scoped* with respect to the nested function. Externally scoped variables that are used in nested functions for which a function handle exists are stored within the function handle. So, function handles not only contain information about accessing a function. For nested functions, a function handle also stores the values of any externally scoped variables required to execute the function.

Example Using Externally Scoped Variables

The `sCountFun` and `nCountFun` functions shown below return function handles for subfunction `subCount` and nested function `nestCount`, respectively.

These two inner functions store a persistent value in memory (the value is retained in memory between function calls), and then increment this value on every subsequent call. `subCount` makes its count value persistent with an explicit persistent declaration. In `nestCount`, the count variable is externally scoped and thus is maintained in the function handle:

Using a Subfunction	Using a Nested Function
<pre>function h = sCountFun(X) h = @subCount; count = X subCount(0, count); function subCount(incr, ini) persistent count; initializing = nargin > 1; if initializing count = ini; else count = count + incr end end</pre>	<pre>function h = nCountFun(X) h = @nestCount; count = X function nestCount(incr) count = count + incr end end end</pre>

When `sCountFun` executes, it passes the initial value for `count` to the `subCount` subfunction. Keep in mind that the `count` variable in `sCountFun` is not the same as the `count` variable in `subCount`; they are entirely independent of each other. Whenever `subCount` is called via its function handle, the value for `count` comes from its persistent place in memory.

In `nestCount`, the `count` variable again gets its value from the primary function when called from within the file. However, in this case the `count` variable in the primary and nested functions are one and the same. When `nestCount` is called by means of its function handle, the value for `count` is assigned from its storage within the function handle.

Running the Example. The `subCount` and `nestCount` functions increment a value in memory by another value that you pass as an input argument. Both of these functions give the same results.

Get the function handle to `nestCount`, and initialize the `count` value to a four-element vector:

```
h = nCountFun([100 200 300 400])
```

```
count =
    100  200  300  400
```

Increment the persistent vector by 25, and then by 42:

```
h(25)
count =
    125  225  325  425
```

```
h(42)
count =
    167  267  367  467
```

Now do the same using `sCountFun` and `subCount`, and verify that the results are the same.

Note If you construct a new function handle to `subCount` or `nestCount`, the former value for `count` is no longer retained in memory. It is replaced by the new value.

Separate Instances of Externally Scoped Variables

The code shown below constructs two separate function handles to the same nested function, `nestCount`, that was used in the last example. It assigns the handles to fields `counter1` and `counter2` of structure `s`. These handles reference different instances of the `nestCount` function. Each handle also maintains its own separate value for the externally scoped count variable.

Call `nCountFun` twice to get two separate function handles to `nestCount`. Initialize the two instances of `count` to two different vectors:

```
s.counter1 = nCountFun([100 200 300 400]);
count =
    100  200  300  400

s.counter2 = nCountFun([-100 -200 -300 -400]);
count =
   -100  -200  -300  -400
```

Now call `nestCount` by means of each function handle to demonstrate that MATLAB increments the two count variables individually.

Increment the first counter:

```
s.counter1(25)
count =
    125    225    325    425
s.counter1(25)
count =
    150    250    350    450
```

Now increment the second counter:

```
s.counter2(25)
count =
   -75  -175  -275  -375
s.counter2(25)
count =
   -50  -150  -250  -350
```

Go back to the first counter and you can see that it keeps its own value for count:

```
s.counter1(25)
count =
    175    275    375    475
```

Restrictions on Assigning to Variables

The scoping rules for nested, and in some cases anonymous, functions require that all variables used within the function be present in the text of the code. Adding variables to the workspace of this type of function at run time is not allowed.

MATLAB issues an error if you attempt to dynamically add a variable to the workspace of an anonymous function, a nested function, or a function that contains a nested function. Examples of operations that might use dynamic assignment in this way are shown in the table below.

Type of Operation	How to Avoid Using Dynamic Assignment
Evaluating an expression using <code>eval</code> or <code>evalin</code> , or assigning a variable with <code>assignin</code>	As a general suggestion, it is best to avoid using the <code>eval</code> , <code>evalin</code> , and <code>assignin</code> functions altogether.
Loading variables from a MAT-file with the <code>load</code> function	Use the form of <code>load</code> that returns a MATLAB structure.
Assigning to a variable in a MATLAB script	Convert the script to a function, where argument- and result-passing can often clarify the code as well.
Assigning to a variable in the MATLAB debugger	<p>You can declare the variable to be <code>global</code>. For example, to create a variable <code>X</code> for temporary use in debugging, use</p> <pre data-bbox="817 756 1199 786">K>> global X; X = value</pre>

One way to avoid this error in the other cases is to pre-declare the variable in the desired function.

Examples of Nested Functions

This section shows a few examples of how you can use nested functions. These examples are intended to show you how to program with this type of function.

The examples in this section include:

- “Example 1 — Creating a Function Handle for a Nested Function” on page 15-27
- “Example 2 — Function-Generating Functions” on page 15-29

Example 1 — Creating a Function Handle for a Nested Function

The following example constructs a function handle for a nested function and then passes the handle to the MATLAB `fplot` function to plot the parabola shape. The `makeParabola` function shown here constructs and returns a

function handle fhandle for the nested parabola function. This handle gets passed to fplot:

```
function fhandle = makeParabola(a, b, c)
% MAKEPARABOLA returns a function handle with parabola
% coefficients.

fhandle = @parabola;    % @ is the function handle constructor

    function y = parabola(x)
        y = a*x.^2 + b*x + c;
    end
end
```

Assign the function handle returned from the call to a variable (h) and evaluate the function at points 0 and 25:

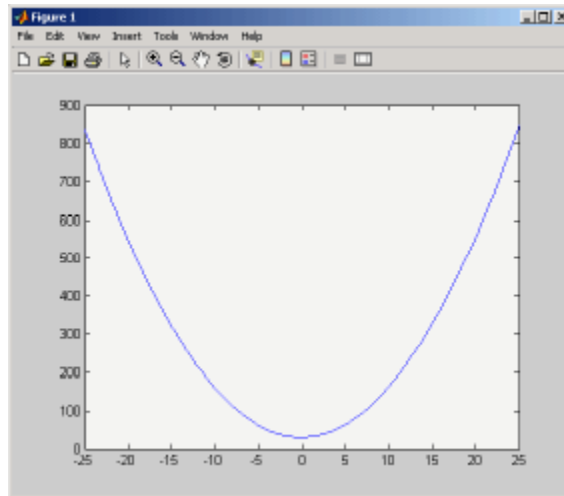
```
h = makeParabola(1.3, .2, 30)
h =
    @makeParabola/parabola

h(0)
ans =
    30

h(25)
ans =
    847.5000
```

Now pass the function handle `h` to the `fplot` function, evaluating the parabolic equation from $x = -25$ to $x = +25$:

```
fplot(h, [-25 25])
```



Example 2 – Function-Generating Functions

The fact that a function handle separately maintains a unique instance of the function from which it is constructed means that you can generate multiple handles for a function, each operating independently from the others. The function in this example makes IIR filtering functions by constructing function handles from nested functions. Each of these handles maintains its own internal state independent of the others.

The function `makeFilter` takes IIR filter coefficient vectors `a` and `b` and returns a filtering function in the form of a function handle. Each time a new input value x_n is available, you can call the filtering function to get the new output value y_n . Each filtering function created by `makeFilter` keeps its own private `a` and `b` vectors, in addition to its own private state vector, in the form of a transposed direct form II delay line:

```
function [filtfcn, statefcn] = makeFilter(b, a)
%   FILTFCN = MAKEFILTER(B, A) creates an IIR filtering
%   function and returns it in the form of a function handle,
```

```
% FILTFCN. Each time you call FILTFCN with a new filter
% input value, it computes the corresponding new filter
% output value, updating its internal state vector at the
% same time.
%
% [FILTFCN, STATEFCN] = MAKEFILTER(B, A) also returns a
% function (in the form of a function handle, STATEFCN)
% that can return the filter's internal state. The internal
% state vector is in the form of a transposed direct form
% II delay line.

% Initialize state vector. To keep this example a bit
% simpler, assume that a and b have the same length.
% Also assume that a(1) is 1.

v = zeros(size(a));

filtfcn = @iirFilter;
statefcn = @getState;

function yn = iirFilter(xn)
    % Update the state vector
    v(1) = v(2) + b(1) * xn;
    v(2:end-1) = v(3:end) + b(2:end-1) * xn - ...
        a(2:end-1) * v(1);
    v(end) = b(end) * xn - a(end) * v(1);

    % Output is the first element of the state vector.
    yn = v(1);
end

function vOut = getState
    vOut = v;
end
end
```

This sample session shows how `makeFilter` works. Make a filter that has a decaying exponential impulse response and then call it a few times in succession to see the output values change:


```
[filt1, state1] = makeFilter([1 0], [1 -.5]);

% First input to the filter is 1.
filt1(1)
ans =
    1

% Second input to the filter is 0.
filt1(0)
ans =
    0.5000

filt1(0)
ans =
    0.2500

% Show the filter's internal state.
state1()
ans =
    0.2500    0.1250

% Hit the filter with another impulse.
filt1(1)
ans =
    1.1250

% How did the state change?
state1()
ans =
    1.1250    0.5625

% Make an averaging filter.
filt2 = makeFilter([1 1 1]/3, [1 0 0]);

% Put a step input into filt2.
filt2(1)
ans =
    0.3333

filt2(1)
```

```
ans =  
    0.6667  
  
filt2(1)  
ans =  
    1  
  
% The two filter functions can be used independently.  
filt1(0)  
ans =  
    0.5625
```

As an extension of this example, suppose you were looking for a way to develop simulations of different filtering structures and compare them. This might be useful if you were interested in obtaining the range of values taken on by elements of the state vector, and how those values compare with a different filter structure. Here is one way you could capture the filter state at each step and save it for later analysis:

Call `makeFilter` with inputs `v1` and `v2` to construct function handles to the `iirFilter` and `getState` subfunctions:

```
[filtfcn, statefcn] = makeFilter(v1, v2);
```

Call the `iirFilter` and `getState` functions by means of their handles, passing in random values:

```
x = rand(1, 20);  
for k = 1:20  
    y(k) = filtfcn(x(k));  
    states{k} = statefcn(); % Save the state at each step.  
end
```

Subfunctions

In this section...

“Overview” on page 15-33

“Calling Subfunctions” on page 15-34

“Accessing Help for a Subfunction” on page 15-34

Overview

MATLAB program files can contain code for more than one function. Additional functions within the file are called *subfunctions*, and these are only visible to the primary function or to 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)           % Subfunction
% Calculate average.
a = sum(v)/n;

function m = median(v, n)        % Subfunction
% 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
```

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 length `n`.

Subfunctions cannot access variables used by other subfunctions, even within the same file, or variables used by the primary function of that file, unless you declare them as global within the pertinent functions, or pass them as arguments.

Calling Subfunctions

When you call a function from within a program 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 program file or built-in function on your search path. Because it checks for a subfunction first, you can override existing files using subfunctions with the same name.

Accessing Help for a Subfunction

You can write help for subfunctions using the same rules that apply to primary functions. To display the help for a subfunction, precede the subfunction name with the name of the file that contains the subfunction (minus file extension) and a `>` character.

For example, to get help on subfunction `mysubfun` in file `myfun.m`, type

```
help myfun>mysubfun
```

Private Functions

In this section...
“Overview” on page 15-35
“Private Folders” on page 15-35
“Accessing Help for a Private Function” on page 15-36

Overview

Private functions are functions that reside in subfolders with the special name `private`. These functions are called *private* because they are visible only to functions and scripts that meet these conditions:

- A function that calls a private function must be defined in a program file that resides in the folder immediately above that `private` subfolder.
- A script that calls a private function must itself be called from a function that has access to the private function according to the above rule.

For example, assume the folder `newmath` is on the MATLAB search path. A subfolder of `newmath` called `private` can contain functions that only the functions in `newmath` can call.

Because private functions are invisible outside the parent folder, they can use the same names as functions in other folders. This is useful if you want to create your own version of a particular function while retaining the original in another folder. Because MATLAB looks for private functions before standard functions, it finds a private function named `test.m` before a nonprivate program file named `test.m`.

Primary functions and subfunctions can also be implemented as private functions.

Private Folders

You can create your own private folders simply by creating subfolders called `private` using the standard procedures for creating folders on your computer. Do not place these private folders on your path.

Accessing Help for a Private Function

You can write help for private functions using the same rules that apply to primary functions. To display the help for a private function, precede the private function name with `private/`.

For example, to get help on private function `myprivfun`, type

```
help private/myprivfun
```

Overloaded Functions

Overloaded functions are useful when you need to create a function that responds to different types of inputs accordingly. For instance, you might want one of your functions to accept both double-precision and integer input, but to handle each type somewhat differently. You can make this difference invisible to the user by creating two separate functions having the same name, and designating one to handle `double` types and one to handle integers.

MATLAB overloaded functions reside in subfolders having a name starting with the symbol `@` and followed by the name of a recognized MATLAB class. For example, functions in the `@double` folder execute when invoked with arguments of MATLAB type `double`. Those in an `@int32` folder execute when invoked with arguments of MATLAB type `int32`.

See “Overloading MATLAB Functions” for more information on overloading functions in MATLAB.

Function Arguments

- “Find Number of Function Arguments” on page 16-2
- “Support Variable Number of Inputs” on page 16-4
- “Support Variable Number of Outputs” on page 16-6
- “Validate Number of Function Arguments” on page 16-8
- “Argument Checking in Nested Functions” on page 16-11
- “Ignore Function Inputs” on page 16-13
- “Check Function Inputs with `validateattributes`” on page 16-14
- “Parse Function Inputs” on page 16-17
- “Input Parser Validation Functions” on page 16-22

Find Number of Function Arguments

This example shows how to determine how many input or output arguments your function receives using `nargin` and `nargout`.

Input Arguments

Create a function in a file named `addme.m` that accepts up to two inputs. Identify the number of inputs with `nargin`.

```
function c = addme(a,b)

switch nargin
    case 2
        c = a + b;
    case 1
        c = a + a;
    otherwise
        c = 0;
end
```

Call `addme` with one, two, or zero input arguments.

```
addme(42)
```

```
ans =
    84
```

```
addme(2,4000)
```

```
ans =
    4002
```

```
addme
```

```
ans =
     0
```

Output Arguments

Create a new function in a file named `addme2.m` that can return one or two outputs (a result and its absolute value). Identify the number of requested outputs with `nargout`.

```
function [result,absResult] = addme2(a,b)

switch nargin
    case 2
        result = a + b;
    case 1
        result = a + a;
    otherwise
        result = 0;
end

if nargin > 1
    absResult = abs(result);
end
```

Call `addme2` with one or two output arguments.

```
value = addme2(11,-22)
```

```
value =
    -11
```

```
[value,absValue] = addme2(11,-22)
```

```
value =
    -11
```

```
absValue =
     11
```

Functions return outputs in the order they are declared in the function definition.

See Also

`nargin` | `narginchk` | `nargout` | `nargoutchk`

Support Variable Number of Inputs

This example shows how to define a function that accepts a variable number of input arguments using `varargin`. The `varargin` argument is a cell array that contains the function inputs, where each input is in its own cell.

Create a function in a file named `plotWithTitle.m` that accepts a variable number of paired (x,y) inputs for the `plot` function and an optional title. If the function receives an odd number of inputs, it assumes that the last input is a title.

```
function plotWithTitle(varargin)
if rem(nargin,2) ~= 0
    myTitle = varargin{nargin};
    numPlotInputs = nargin - 1;
else
    myTitle = 'Default Title';
    numPlotInputs = nargin;
end

plot(varargin{1:numPlotInputs})
title(myTitle)
```

Because `varargin` is a cell array, you access the contents of each cell using curly braces, `{}`. The syntax `varargin{1:numPlotInputs}` creates a comma-separated list of inputs to the `plot` function.

Call `plotWithTitle` with two sets of (x,y) inputs and a title.

```
x = [1:.1:10]; y1 = sin(x); y2 = cos(x);
plotWithTitle(x,y1,x,y2,'Sine and Cosine')
```

You can use `varargin` alone in an input argument list, or at the end of the list of inputs, such as

```
function myfunction(a,b,varargin)
```

In this case, `varargin{1}` corresponds to the third input passed to the function, and `nargin` returns `length(varargin) + 2`.

See Also

`nargin` | `varargin`

Related Examples

- “Access Data in a Cell Array” on page 8-5

More About

- “Argument Checking in Nested Functions” on page 16-11
- “Comma-Separated Lists” on page 2-100

Support Variable Number of Outputs

This example shows how to define a function that returns a variable number of output arguments using `varargout`. Output `varargout` is a cell array that contains the function outputs, where each output is in its own cell.

Create a function in a file named `magicfill.m` that assigns a magic square to each requested output.

```
function varargout = magicfill
    nOutputs = nargin;
    varargout = cell(1,nOutputs);

    for k = 1:nOutputs;
        varargout{k} = magic(k);
    end
```

Indexing with curly braces `{}` updates the contents of a cell.

Call `magicfill` and request three outputs.

```
[first,second,third] = magicfill
```

```
first =
     1

second =
     1     3
     4     2

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

MATLAB assigns values to the outputs according to their order in the `varargout` array. For example, `first == varargout{1}`.

You can use `varargout` alone in an output argument list, or at the end of the list of outputs, such as

```
function [x,y,varargout] = myfunction(a,b)
```

In this case, `varargout{1}` corresponds to the third output that the function returns, and `nargout` returns `length(varargout) + 2`.

See Also

`nargout` | `varargout`

Related Examples

- “Access Data in a Cell Array” on page 8-5

More About

- “Argument Checking in Nested Functions” on page 16-11

Validate Number of Function Arguments

This example shows how to check whether your custom function receives a valid number of input or output arguments. MATLAB performs some argument checks automatically. For other cases, you can use `narginchk` or `nargoutchk`.

Automatic Argument Checks

MATLAB checks whether your function receives more arguments than expected when it can determine the number from the function definition. For example, this function accepts up to two outputs and three inputs:

```
function [x,y] = myFunction(a,b,c)
```

If you pass too many inputs to `myFunction`, MATLAB issues an error. You do not need to call `narginchk` to check for this case.

```
[X,Y] = myFunction(1,2,3,4)
```

```
Error using myFunction
Too many input arguments.
```

Use the `narginchk` and `nargoutchk` functions to verify that your function receives:

- A minimum number of required arguments.
- No more than a maximum number of arguments, when your function uses `varargin` or `varargout`.

Input Checks with `narginchk`

Define a function in a file named `testValues.m` that requires at least two inputs. The first input is a threshold value to compare against the other inputs.

```
function testValues(threshold,varargin)
minInputs = 2;
maxInputs = Inf;
narginchk(minInputs,maxInputs)
```



```

for k = 1:(nargin-1)
    if (varargin{k} > threshold)
        fprintf('Test value %d exceeds %d\n',k,threshold);
    end
end

```

Call `testValues` with too few inputs.

```
testValues(10)
```

```

Error using testValues (line 4)
Not enough input arguments.

```

Call `testValues` with enough inputs.

```
testValues(10,1,11,111)
```

```

Test value 2 exceeds 10
Test value 3 exceeds 10

```

Output Checks with `nargoutchk`

Define a function in a file named `mysize.m` that returns the dimensions of the input array in a vector (from the `size` function), and optionally returns scalar values corresponding to the sizes of each dimension. Use `nargoutchk` to verify that the number of requested individual sizes does not exceed the number of available dimensions.

```

function [sizeVector,varargout] = mysize(x)
minOutputs = 0;
maxOutputs = ndims(x) + 1;
nargoutchk(minOutputs,maxOutputs)

sizeVector = size(x);

varargout = cell(1,nargout-1);
for k = 1:length(varargout)
    varargout{k} = sizeVector(k);
end

```

Call `mysize` with a valid number of outputs.

```
A = rand(3,4,2);  
[fullsize,nrows,ncols,npages] = mysize(A)
```

```
fullsize =  
    3     4     2
```

```
nrows =  
    3
```

```
ncols =  
    4
```

```
npages =  
    2
```

Call `mysize` with too many outputs.

```
A = 1;  
[fullsize,nrows,ncols,npages] = mysize(A)
```

```
Error using mysize (line 4)  
Too many output arguments.
```

See Also

[narginchk](#) | [nargoutchk](#)

Related Examples

- “Support Variable Number of Inputs” on page 16-4
- “Support Variable Number of Outputs” on page 16-6

Argument Checking in Nested Functions

This topic explains special considerations for using `varargin`, `varargout`, `nargin`, and `nargout` with nested functions.

`varargin` and `varargout` allow you to create functions that accept variable numbers of input or output arguments. Although `varargin` and `varargout` look like function names, they refer to variables, not functions. This is significant because nested functions share the workspaces of the functions that contain them.

If you do not use `varargin` or `varargout` in the declaration of a nested function, then `varargin` or `varargout` within the nested function refers to the arguments of an outer function.

For example, create a function in a file named `showArgs.m` that uses `varargin` and has two nested functions, one that uses `varargin` and one that does not.

```
function showArgs(varargin)
    nested1(3,4)
    nested2(5,6,7)

    function nested1(a,b)
        disp('nested1: Contents of varargin{1}')
        disp(varargin{1})
    end

    function nested2(varargin)
        disp('nested2: Contents of varargin{1}')
        disp(varargin{1})
    end

end
```

Call the function and compare the contents of `varargin{1}` in the two nested functions.

```
showArgs(0,1,2)

nested1: Contents of varargin{1}
```

```
0
```

```
nested2: Contents of varargin{1}
5
```

On the other hand, `nargin` and `nargout` are functions. Within any function, including nested functions, calls to `nargin` or `nargout` return the number of arguments for that function. If a nested function requires the value of `nargin` or `nargout` from an outer function, pass the value to the nested function.

For example, create a function in a file named `showNumArgs.m` that passes the number of input arguments from the primary (parent) function to a nested function.

```
function showNumArgs(varargin)

disp(['Number of inputs to showNumArgs: ',int2str(nargin)]);
nestedFx(nargin,2,3,4)

    function nestedFx(n,varargin)
        disp(['Number of inputs to nestedFx: ',int2str(nargin)]);
        disp(['Number of inputs to its parent: ',int2str(n)]);
    end

end
```

Call `showNumArgs` and compare the output of `nargin` in the parent and nested functions.

```
showNumArgs(0,1)
```

```
Number of inputs to showNumArgs: 2
Number of inputs to nestedFx: 4
Number of inputs to its parent: 2
```

See Also

`nargin` | `nargout` | `varargin` | `varargout`

Ignore Function Inputs

This example shows how to ignore inputs in your function definition using the tilde (~) operator.

Use this operator when your function must accept a predefined set of inputs, but your function does not use all of the inputs. Common applications include defining callback functions, as shown here, or deriving a class from a superclass.

Define a callback for a push button in a file named `colorButton.m` that does not use the `eventdata` input. Ignore the input with a tilde.

```
function colorButton
figure;
uicontrol('Style','pushbutton','String','Click me','Callback',@btnCallback)

function btnCallback(h,~)
set(h,'BackgroundColor',rand(3,1))
```

The function declaration for `btnCallback` is essentially the same as

```
function btnCallback(h,eventdata)
```

However, using the tilde prevents the addition of `eventdata` to the function workspace and makes it clearer that the function does not use `eventdata`.

You can ignore any number of function inputs, in any position in the argument list. Separate consecutive tildes with a comma, such as

```
myfunction(myinput,~,~)
```

Check Function Inputs with `validateattributes`

This example shows how to verify that the inputs to your function conform to a set of requirements using the `validateattributes` function.

`validateattributes` requires that you pass the variable to check and the supported data types for that variable. Optionally, pass a set of attributes that describe the valid dimensions or values.

Check Data Type and Other Attributes

Define a function in a file named `checkme.m` that accepts up to three inputs: `a`, `b`, and `c`. Check whether:

- `a` is a two-dimensional array of positive double-precision values.
- `b` contains 100 numeric values in an array with 10 columns.
- `c` is a nonempty character string or cell array.

```
function checkme(a,b,c)

validateattributes(a,{'double'},{'positive','2d'})
validateattributes(b,{'numeric'},{'numel',100,'ncols',10})
validateattributes(c,{'char','cell'},{'nonempty'})

disp('All inputs are ok.')
```

The curly braces `{}` indicate that the set of data types and the set of additional attributes are in cell arrays. Cell arrays allow you to store combinations of text and numeric data, or text strings of different lengths, in a single variable.

Call `checkme` with valid inputs.

```
checkme(pi,rand(5,10,2),'text')
```

```
All inputs are ok.
```

The scalar value `pi` is two-dimensional because `size(pi) = [1,1]`.

Call `checkme` with invalid inputs. The `validateattributes` function issues an error for the first input that fails validation, and `checkme` stops processing.

```
checkme(-4)
```

```
Error using checkme (line 3)  
Expected input to be positive.
```

```
checkme(pi,rand(3,4,2))
```

```
Error using checkme (line 4)  
Expected input to be an array with number of elements equal to 100.
```

```
checkme(pi,rand(5,10,2),struct)
```

```
Error using checkme (line 5)  
Expected input to be one of these types:
```

```
char, cell
```

```
Instead its type was struct.
```

The default error messages use the generic term `input` to refer to the argument that failed validation. When you use the default error message, the only way to determine which input failed is to view the specified line of code in `checkme`.

Add Input Name and Position to Errors

Define a function in a file named `checkdetails.m` that performs the same validation as `checkme`, but adds details about the input name and position to the error messages.

```
function checkdetails(a,b,c)  
  
validateattributes(a,{'double'},{'positive','2d'},'', 'First',1)  
validateattributes(b,{'numeric'},{'numel',100,'ncols',10},'', 'Second',2)  
validateattributes(c,{'char'},{'nonempty'},'', 'Third',3)  
  
disp('All inputs are ok.')
```

The empty string '' for the fourth input to `validateattributes` is a placeholder for an optional function name string. You do not need to specify a function name because it already appears in the error message. Specify the function name when you want to include it in the error identifier for additional error handling.

Call `checkdetails` with invalid inputs.

```
checkdetails(-4)
```

```
Error using checkdetails (line 3)  
Expected input number 1, First, to be positive.
```

```
checkdetails(pi,rand(3,4,2))
```

```
Error using checkdetails (line 4)  
Expected input number 2, Second, to be an array with  
number of elements equal to 100.
```

See Also

`validateattributes` | `validatestring`

Parse Function Inputs

This example shows how to define required and optional inputs, assign defaults to optional inputs, and validate all inputs to a custom function using the Input Parser.

The Input Parser provides a consistent way to validate and assign defaults to inputs, improving the robustness and maintainability of your code. To validate the inputs, you can take advantage of existing MATLAB functions or write your own validation routines.

Step 1. Define your function.

Create a function in a file named `printPhoto.m`. The `printPhoto` function has one required input for the file name, and optional inputs for the finish (glossy or matte), color space (RGB or CMYK), width, and height.

```
function printPhoto(filename,varargin)
```

In your function declaration statement, specify required inputs first. Use `varargin` to support optional inputs.

Step 2. Create an InputParser object.

Within your function, call `inputParser` to create a parser object.

```
p = inputParser;
```

Step 3. Add inputs to the scheme.

Add inputs to the parsing scheme in your function using `addRequired`, `addOptional`, or `addParamValue`. For optional inputs, specify default values.

For each input, you can specify a handle to a validation function that checks the input and returns a scalar logical (`true` or `false`) or errors. The validation function can be an existing MATLAB function (such as `ischar` or `isnumeric`) or a function that you create (such as an anonymous function or a subfunction).

In the `printPhoto` function, `filename` is a required input. Define `finish` and `color` as optional input strings, and `width` and `height` as optional parameter value pairs.

```
defaultFinish = 'glossy';
validFinishes = {'glossy','matte'};
checkFinish = @(x) any(validatestring(x,validFinishes));

defaultColor = 'RGB';
validColors = {'RGB','CMYK'};
checkColor = @(x) any(validatestring(x,validColors));

defaultWidth = 6;
defaultHeight = 4;

addRequired(p,'filename',@ischar);
addOptional(p,'finish',defaultFinish,checkFinish);
addOptional(p,'color',defaultColor,checkColor);
addParamValue(p,'width',defaultWidth,@isnumeric);
addParamValue(p,'height',defaultHeight,@isnumeric);
```

Inputs that you add with `addRequired` or `addOptional` are *positional* arguments. When you call a function with positional inputs, specify those values in the order they are added to the parsing scheme.

Inputs added with `addParamValue` are not positional, so you can pass values for `height` before or after values for `width`. However, parameter value inputs require that you pass the input name ('`height`' or '`width`') along with the value of the input.

If your function accepts optional input strings and parameter name and value pairs, specify validation functions for the optional input strings. Otherwise, the Input Parser interprets the optional strings as parameter names. For example, the `checkFinish` validation function ensures that `printPhoto` interprets '`glossy`' as a value for `finish` and not as an invalid parameter name.

Step 4. Set properties to adjust parsing (optional).

By default, the Input Parser makes assumptions about case sensitivity, function names, structure array inputs, and whether to allow additional parameter names and values that are not in the scheme. Properties allow you to explicitly define the behavior. Set properties using dot notation, similar to assigning values to a structure array.

Allow `printPhoto` to accept additional parameter value inputs that do not match the input scheme by setting the `KeepUnmatched` property of the Input Parser.

```
p.KeepUnmatched = true;
```

If `KeepUnmatched` is `false` (default), the Input Parser issues an error when inputs do not match the scheme.

Step 5. Parse the inputs.

Within your function, call the `parse` method. Pass the values of all of the function inputs.

```
parse(p,filename,varargin{:});
```

Step 6. Use the inputs in your function.

Access parsed inputs using these properties of the `inputParser` object:

- `Results` — Structure array with names and values of all inputs in the scheme.
- `Unmatched` — Structure array with parameter names and values that are passed to the function, but are not in the scheme (when `KeepUnmatched` is `true`).
- `UsingDefaults` — Cell array with names of optional inputs that are assigned their default values because they are not passed to the function.

Within the `printPhoto` function, display the values for some of the inputs:

```
disp(['File name: ',p.Results.filename])
disp(['Finish: ', p.Results.finish])

if ~isempty(fieldnames(p.Unmatched))
    disp('Extra inputs:')
    disp(p.Unmatched)
end
if ~isempty(p.UsingDefaults)
    disp('Using defaults: ')
    disp(p.UsingDefaults)
```

end

Step 7. Call your function.

The Input Parser expects to receive inputs as follows:

- Required inputs first, in the order they are added to the parsing scheme with `addRequired`.
- Optional positional inputs in the order they are added to the scheme with `addOptional`.
- Positional inputs before parameter name and value pair inputs.
- Parameter names and values in the form `Name1,Value1,...,NameN,ValueN`.

Pass several combinations of inputs to `printPhoto`, some valid and some invalid:

```
printPhoto('myfile.jpg')
```

```
File name: myfile.jpg
Finish: glossy
Using defaults:
    'finish'    'color'    'width'    'height'
```

```
printPhoto(100)
```

```
Error using printPhoto (line 23)
Argument 'filename' failed validation ischar.
```

```
printPhoto('myfile.jpg','satin')
```

```
Error using printPhoto (line 23)
Argument 'finish' failed validation with error:
Expected input to match one of these strings:
```

```
glossy, matte
```

The input, 'satin', did not match any of the valid strings.

```
printPhoto('myfile.jpg','height',10,'width',8)
```

```
File name: myfile.jpg
Finish: glossy
Using defaults:
    'finish'    'color'
```

To pass a value for the n th positional input, either specify values for the previous ($n - 1$) inputs or pass the input as a parameter name and value pair. For example, these function calls assign the same values to `finish` (default 'glossy') and `color`:

```
printPhoto('myfile.gif', 'glossy', 'CMYK'); % finish and color: positional
printPhoto('myfile.gif', 'color', 'CMYK');  % color: name and value
```

See Also

`inputParser` | `varargin`

More About

- “Input Parser Validation Functions” on page 16-22

Input Parser Validation Functions

This topic shows ways to define validation functions that you pass to the Input Parser to check custom function inputs.

The Input Parser methods `addRequired`, `addOptional`, and `addParamValue` each accept an optional handle to a validation function. Designate function handles with an at (`@`) symbol.

Validation functions must accept a single input argument, and they must either return a scalar logical value (`true` or `false`) or error. If the validation function returns `false`, the Input Parser issues an error and your function stops processing.

There are several ways to define validation functions:

- Use an existing MATLAB function such as `ischar` or `isnumeric`. For example, check that a required input named `num` is numeric:

```
p = inputParser;  
checknum = @isnumeric;  
addRequired(p, 'num', checknum);  
  
parse(p, 'text');
```

```
Argument 'num' failed validation isnumeric.
```

- Create an anonymous function. For example, check that input `num` is a numeric scalar greater than zero:

```
p = inputParser;  
checknum = @(x) isnumeric(x) && isscalar(x) && (x > 0);  
addRequired(p, 'num', checknum);  
  
parse(p, rand(3));
```

```
Argument 'num' failed validation @(x)isnumeric(x)&&isscalar(x)&&(x>0).
```

- Define your own function, typically a subfunction in the same file as your primary function. For example, in a file named `usenum.m`, define a

subfunction named `checknum` that issues custom error messages when the input `num` to `usenum` is not a numeric scalar greater than zero:

```
function usenum(num)
    p = inputParser;
    addRequired(p, 'num', @checknum);
    parse(p, num);

function TF = checknum(x)
    TF = false;
    if ~isscalar(x)
        error('Input is not scalar');
    elseif ~isnumeric(x)
        error('Input is not numeric');
    elseif (x <= 0)
        error('Input must be > 0');
    else
        TF = true;
    end
```

Call the function with an invalid input:

```
usenum(-1)
```

```
Error using usenum (line 4)
Argument 'num' failed validation with error:
Input must be > 0
```

See Also

[inputParser](#) | [is*](#) | [validateattributes](#)

Related Examples

- “Parse Function Inputs” on page 16-17

More About

- Chapter 9, “Function Handles”
- “Anonymous Functions” on page 15-3

Programming Tips

- “Introduction” on page 17-2
- “Command and Function Syntax” on page 17-3
- “Help” on page 17-6
- “Development Environment” on page 17-10
- “Functions” on page 17-12
- “Function Arguments” on page 17-15
- “Program Development” on page 17-18
- “Debugging” on page 17-21
- “Variables” on page 17-25
- “Strings” on page 17-29
- “Evaluating Expressions” on page 17-32
- “MATLAB Path” on page 17-34
- “Program Control” on page 17-38
- “Save and Load” on page 17-42
- “Files and Filenames” on page 17-45
- “Input/Output” on page 17-48
- “Starting MATLAB” on page 17-50
- “Operating System Compatibility” on page 17-51
- “For More Information” on page 17-53

Introduction

This section is a categorized compilation of tips for the MATLAB programmer. Each item is relatively brief to help you browse through them and find information that is useful. Many of the tips include a reference to specific MATLAB documentation that gives you more complete coverage of the topic.

For suggestions on how to improve the performance of your MATLAB programs, and how to write programs that use memory more efficiently, see [Improving Performance and Memory Usage](#).

Command and Function Syntax

In this section...

“Syntax Help” on page 17-3
 “Command and Function Syntaxes” on page 17-3
 “Command Line Continuation” on page 17-3
 “Completing Commands Using the Tab Key” on page 17-4
 “Recalling Commands” on page 17-4
 “Clearing Commands” on page 17-5
 “Suppressing Output to the Screen” on page 17-5

Syntax Help

For help about the general syntax of MATLAB functions and commands, type

```
help syntax
```

Command and Function Syntaxes

You can enter MATLAB commands using either a *command* or *function* syntax. It is important to learn the restrictions and interpretation rules for both.

```
functionname arg1 arg2 arg3           % Command syntax
functionname('arg1','arg2','arg3')    % Function syntax
```

For more information: See “Command vs. Function Syntax” on page 1-13.

Command Line Continuation

You can continue most statements to one or more additional lines by terminating each incomplete line with an ellipsis (...). Breaking down a statement into a number of lines can sometimes result in a clearer programming style.

```
fprintf ('Example %d shows a command coded on %d lines.\n', ...
        exampleNumber, ...
        numberOfLines)
```

Note that you cannot continue an incomplete string to another line.

```
disp 'This statement attempts to continue a string ...  
    to another line, resulting in an error.'
```

For more information: See “Continue Long Statements on Multiple Lines” on page 1-7.

Completing Commands Using the Tab Key

You can save some typing when entering commands by entering only the first few letters of the command, variable, property, etc. followed by the **Tab** key. Typing the second line below (with **T** representing **Tab**) yields the expanded, full command shown in the third line:

```
f = figure;  
set(f, 'papTuT','cT')                % Type this line.  
set(f, 'paperunits','centimeters')  % This is what you get.
```

If there are too many matches for the string you are trying to complete, you will get no response from the first **Tab**. Press **Tab** again to see all possible choices:

```
set(f, 'paTT  
PaperOrientation  PaperPositionMode  PaperType      Parent  
PaperPosition    PaperSize          PaperUnits
```

For more information: See Tab Completion in the Command Window.

Recalling Commands

Use any of the following methods to simplify recalling previous commands to the screen:

- To recall an earlier command to the screen, press the up arrow key one or more times, until you see the command you want. If you want to modify the recalled command, you can edit its text before pressing **Enter** or **Return** to execute it.

- To recall a specific command by name without having to scroll through your earlier commands one by one, type the starting letters of the command, followed by the up arrow key.
- Open the Command History window (**Desktop > Command History**) to see all previous commands. Double-click the command you want to execute.

For more information: See Command History Window.

Clearing Commands

If you have typed a command that you then decide not to execute, you can clear it from the Command Window by pressing the Escape (**Esc**) key.

Suppressing Output to the Screen

To suppress output to the screen, end statements with a semicolon. This can be particularly useful when generating large matrices.

```
A = magic(100);    % Create matrix A, but do not display it.
```

Help

In this section...

- “Using the Help Browser” on page 17-6
- “Help on Functions from the Help Browser” on page 17-6
- “Help on Functions from the Command Window” on page 17-7
- “Topical Help” on page 17-7
- “Paged Output” on page 17-8
- “Writing Your Own Help” on page 17-8
- “Help for Subfunctions and Private Functions” on page 17-8
- “Help for Methods and Overloaded Functions” on page 17-9

Using the Help Browser



Open the Help browser from the MATLAB Command Window using one of the following:

- Click the question mark symbol in the toolbar.
- Select **Help > Product Help** from the menu.
- Type the word `doc` at the command prompt.

For more information: See “Ways to Get Function Help”.

Help on Functions from the Help Browser

You can find help on a MATLAB function in any of the following ways:

- Click the  **Functions** button in the left pane of the Help browser. This brings you to that part of the Function Reference documentation that is organized by category. To use an alphabetical list to get help on a specific function, click Alphabetical List at the top of that page.
- Click the  **MATLAB** button in the left pane of the Help browser. Look in the upper left corner of the page for links to either Functions: By

Category, or Functions: Alphabetical List and click there for the type of documentation access you prefer.

- Type `doc functionname` at the command line.

Help on Functions from the Command Window

Several types of help on functions are available from the Command Window:

- To list all categories that you can request help on from the Command Window, just type

```
help
```

- To see a list of functions for one of these categories, along with a brief description of each function, type `help category`. For example,

```
help datafun
```

- To get help on a particular function, type `help functionname`. For example,

```
help sortrows
```

Topical Help

In addition to the help on individual functions, you can get help on any of the following topics by typing `help topicname` at the command line.

Topic Name	Description
<code>arith</code>	Arithmetic operators
<code>relop</code>	Relational and logical operators
<code>punct</code>	Special character operators
<code>slash</code>	Arithmetic division operators
<code>paren</code>	Parentheses, braces, and bracket operators
<code>precedence</code>	Operator precedence
<code>datatypes</code>	MATLAB classes, their associated functions, and operators that you can overload
<code>lists</code>	Comma separated lists

Topic Name	Description
strings	Character strings
function_handle	Function handles and the @ operator
debug	Debugging functions
java	Using Sun Java from within the MATLAB software.
changeNotification	Microsoft Windows change notification

Paged Output

Before displaying a lengthy section of help text or code, put MATLAB into its paged output mode by typing `more on`. This breaks up any ensuing display into pages for easier viewing. Turn off paged output with `more off`.

Page through the displayed text using the space bar key. Or step through line by line using **Enter** or **Return**. Discontinue the display by pressing the **Q** key or **Ctrl+C**.

Writing Your Own Help

Start each program you write with a section of text providing help on how and when to use the function. If formatted properly, the MATLAB `help` function displays this text when you enter

```
help functionname
```

MATLAB considers the first group of consecutive lines immediately following the function definition line that begin with `%` to be the help section for the function. The first line without `%` as the left-most character ends the help.

For more information: See Help Text.

Help for Subfunctions and Private Functions

You can write help for subfunctions using the same rules that apply to main functions. To display the help for the subfunction `mysubfun` in file `myfun.m`, type


```
help myfun>mysubfun
```

To display the help for a private function, precede the function name with `private/`. To get help on private function `myprivfun`, type

```
help private/myprivfun
```

Help for Methods and Overloaded Functions

You can write help text for object-oriented class methods implemented as MATLAB functions. Display help for the method by typing

```
help classname/methodname
```

where the file `methodname.m` resides in subfolder `@classname`.

For example, if you write a `plot` method for a class named `polynom`, (where the `plot` method is defined in the file `@polynom/plot.m`), you can display this help by typing

```
help polynom/plot
```

You can get help on overloaded MATLAB functions in the same way. To display the help text for the `eq` function as implemented in `matlab/iofun/@serial`, type

```
help serial/eq
```

Development Environment

In this section...
“Workspace Browser” on page 17-10
“Using the Find Utility” on page 17-10
“Commenting Out a Block of Code” on page 17-11
“Creating Functions from Command History” on page 17-11
“Editing Functions in EMACS” on page 17-11

Workspace Browser

The Workspace browser is a graphical interface to the variables stored in the MATLAB base and function workspaces. You can view, modify, save, load, and create graphics from workspace data using the browser. Select **Desktop > Workspace** to open the browser.

To view function workspaces, you need to be in debug mode.

For more information: See MATLAB Workspace.

Using the Find Utility

Find any word or phrase in a group of files using the Find utility. Click

Desktop > Current Folder, click the  icon at the top of the **Current Folder** window, and then select **Find Files** from the menu that appears.

When entering search text, you do not need to put quotes around a phrase. In fact, parts of words, like win for windows, will not be found if enclosed in quotes.

For more information: See Finding and Replacing Text in the Current File.

Commenting Out a Block of Code

To comment out a block of text or code within the MATLAB editor,

- 1 Highlight the block of text you would like to comment out.
- 2 Holding the mouse over the highlighted text, select **Text > Comment** (or **Uncomment**, to do the reverse) from the toolbar. (You can also get these options by right-clicking the mouse.)

For more information: See Adding Comments.

Creating Functions from Command History

If there is part of your current MATLAB session that you would like to add to a function, this is easily done using the Command History window:

- 1 Open this window by selecting **Desktop > Command History**.
- 2 Use **Shift+Click** or **Ctrl+Click** to select the lines you want to use. MATLAB highlights the selected lines.
- 3 Right-click once, and select **Create Script** from the menu that appears. MATLAB creates a new Editor window displaying the selected code.

Editing Functions in EMACS

If you use Emacs, you can download editing modes for editing MATLAB functions with GNU-Emacs or with early versions of Emacs from the MATLAB Central Web site:

<http://www.mathworks.com/matlabcentral/>

At this Web site, select **File Exchange**, and then **Utilities > Emacs**.

For more information: See General Preferences for the Editor/Debugger.

Functions

In this section...

“Function Structure” on page 17-12
 “Using Lowercase for Function Names” on page 17-12
 “Getting a Function’s Name and Path” on page 17-13
 “What Files Does a Function Use?” on page 17-13
 “Dependent Functions, Built-Ins, Classes” on page 17-14

Function Structure

An MATLAB function consists of the components shown here:

```
function [x, y] = myfun(a, b, c)    % Function definition line
% H1 line -- A one-line summary of the function's purpose.
% Help text -- One or more lines of help text that explain
%   how to use the function. This text is displayed when
%   the user types "help functionname".

% The Function body normally starts after the first blank line.
% Comments -- Description (for internal use) of what the
%   function does, what inputs are expected, what outputs
%   are generated. Typing "help functionname" does not display
%   this text.

x = prod(a, b);                    % Start of Function code
```

For more information: See Basic Parts of a Function.

Using Lowercase for Function Names

Function names appear in uppercase in MATLAB help text only to make the help easier to read. In practice, however, it is usually best to use lowercase when calling functions.

Case requirements depend on the case sensitivity of the operating system you are using. As a rule, naming and calling functions using lowercase generally makes them more portable from one operating system to another.

Getting a Function's Name and Path

To obtain the file name for the function currently being executed, use the following function in your code.

```
mfilename
```

To include the path along with the file name, use:

```
x = mfilename('fullpath')
```

For more information: See the `mfilename` function reference page.

What Files Does a Function Use?

For a simple display of all functions referenced by a particular function, follow the steps below:

- 1 Type `clear functions` to clear all functions from memory (see Note below).
- 2 Execute the function you want to check. Note that the function arguments you choose to use in this step are important, since you can get different results when calling the same function with different arguments.
- 3 Type `inmem` to display all MATLAB function files that were used when the function ran. If you want to see what MEX-files were used as well, specify an additional output, as shown here:

```
[mfiles, mexfiles] = inmem
```

Note `clear functions` does not clear functions locked by `mlock`. If you have locked functions, (which you can check using `inmem`), unlock them with `munlock`, and then repeat step 1.

Dependent Functions, Built-Ins, Classes

For a much more detailed display of dependent function information, use the `depfun` function. In addition to MATLAB function files, `depfun` shows which built-ins and classes a particular function depends on.

Function Arguments

In this section...

“Getting the Input and Output Arguments” on page 17-15

“Variable Numbers of Arguments” on page 17-15

“String or Numeric Arguments” on page 17-16

“Passing Arguments in a Structure” on page 17-16

“Passing Arguments in a Cell Array” on page 17-16

Getting the Input and Output Arguments

Use `nargin` and `nargout` to determine the number of input and output arguments in a particular function call. Use `narginchk` and `nargoutchk` to verify that your function is called with the required number of input and output arguments.

```
function [x, y] = myplot(a, b, c, d)
narginchk(2, 4)           % Allow 2 to 4 inputs
nargoutchk(0, 2)         % Allow 0 to 2 outputs

x = plot(a, b);
if nargin == 4
    y = myfun(c, d);
end
```

Variable Numbers of Arguments

You can call functions with fewer input and output arguments than you have specified in the function definition, but not more. If you want to call a function with a variable number of arguments, use the `varargin` and `varargout` function parameters in the function definition.

This function returns the size vector and, optionally, individual dimensions:

```
function [s, varargout] = mysize(x)
nout = max(nargout, 1) - 1;
s = size(x);
for k = 1:nout
```

```
    varargout(k) = {s(k)};
end
```

Try calling it with

```
[s, rows, cols] = mysize(rand(4, 5))
```

String or Numeric Arguments

If you are passing only string arguments into a function, you can use MATLAB command syntax. All arguments entered in command syntax are interpreted as strings.

```
strcmp string1 string1
ans =
    1
```

When passing numeric arguments, it is best to use function syntax unless you want the number passed as a string. The right-hand example below passes the number 75 as the string, '75'.

```
isnumeric(75)                isnumeric '75'
ans =                          ans =
    1                          0
```

For more information: See Command vs. Function Syntax.

Passing Arguments in a Structure

Instead of requiring an additional argument for every value you want to pass in a function call, you can package them in a MATLAB structure and pass the structure. Make each input you want to pass a separate field in the structure argument, using descriptive names for the fields.

Structures allow you to change the number, contents, or order of the arguments without having to modify the function. They can also be useful when you have a number of functions that need similar information.

Passing Arguments in a Cell Array

You can also group arguments into cell arrays. The disadvantage over structures is that you do not have field names to describe each variable. The

advantage is that cell arrays are referenced by index, allowing you to loop through a cell array and access each argument passed in or out of the function.

Program Development

In this section...
“Planning the Program” on page 17-18
“Using Pseudo-Code” on page 17-18
“Selecting the Right Data Structures” on page 17-18
“General Coding Practices” on page 17-19
“Naming a Function Uniquely” on page 17-19
“The Importance of Comments” on page 17-19
“Coding in Steps” on page 17-20
“Making Modifications in Steps” on page 17-20
“Functions with One Calling Function” on page 17-20
“Testing the Final Program” on page 17-20

Planning the Program

When planning how to write a program, take the problem you are trying to solve and break it down into a series of smaller, independent tasks. Implement each task as a separate function. Try to keep functions fairly short, each having a single purpose.

Using Pseudo-Code

You may find it helpful to write the initial draft of your program in a structured format using your own natural language. This *pseudo-code* is often easier to think through, review, and modify than using a formal programming language, yet it is easily translated into a programming language in the next stage of development.

Selecting the Right Data Structures

Look at what classes and data structures are available to you in MATLAB and determine which of those best fit your needs in storing and passing your data.

General Coding Practices

A few suggested programming practices:

- Use descriptive function and variable names to make your code easier to understand.
- Order subfunctions alphabetically in a file to make them easier to find.
- Precede each subfunction with a block of help text describing what that subfunction does. This not only explains the subfunctions, but also helps to visually separate them.
- Do not extend lines of code beyond the 80th column. Otherwise, it will be hard to read when you print it out.
- Use full Handle Graphics property and value names. Abbreviated names are often allowed, but can make your code unreadable. They also could be incompatible in future releases of MATLAB.

Naming a Function Uniquely

To avoid choosing a name for a new function that might conflict with a name already in use, check for any occurrences of the name using this command:

```
which -all functionname
```

For more information: See the `which` function reference page.

The Importance of Comments

Be sure to document your programs well to make it easier for you or someone else to maintain them. Add comments generously, explaining each major section and any smaller segments of code that are not obvious. You can add a block of comments as shown here.

```
%-----  
% This function computes the ... <and so on>  
%-----
```

For more information: See `Comments` .

Coding in Steps

Do not try to write the entire program all at once. Write a portion of it, and then test that piece out. When you have that part working the way you want, then write the next piece, and so on. It is much easier to find programming errors in a small piece of code than in a large program.

Making Modifications in Steps

When making modifications to a working program, do not make widespread changes all at one time. It is better to make a few small changes, test and debug, make a few more changes, and so on. Tracking down a difficult bug in the small section that you have changed is much easier than trying to find it in a huge block of new code.

Functions with One Calling Function

If you have a function that is called by only one other function, put it in the same file as the calling function, making it a subfunction.

For more information: See “String Comparisons” on page 6-25.

Testing the Final Program

One suggested practice for testing a new program is to step through the program in the MATLAB debugger while keeping a record of each line that gets executed on a printed copy of the program. Use different combinations of inputs until you have observed that every line of code is executed at least once.

Debugging

In this section...

“The MATLAB Debug Functions” on page 17-21
 “More Debug Functions” on page 17-21
 “The MATLAB Graphical Debugger” on page 17-22
 “A Quick Way to Examine Variables” on page 17-22
 “Setting Breakpoints from the Command Line” on page 17-22
 “Finding Line Numbers to Set Breakpoints” on page 17-23
 “Stopping Execution on an Error or Warning” on page 17-23
 “Locating an Error from the Error Message” on page 17-23
 “Using Warnings to Help Debug” on page 17-23
 “Making Code Execution Visible” on page 17-24
 “Debugging Scripts” on page 17-24

The MATLAB Debug Functions

For a brief description of the main debug functions in MATLAB, type

```
help debug
```

For more information: See Debugging Process and Features.

More Debug Functions

Other functions you may find useful in debugging are listed below.

Function	Description
echo	Display function or script code as it executes.
disp	Display specified values or messages.
sprintf, fprintf	Display formatted data of different types.

Function	Description
whos	List variables in the workspace.
size	Show array dimensions.
keyboard	Interrupt program execution and allow input from keyboard.
return	Resume execution following a keyboard interruption.
warning	Display specified warning message.
MException	Access information on the cause of an error.
lastwarn	Return warning message that was last issued.

The MATLAB Graphical Debugger

Learn to use the MATLAB graphical debugger. You can view the function and its calling functions as you debug, set and clear breakpoints, single-step through the program, step into or over called functions, control visibility into all workspaces, and find and replace strings in your files.

Start out by opening the file you want to debug using **File > Open** or the open function. Use the debugging functions available on the toolbar and pull-down menus to set breakpoints, run or step through the program, and examine variables.

For more information: See Debugging Process and Features.

A Quick Way to Examine Variables

To see the value of a variable from the Editor/Debugger window, hold the mouse cursor over the variable name for a second or two. You will see the value of the selected variable displayed.

Setting Breakpoints from the Command Line

You can set breakpoints with `dbstop` in any of the following ways:

- Break at a specific file line number.

- Break at the beginning of a specific subfunction.
- Break at the first executable line in a file.
- Break when a warning, or error, is generated.
- Break if any infinite or NaN values are encountered.

For more information: See Setting Breakpoints.

Finding Line Numbers to Set Breakpoints

When debugging from the command line, a quick way to find line numbers for setting breakpoints is to use `dbtype`. The `dbtype` function displays all or part of the file, also numbering each line. To display `delaunay.m`, use

```
dbtype delaunay
```

To display only lines 35 through 41, use

```
dbtype delaunay 35:41
```

Stopping Execution on an Error or Warning

Use `dbstop if error` to stop program execution on any error and enter debug mode. Use `dbstop if warning` to stop execution on any warning and enter debug mode.

For more information: See “Debugging Process and Features”.

Locating an Error from the Error Message

Click on the underlined text in an error message, and MATLAB opens the file being executed in its editor and places the cursor at the point of error.

Using Warnings to Help Debug

You can detect erroneous or unexpected behavior in your programs by inserting warning messages that MATLAB will display under the conditions you specify. See the section on Warning Control in the MATLAB Programming Fundamentals documentation to find out how to selectively enable warnings.

For more information: See the warning function reference page.

Making Code Execution Visible

An easy way to see the end result of a particular line of code is to edit the program and temporarily remove the terminating semicolon from that line. Then, run your program and the evaluation of that statement is displayed on the screen.

Debugging Scripts

Scripts store their variables in a workspace that is shared with the caller of the script. So, when you debug a script from the command line, the script uses variables from the base workspace. To avoid errors caused by workspace sharing, type `clear all` before starting to debug your script to clear the base workspace.

Variables

In this section...

“Rules for Variable Names” on page 17-25

“Making Sure Variable Names Are Valid” on page 17-25

“Do Not Use Function Names for Variables” on page 17-26

“Checking for Reserved Keywords” on page 17-26

“Avoid Using i and j for Variables” on page 17-26

“Avoid Overwriting Variables in Scripts” on page 17-27

“Persistent Variables” on page 17-27

“Protecting Persistent Variables” on page 17-27

“Global Variables” on page 17-27

Rules for Variable Names

Although variable names can be of any length, MATLAB uses only the first *N* characters of the name, (where *N* is the number returned by the function `namelengthmax`), and ignores the rest. Hence, it is important to make each variable name unique in the first *N* characters to enable MATLAB to distinguish variables. Also note that variable names are case sensitive.

`N = namelengthmax`

`N =`

63

For more information: See “Variable Names” on page 1-10.

Making Sure Variable Names Are Valid

Before using a new variable name, you can check to see if it is valid with the `isvarname` function. Note that `isvarname` does not consider names longer than `namelengthmax` characters to be valid.

For example, the following name cannot be used for a variable since it begins with a number.

```
isvarname 8thColumn
ans =
     0
```

For more information: See “Variable Names” on page 1-10.

Do Not Use Function Names for Variables

When naming a variable, make sure you are not using a name that is already used as a function name. If you do define a variable with a function name, you will not be able to call that function until you `clear` the variable from memory. (If it is a MATLAB built-in function, then you will still be able to call that function but you must do so using `builtin`.)

To test whether a proposed variable name is already used as a function name, use

```
which -all name
```

For more information: See “Conflicts with Function Names” on page 1-10.

Checking for Reserved Keywords

MATLAB reserves certain keywords for its own use and does not allow you to override them. Attempts to use these words may result in any one of a number of error messages, some of which are shown here:

```
Error: Expected a variable, function, or constant, found "=".
Error: "End of Input" expected, "case" found.
Error: Missing operator, comma, or semicolon.
Error: "identifier" expected, "=" found.
```

Use the `iskeyword` function with no input arguments to list all reserved words.

Avoid Using i and j for Variables

MATLAB uses the characters `i` and `j` to represent imaginary units. Avoid using `i` and `j` for variable names if you intend to use them in complex arithmetic.

If you want to create a complex number without using `i` and `j`, you can use the `complex` function.

Avoid Overwriting Variables in Scripts

MATLAB scripts store their variables in a workspace that is shared with the caller of the script. When called from the command line, they share the base workspace. When called from a function, they share that function's workspace. If you run a script that alters a variable that already exists in the caller's workspace, that variable is overwritten by the script.

For more information: See `Scripts`.

Persistent Variables

To get the equivalent of a static variable in MATLAB, use `persistent`. When you declare a variable to be `persistent` within a function, its value is retained in memory between calls to that function. Unlike `global` variables, `persistent` variables are known only to the function in which they are declared.

For more information: See `Persistent Variables`.

Protecting Persistent Variables

You can inadvertently clear `persistent` variables from memory by either modifying the function in which the variables are defined, or by clearing the function with one of the following commands:

```
clear all
clear functions
```

Locking the file in memory with `mlock` prevents any `persistent` variables defined in the file from being reinitialized.

Global Variables

Use global variables sparingly. The global workspace is shared by all of your functions and also by your interactive MATLAB session. The more global variables you use, the greater the chances of unintentionally reusing a

variable name, thus leaving yourself open to having those variables change in value unexpectedly. This can be a difficult bug to track down.

For more information: See Global Variables.

Strings

In this section...

“Creating Strings with Concatenation” on page 17-29
“Comparing Methods of Concatenation” on page 17-29
“Store Arrays of Strings in a Cell Array” on page 17-30
“Converting Between Strings and Cell Arrays” on page 17-30
“Search and Replace Using Regular Expressions” on page 17-30

Creating Strings with Concatenation

Strings are often created by concatenating smaller elements together (e.g., strings, values, etc.). Two common methods of concatenating are to use the MATLAB concatenation operator (`[]`) or the `sprintf` function. The second and third line below illustrate both of these methods. Both lines give the same result:

```
numChars = 28;  
s = ['There are ' int2str(numChars) ' characters here']  
s = sprintf('There are %d characters here', numChars)
```

For more information: See “Creating Character Arrays” on page 6-2 and Converting from Numeric to String.

Comparing Methods of Concatenation

When building strings with concatenation, `sprintf` is often preferable to `[]` because

- It is easier to read, especially when forming complicated expressions
- It gives you more control over the output format
- It often executes more quickly

You can also concatenate using the `strcat` function. However, for simple concatenations, `sprintf` and `[]` are faster.

Store Arrays of Strings in a Cell Array

It is usually best to store an array of strings in a cell array instead of a character array, especially if the strings are of different lengths. Strings in a character array must be of equal length, which often requires padding the strings with blanks. This is not necessary when using a cell array of strings that has no such requirement.

The `cellRecord` below does not require padding the strings with spaces:

```
cellRecord = {'Allison Jones'; 'Development'; 'Phoenix'};
```

For more information: See Cell Arrays of Strings.

Converting Between Strings and Cell Arrays

You can convert between standard character arrays and cell arrays of strings using the `cellstr` and `char` functions:

```
charRecord = ['Allison Jones'; 'Development  '; ...  
             'Phoenix      '];  
cellRecord = cellstr(charRecord);
```

Also, a number of the MATLAB string operations can be used with either character arrays, or cell arrays, or both:

```
cellRecord2 = {'Brian Lewis'; 'Development'; 'Albuquerque'};  
strcmp(charRecord, cellRecord2)  
ans =  
     0  
     1  
     0
```

For more information: See Converting to a Cell Array of Strings and String Comparisons.

Search and Replace Using Regular Expressions

Using regular expressions in MATLAB offers a very versatile way of searching for and replacing characters or phrases within a string. See the help on these functions for more information.

Function	Description
regexp	Match regular expression.
regexpi	Match regular expression, ignoring case.
regexprep	Replace string using regular expression.

For more information: See “Regular Expressions” on page 2-40.

Evaluating Expressions

In this section...

“Find Alternatives to Using `eval`” on page 17-32

“Assigning to a Series of Variables” on page 17-32

“Short-Circuit Logical Operators” on page 17-32

“Changing the Counter Variable within a for Loop” on page 17-33

Find Alternatives to Using `eval`

While the `eval` function can provide a convenient solution to certain programming challenges, it is best to limit its use. The main reason is that code that uses `eval` is often difficult to read and hard to debug. A second reason is that an `eval` statement that contains one or more commands will hide any dependencies on those commands from the MATLAB Compiler.

If you are evaluating a function, it is more efficient to use `feval` than `eval`. The `feval` function is made specifically for this purpose and is optimized to provide better performance.

For more information: See “Alternatives to the `eval` Function” on page 2-108.

Assigning to a Series of Variables

One common pattern for creating variables is to use a variable name suffixed with a number (e.g., `phase1`, `phase2`, `phase3`, etc.). We recommend using a cell array to build this type of variable name series, as it makes code more readable and executes more quickly than some other methods. For example:

```
for k = 1:800
    phase{k} = expression;
end
```

Short-Circuit Logical Operators

MATLAB has logical AND and OR operators (`&&` and `||`) that enable you to partially evaluate, or *short-circuit*, logical expressions. Short-circuit operators

are useful when you want to evaluate a statement only when certain conditions are satisfied.

In this example, MATLAB does not execute the function `myfun` unless the file that defines `myfun` exists on the current path.

```
comp = (exist('myfun.m') == 2) && (myfun(x) >= y)
```

For more information: See “Short-Circuit Operators” on page 2-10.

Changing the Counter Variable within a for Loop

You cannot change the value of the loop counter variable (e.g., the variable `k` in the example below) in the body of a `for` loop. For example, this loop executes just 10 times, even though `k` is set back to 1 on each iteration.

```
for k = 1:10
    fprintf('Pass %d\n', k)
    k = 1;
end
```

Although MATLAB does allow you to use a variable of the same name as the loop counter within a loop, this is not a recommended practice.

MATLAB Path

In this section...
“Precedence Rules” on page 17-34
“Adding a Folder to the Search Path” on page 17-35
“Handles to Functions Not on the Path” on page 17-35
“Making Toolbox File Changes Visible to MATLAB” on page 17-36
“Making Nontoolbox File Changes Visible to MATLAB” on page 17-36
“Change Notification on Windows” on page 17-37

Precedence Rules

When MATLAB is given a name to interpret, it determines its usage by checking the name against each of the entities listed below, and in the order shown:

- 1 Variable
- 2 Nested function within the current function
- 3 Local function within the current file
- 4 Private function
- 5 Class constructor
- 6 Overloaded method
- 7 Function in the current folder
- 8 Function elsewhere on the path, in order of appearance

If you refer to a file by its filename only (leaving out the file extension), and there is more than one file of this name in the folder, MATLAB selects the file to use according to the following precedence:

- 1 Built-in function

- 2 MEX-function
- 3 Simulink model, with file types in this order:
 - a SLX file
 - b MDL file
- 4 P-file (that is, an encoded program file with a .p extension)
- 5 Program file with a .m extension

For more information: See “Function Precedence Order” on page 14-35.

Adding a Folder to the Search Path

To add a folder to the search path, use either of the following:

- At the toolbar, select **File > Set Path**.
- At the command line, use the `addpath` function.

You can also add a folder and all of its subfolders in one operation by either of these means. To do this from the command line, use `genpath` together with `addpath`. The online help for the `genpath` function shows how to do this.

This example adds `/control` and all of its subfolders to the MATLAB path:

```
addpath(genpath('K:/toolbox/control'))
```

For more information: See Search Path.

Handles to Functions Not on the Path

You cannot create handles to functions that are not on the MATLAB path. But you can achieve essentially the same thing by creating the handles through a script file placed in the same off-path folder as the functions. If you then run the script, using `run path/script`, you will have created the handles that you need.

For example,

- 1 Create a script in this off-path folder that constructs function handles and assigns them to variables. That script might look something like this:

```
File E:/testdir/createFhandles.m
    fhset = @setItem
    fhsort = @sortItems
    fhdel = @deleteItem
```

- 2 Run the script from your Current Folder to create the function handles:

```
run E:/testdir/createFhandles
```

- 3 You can now execute one of the functions by means of its handle.

```
fhset(item, value)
```

Making Toolbox File Changes Visible to MATLAB

Unlike functions in user-supplied folders, MATLAB function files (and MEX-files) in the *matlabroot/toolbox* folders are not time-stamp checked, so MATLAB does not automatically see changes to them. If you modify one of these files, and then rerun it, you may find that the behavior does not reflect the changes that you made. This is most likely because MATLAB is still using the previously loaded version of the file.

To force MATLAB to reload a function from disk, you need to explicitly clear the function from memory using `clear functionname`. Note that there are rare cases where `clear` will not have the desired effect, (for example, if the file is locked, or if it is a class constructor and objects of the given class exist in memory).

Similarly, MATLAB does not automatically detect the presence of new files in *matlabroot/toolbox* folders. If you add (or remove) files from these folders, use `rehash toolbox` to force MATLAB to see your changes. Note that if you use the MATLAB Editor to create files, these steps are unnecessary, as the Editor automatically informs MATLAB of such changes.

Making Nontoolbox File Changes Visible to MATLAB

For functions outside of the toolbox folders, MATLAB sees the changes made to these files by comparing timestamps and reloads any file that has changed the next time you execute the corresponding function.

If MATLAB does not see the changes you make to one of these files, try clearing the old copy of the function from memory using `clear functionname`. You can verify that MATLAB has cleared the function using `inmem` to list all functions currently loaded into memory.

Change Notification on Windows

If MATLAB, running on Windows, is unable to see new files or changes you have made to an existing file, the problem may be related to operating system change notification handles.

Type the following for more information:

```
help changeNotification
help changeNotificationAdvanced
```

Program Control

In this section...
“Using break, continue, and return” on page 17-38
“Using switch Versus if” on page 17-39
“MATLAB case Evaluates Strings” on page 17-39
“Multiple Conditions in a case Statement” on page 17-39
“Implicit Break in switch-case” on page 17-39
“Variable Scope in a switch” on page 17-40
“Catching Errors with try-catch” on page 17-40
“Nested try-catch Blocks” on page 17-41
“Forcing an Early Return from a Function” on page 17-41

Using break, continue, and return

It is easy to confuse the break, continue, and return functions as they are similar in some ways. Make sure you use these functions appropriately.

Function	Where to Use It	Description
break	for or while loops	Exits the loop in which it appears. In nested loops, control passes to the next outer loop.
continue	for or while loops	Skips any remaining statements in the current loop. Control passes to next iteration of the same loop.
return	Anywhere	Immediately exits the function in which it appears. Control passes to the caller of the function.

Using switch Versus if

It is possible, but usually not advantageous, to implement switch-case statements using if-elseif instead. See pros and cons in the table.

switch-case Statements	if-elseif Statements
Easier to read.	Can be difficult to read.
Can compare strings of different lengths.	You need strcmp to compare strings of different lengths.
Test for equality only.	Test for equality or inequality.

MATLAB case Evaluates Strings

A useful difference between switch-case statements in MATLAB and C is that you can specify string values in MATLAB case statements, which you cannot do in C.

```
switch(method)
    case 'linear'
        disp('Method is linear')
    case 'cubic'
        disp('Method is cubic')
end
```

Multiple Conditions in a case Statement

You can test against more than one condition with switch. The first case below tests for either a linear or bilinear method by using a cell array in the case statement.

```
switch(method)
    case {'linear', 'bilinear'}
        disp('Method is linear or bilinear')
    case (<and so on>)
end
```

Implicit Break in switch-case

In C, if you do not end each case with a break statement, code execution falls through to the following case. In MATLAB, case statements do not fall

through; only one case may execute. Using `break` within a case statement is not only unnecessary, it is also invalid and generates a warning.

In this example, if `result` is 52, only the first `disp` statement executes, even though the second is also a valid match:

```
switch(result)
    case 52
        disp('result is 52')
    case {52, 78}
        disp('result is 52 or 78')
end
```

Variable Scope in a switch

Since MATLAB executes only one case of any `switch` statement, variables defined within one case are not known in the other cases of that `switch` statement. The same holds true for `if-elseif` statements.

In these examples, you get an error when `choice` equals 2, because `x` is undefined.

```
-- SWITCH-CASE --
switch choice
    case 1
        x = -pi:0.01:pi;
    case 2
        plot(x, sin(x));
end

-- IF-ELSEIF --
if choice == 1
    x = -pi:0.01:pi;
elseif choice == 2
    plot(x, sin(x));
end
```

Catching Errors with try-catch

When you have statements in your code that could possibly generate unwanted results, put those statements into a `try-catch` block that will catch any errors and handle them appropriately.

The example below shows a `try-catch` block within a function that multiplies two matrices. If a statement in the `try` segment of the block fails, control passes to the `catch` segment. In this case, the `catch` statements check the error message that was issued (returned in `MException` object, `err`) and respond appropriately:


```
try
    X = A * B
catch err
    errmsg = err.message;
    if(strfind(errmsg, 'Inner matrix dimensions'))
        disp('** Wrong dimensions for matrix multiply')
    end
end
```

For more information: See “The try-catch Statement” on page 18-18.

Nested try-catch Blocks

You can also nest try-catch blocks, as shown here. You can use this to attempt to recover from an error caught in the first try section:

```
try
    statement1                    % Try to execute statement1
catch
    try
        statement2                % Attempt to recover from error
    catch
        disp 'Operation failed'    % Handle the error
    end
end
end
```

Forcing an Early Return from a Function

To force an early return from a function, place a return statement in the function at the point where you want to exit. For example,

```
if <done>
    return
end
```

Save and Load

In this section...
“Saving Data from the Workspace” on page 17-42
“Loading Data into the Workspace” on page 17-42
“Viewing Variables in a MAT-File” on page 17-43
“Appending to a MAT-File” on page 17-43
“Save and Load on Startup or Quit” on page 17-44
“Saving to an ASCII File” on page 17-44

Saving Data from the Workspace

To save data from your workspace, you can do any of the following:

- Copy from the MATLAB Command Window and paste into a text file.
- Record part of your session in a diary file, and then edit the file in a text editor.
- Save to a binary or ASCII file using the `save` function.
- Save spreadsheet, scientific, image, or audio data with appropriate function.
- Save to a file using low-level file I/O functions (`fwrite`, `fprintf`, ...).

For more information: See Saving the Current Workspace and “Writing to Text Data Files with Low-Level I/O”.

Loading Data into the Workspace

Similarly, to load new or saved data into the workspace, you can do any of the following:

- Enter or paste data at the command line.
- Create a script file to initialize large matrices or data structures.
- Read a binary or ASCII file using `load`.

- Load spreadsheet, scientific, image, or audio data with appropriate function.
- Load from a file using low-level file I/O functions (`fread`, `fscanf`, ...).

For more information: See Loading a Saved Workspace and Importing Data and “Importing Data”.

Viewing Variables in a MAT-File

To see what variables are saved in a MAT-file, use `who` or `whos` as shown here (the `.mat` extension is not required). `who` returns a cell array and `whos` returns a structure array.

```
mydataVariables = who('-file', 'mydata.mat');
```

Appending to a MAT-File

To save additional variables to an existing MAT-file, use

```
save matfilename -append
```

Any variables you save that do not yet exist in the MAT-file are added to the file. Any variables you save that already exist in the MAT-file overwrite the old values.

Note Saving with the `-append` switch does not append additional elements to an array that is already saved in a MAT-file. See the example below.

In this example, the second `save` operation does not concatenate new elements to vector `A`, (making `A` equal to `[1 2 3 4 5 6 7 8]`) in the MAT-file. Instead, it replaces the 5 element vector, `A`, with a 3 element vector, also retaining all other variables that were stored on the first `save` operation.

```
A = [1 2 3 4 5];    B = 12.5;    C = rand(4);  
save savefile;  
A = [6 7 8];  
save savefile A -append;
```

Save and Load on Startup or Quit

You can automatically save your variables at the end of each MATLAB session by creating a `finish.m` file to save the contents of your base workspace every time you quit MATLAB. Load these variables back into your workspace at the beginning of each session by creating a `startup.m` file that uses the `load` function to load variables from your MAT-file.

For more information: See the `startup` and `finish` function reference pages.

Saving to an ASCII File

When you save matrix data to an ASCII file using `save -ascii`, MATLAB combines the individual matrices into one collection of numbers. Variable names are not saved. If this is not acceptable for your application, use `fprintf` to store your data instead.

For more information: See “Writing to Delimited Data Files”.

Files and Filenames

In this section...

“Naming Functions” on page 17-45

“Naming Other Files” on page 17-45

“Passing Filenames as Arguments” on page 17-46

“Passing Filenames to ASCII Files” on page 17-46

“Determining Filenames at Run-Time” on page 17-46

“Returning the Size of a File” on page 17-46

Naming Functions

A valid name for a MATLAB function file is composed of a string of letters, digits, and underscores, totaling not more than `namelengthmax` characters and beginning with a letter.

`N = namelengthmax`

`N =`

63

Since variables must obey similar rules, you can use the `isvarname` function to check whether a filename (minus its `.m` file extension) is valid for a MATLAB function file.

```
isvarname mfilename
```

Naming Other Files

The names of other files that MATLAB interacts with (e.g., MAT, MEX, and MDL-files) follow the same rules as the MATLAB function files, but may be of any length.

Depending on your operating system, you may be able to include certain nonalphanumeric characters in your filenames. Check your operating system manual for information on valid filename restrictions.

Passing Filenames as Arguments

In MATLAB commands, you can specify a filename argument using the MATLAB command or function syntax. For example, either of the following are acceptable. (The `.mat` file extension is optional for `save` and `load`).

```
load mydata.mat           % Command syntax
load('mydata.mat')      % Function syntax
```

If you assign the output to a variable, you must use the function syntax.

```
savedData = load('mydata.mat')
```

Passing Filenames to ASCII Files

ASCII files are specified as follows. Here, the file extension is required.

```
load mydata.dat -ascii   % Command syntax
load('mydata.dat', '-ascii') % Function syntax
```

Determining Filenames at Run-Time

There are several ways that your function code can work on specific files without you having to hardcode their filenames into the program. You can

- Pass the filename in as an argument

```
function myfun(datafile)
```

- Prompt for the filename using the `input` function

```
filename = input('Enter name of file: ', 's');
```

- Browse for the file using the `uigetfile` function

```
[filename, pathname] =
    uigetfile('*.mat', 'Select MAT-file');
```

For more information: See the `input` and `uigetfile` function reference pages.

Returning the Size of a File

Two ways to have your program determine the size of a file are shown here.

```
-- METHOD #1 --
s = dir('myfile.dat');
filesize = s.bytes

-- METHOD #2 --
fid = fopen('myfile.dat');
fseek(fid, 0, 'eof');
filesize = ftell(fid)
fclose(fid);
```

The `dir` function also returns the filename (`s.name`), last modification date (`s.date`), and whether or not it is a folder (`s.isdir`).

(The second method requires read access to the file.)

For more information: See the `fopen`, `fseek`, `ftell`, and `fclose` function reference pages.

Input/Output

In this section...
“Common I/O Functions” on page 17-48
“Loading Mixed Format Data” on page 17-48
“Reading Files with Different Formats” on page 17-49
“Interactive Input into Your Program” on page 17-49

For more information and examples on importing and exporting data, see *MATLAB Data Import and Export*.

Common I/O Functions

The most commonly used, high-level, file I/O functions in MATLAB are `save` and `load`. For help on these, type `doc save` or `doc load`.

Functions for I/O to text files with delimited values are `textscan`, `dlmread`, `dlmwrite`.

To select and import data from files interactively, select **File > Import Data**.

For more information: See “Supported File Formats”.

Loading Mixed Format Data

To load data that is in mixed formats, use `textscan` instead of `load`. The `textscan` function lets you specify the format of each piece of data.

If the first line of file `mydata.dat` is

```
Sally    12.34 45
```

Read the first line of the file as a free format file using the `%` format:

```
fid = fopen('mydata.dat');  
c = textscan(fid, '%s %f %d', 1);  
fclose(fid);
```


returns

```
c =  
    {1x1 cell}    [12.3400]    [45]
```

Reading Files with Different Formats

Attempting to read data from a file that was generated on a different platform may result in an error because the binary formats of the platforms may differ. Using the `fopen` function, you can specify a machine format when you open the file to avoid these errors.

Interactive Input into Your Program

Your program can accept interactive input from users during execution. Use the `input` function to prompt the user for input, and then read in a response. When executed, `input` causes the program to display your prompt, pause while a response is entered, and then resume when the **Enter** key is pressed.

Starting MATLAB

Getting MATLAB to Start Up Faster

Here are some things that you can do to make MATLAB start up faster.

- Make sure toolbox path caching is enabled.
- Make sure that the system on which MATLAB is running has enough RAM.
- Choose only the windows you need in the MATLAB desktop.
- Close the Help browser before exiting MATLAB. When you start your next session, MATLAB will not open the Help browser, and thus will start faster.
- If disconnected from the network, check the `LM_LICENSE_FILE` variable. See <http://www.mathworks.com/support/solutions/data/1-17VEB.html> for a more detailed explanation.

For more information: See Toolbox Path Caching in MATLAB.

Operating System Compatibility

In this section...

“Executing O/S Commands from MATLAB” on page 17-51

“Searching Text with grep” on page 17-51

“Constructing Paths and Filenames” on page 17-51

“Finding the MATLAB Root Folder” on page 17-52

“Temporary Directories and Filenames” on page 17-52

Executing O/S Commands from MATLAB

To execute a command from your operating system prompt without having to exit MATLAB, precede the command with the MATLAB ! operator.

On Windows, you can add an ampersand (&) to the end of the line to make the output appear in a separate window.

For more information: See Running External Programs and the system and dos function reference pages.

Searching Text with grep

grep is a powerful tool for performing text searches in files on UNIX systems. To grep from within MATLAB, precede the command with an exclamation point (!grep).

For example, to search for the word warning in all MATLAB function files of the Current Folder, ignoring case, you would use

```
!grep -i 'warning' *.m
```

Constructing Paths and Filenames

Use the fullfile function to construct path names and filenames rather than entering them as strings into your programs. In this way, you always get the correct path specification, regardless of which operating system you are using at the time.

Finding the MATLAB Root Folder

The `matlabroot` function returns the location of the MATLAB installation on your system. Use `matlabroot` to create a path to MATLAB and toolbox folders that does not depend on a specific platform or MATLAB version.

The following example uses `matlabroot` with `fullfile` to return a platform-independent path to the general toolbox folder:

```
fullfile(matlabroot, 'toolbox', 'matlab', 'general')
```

Temporary Directories and Filenames

If you need to locate the folder on your system that has been designated to hold temporary files, use the `tempdir` function. `tempdir` returns a string that specifies the path to this folder.

To create a new file in this folder, use the `tempname` function. `tempname` returns a string that specifies the path to the temporary file folder, plus a unique filename.

For example, to store some data in a temporary file, you might issue the following command first.

```
fid = fopen(tempname, 'w');
```

For More Information

In this section...
“Current CSSM” on page 17-53
“Archived CSSM” on page 17-53
“MATLAB Technical Support” on page 17-53
“MATLAB Central” on page 17-53
“MATLAB Newsletters (Digest, News & Notes)” on page 17-53
“MATLAB Documentation” on page 17-53
“MATLAB Index of Examples” on page 17-54

Current CSSM

<http://www.mathworks.com/matlabcentral/newsreader>

Archived CSSM

<http://mathforum.org/kb/forum.jspa?forumID=80>

MATLAB Technical Support

<http://www.mathworks.com/support/>

MATLAB Central

<http://www.mathworks.com/matlabcentral/>

MATLAB Newsletters (Digest, News & Notes)

<http://www.mathworks.com/company/newsletters/index.html>

MATLAB Documentation

<http://www.mathworks.com/help/>

MATLAB Index of Examples

http://www.mathworks.com/help/techdoc/demo_example.html

Software Development

- Chapter 18, “Error Handling”
- Chapter 19, “Program Scheduling”
- Chapter 20, “Performance”
- Chapter 21, “Memory Usage”
- Chapter 22, “Create Help and Demos”

Error Handling

- “Error Reporting in a MATLAB Application” on page 18-2
- “Capturing Information About the Error” on page 18-5
- “Throwing an Exception” on page 18-16
- “Responding to an Exception” on page 18-18
- “Warnings” on page 18-23
- “Warning Control” on page 18-25
- “Debugging Errors and Warnings” on page 18-37

Error Reporting in a MATLAB Application

In this section...

“Overview” on page 18-2

“Getting an Exception at the Command Line” on page 18-2

“Getting an Exception in Your Program Code” on page 18-3

“Generating a New Exception” on page 18-4

Overview

No matter how carefully you plan and test the programs you write, they may not always run as smoothly as expected when executed under different conditions. It is always a good idea to include error checking in programs to ensure reliable operation under all conditions.

In the MATLAB software, you can decide how your programs respond to different types of errors. You may want to prompt the user for more input, display extended error or warning information, or perhaps repeat a calculation using default values. The error-handling capabilities in MATLAB help your programs check for particular error conditions and execute the appropriate code depending on the situation.

When MATLAB detects a severe fault in the command or program it is running, it collects information about what was happening at the time of the error, displays a message to help the user understand what went wrong, and terminates the command or program. This is called *throwing an exception*. You can get an exception while entering commands at the MATLAB command prompt or while executing your program code.

Getting an Exception at the Command Line

If you get an exception at the MATLAB prompt, you have several options on how to deal with it as described below.

Determine the Fault from the Error Message

Evaluate the error message MATLAB has displayed. Most error messages attempt to explain at least the immediate cause of the program failure. There

is often sufficient information to determine the cause and what you need to do to remedy the situation.

Review the Failing Code

If the function in which the error occurred is implemented as a MATLAB program file, the error message should include a line that looks something like this:

```
surf
```

```
Error using surf (line 50)  
Not enough input arguments.
```

The text includes the name of the function that threw the error (`surf`, in this case) and shows the failing line number within that function's program file. Click the line number; MATLAB opens the file and positions the cursor at the location in the file where the error originated. You may be able to determine the cause of the error by examining this line and the code that precedes it.

Step Through the Code in the Debugger

You can use the MATLAB Debugger to step through the failing code. Click the underlined error text to open the file in the MATLAB Editor at or near the point of the error. Next, click the hyphen at the beginning of that line to set a breakpoint at that location. When you rerun your program, MATLAB pauses execution at the breakpoint and enables you to step through the program code. The command `dbstop on error` is also helpful in finding the point of error.

See the documentation on “Debugging Process and Features” for more information.

Getting an Exception in Your Program Code

When you are writing your own program in a program file, you can *catch* exceptions and attempt to handle or resolve them instead of allowing your program to terminate. When you catch an exception, you interrupt the normal termination process and enter a block of code that deals with the faulty situation. This block of code is called a *catch block*.

Some of the things you might want to do in the catch block are:

- Examine information that has been captured about the error.
- Gather further information to report to the user.
- Try to accomplish the task at hand in some other way.
- Clean up any unwanted side effects of the error.

When you reach the end of the catch block, you can either continue executing the program, if possible, or terminate it.

The documentation on “Capturing Information About the Error” on page 18-5 describes how to acquire information about what caused the error, and “Responding to an Exception” on page 18-18 presents some ideas on how to respond to it.

Generating a New Exception

When your program code detects a condition that will either make the program fail or yield unacceptable results, it should throw an exception. This procedure

- Saves information about what went wrong and what code was executing at the time of the error.
- Gathers any other pertinent information about the error.
- Instructs MATLAB to throw the exception.

The documentation on “Capturing Information About the Error” on page 18-5 describes how to use an `MException` object to capture information about the error, and “Throwing an Exception” on page 18-16 explains how to initiate the exception process.

Capturing Information About the Error

In this section...

“Overview” on page 18-5

“The MException Class” on page 18-5

“Properties of the MException Class” on page 18-7

“Methods of the MException Class” on page 18-14

Overview

When the MATLAB software throws an exception, it captures information about what caused the error in a data structure called an MException object. This object is an instance of the MATLAB MException class. You can obtain access to the MException object by *catching* the exception before your program aborts and accessing the object constructed for this particular error via the catch command. When throwing an exception in response to an error in your own code, you will have to create a new MException object and store information about the error in that object.

This section describes the MException class and objects constructed from that class:

Information on how to use this class is presented in later sections on “Responding to an Exception” on page 18-18 and “Throwing an Exception” on page 18-16.

The MException Class

The figure shown below illustrates one possible configuration of an object of the MException class. The object has four properties: `identifier`, `message`, `stack`, and `cause`. Each of these properties is implemented as a field of the structure that represents the MException object. The `stack` field is an N-by-1 array of additional structures, each one identifying a function, and line number from the call stack. The `cause` field is an M-by-1 cell array of MException objects, each representing an exception that is related to the current one.

where `identifier` is a MATLAB message identifier of the form

```
component:mnemonic
```

that is enclosed in single quotes, and `message` is a text string, also enclosed in single quotes, that describes the error. The output `ME` is the resulting `MException` object.

If you are responding to an exception rather than throwing one, you do not have to construct an `MException` object. The object has already been constructed and populated by the code that originally detected the error.

Properties of the `MException` Class

The `MException` class has four properties. Each of these properties is implemented as a field of the structure that represents the `MException` object. Each of these properties is described in the sections below and referenced in the sections on “Responding to an Exception” on page 18-18 and “Throwing an Exception” on page 18-16. All are read-only; their values cannot be changed.

The `MException` properties are:

- `identifier`
- `message`
- `stack`
- `cause`

Repeating the `surf` example shown above, but this time catching the exception, you can see the four properties of the `MException` object structure. (This example uses `try-catch` in an atypical fashion. See the section on “The `try-catch` Statement” on page 18-18 for more information on using `try-catch`).

```
try
    surf
catch ME
    ME
end
```

Run this at the command line and MATLAB returns the contents of the MException object:

```
ME =  
MException object with properties:  
  
    identifier: 'MATLAB:narginchk:notEnoughInputs'  
    message: 'Not enough input arguments.'  
    stack: [1x1 struct]  
    cause: {}
```

The `stack` field shows the filename, function, and line number where the exception was thrown:

```
ME.stack  
ans =  
    file: 'matlabroot\toolbox\matlab\graph3d\surf.m'  
    name: 'surf'  
    line: 54
```

The `cause` field is empty in this case. Each field is described in more detail in the sections that follow.

Message Identifiers

A message identifier is a tag that you attach to an error or warning statement that makes that error or warning uniquely recognizable by MATLAB. You can use message identifiers with error reporting to better identify the source of an error, or with warnings to control any selected subset of the warnings in your programs.

The message identifier is a read-only character string that specifies a *component* and a *mnemonic* label for an error or warning. The format of a simple identifier is

```
component:mnemonic
```

A colon separates the two parts of the identifier: `component` and `mnemonic`. If the identifier uses more than one `component`, then additional colons are required to separate them. A message identifier must always contain at least one colon.

Some examples of message identifiers are

```
MATLAB:rmpath:DirNotFound
MATLAB:odearguments:InconsistentDataType
Simulink:actionNotTaken
TechCorp:OpenFile:notFoundInPath
```

Both the component and mnemonic fields must adhere to the following syntax rules:

- No white space (space or tab characters) is allowed anywhere in the identifier.
- The first character must be alphabetic, either uppercase or lowercase.
- The remaining characters can be alphanumeric or an underscore.

There is no length limitation to either the component or mnemonic. The identifier can also be an empty string.

Component Field. The component field specifies a broad category under which various errors and warnings can be generated. Common components are a particular product or toolbox name, such as `MATLAB` or `Control`, or perhaps the name of your company, such as `TechCorp` in the preceding example.

You can also use this field to specify a multilevel component. The following statement has a three-level component followed by a mnemonic label:

```
TechCorp:TestEquipDiv:Waveform:obsoleteSyntax
```

The component field enables you to guarantee the uniqueness of each identifier. Thus, while the internal MATLAB code might use a certain warning identifier like `MATLAB:InconsistentDataType`, that does not preclude you from using the same mnemonic, as long as you precede it with a unique component. For example,

```
warning('TechCorp:InconsistentDataType', ...
    'Value %s is inconsistent with existing properties.' ...
    sprocketDiam)
```

Mnemonic Field. The mnemonic field is a string normally used as a tag relating to the particular message. For example, when reporting an error resulting from the use of ambiguous syntax, a simple component and mnemonic such as the following might be appropriate:

```
MATLAB:ambiguousSyntax
```

Message Identifiers in an MException Object. When throwing an exception, create an appropriate identifier and save it to the MException object at the time you construct the object using the syntax

```
ME = MException(identifier, string)
```

For example,

```
ME = MException('AcctError:NoClient', ...  
               'Client name not recognized.');
```

```
ME.identifier  
ans =  
    AcctError:NoClient
```

When responding to an exception, you can extract the message identifier from the MException object as shown here. Using the surf example again,

```
try  
    surf  
catch ME  
    id = ME.identifier  
end  
  
id =  
    MATLAB:narginchk:notEnoughInputs
```

Text of the Error Message

An error message in MATLAB is a read-only character string issued by the program code and returned in the MException object. This message can assist the user in determining the cause, and possibly the remedy, of the failure.

When throwing an exception, compose an appropriate error message and save it to the `MException` object at the time you construct the object using the syntax

```
ME = MException(identifier, string)
```

If your message string requires formatting specifications, like those available with the `sprintf` function, use this syntax for the `MException` constructor:

```
ME = MException(identifier, formatstring, arg1, arg2, ...)
```

For example,

```
S = 'Accounts'; f1 = 'ClientName';  
ME = MException('AcctError:Incomplete', ...  
    'Field ''%.%s'' is not defined.', S, f1);
```

```
ME.message  
ans =  
    Field 'Accounts.ClientName' is not defined.
```

When responding to an exception, you can extract the error message from the `MException` object as follows:

```
try  
    surf  
catch ME  
    msg = ME.message  
end  
  
msg =  
    Not enough input arguments.
```

The Call Stack

The `stack` field of the `MException` object identifies the line number, function, and filename where the error was detected. If the error occurs in a called function, as in the following example, the `stack` field contains the line number, function name, and filename not only for the location of the immediate error, but also for each of the calling functions. In this case, `stack`

is an N-by-1 array, where N represents the depth of the call stack. That is, the stack field displays the function name and line number where the exception occurred, the name and line number of the caller, the caller's caller, etc., until the top-most function is reached.

When throwing an exception, MATLAB stores call stack information in the stack field. You cannot write to this field; access is read-only.

For example, suppose you have three functions that reside in two separate files:

```
mfileA.m
=====
      .
      .
42 function A1(x, y)
43 B1(x, y);
```

```
mfileB.m
=====
      .
      .
 8 function B1(x, y)
 9 B2(x, y)
      .
      .
26 function B2(x, y)
27      .
28      .
29      .
30      .
31 % Throw exception here
```

Catch the exception in variable ME and then examine the stack field:

```
for k=1:length(ME.stack)
    ME.stack(k)
end
```

```

ans =
    file: 'C:\matlab\test\mfileB.m'
    name: 'B2'
    line: 31
ans =
    file: 'C:\matlab\test\mfileB.m'
    name: 'B1'
    line: 9
ans =
    file: 'C:\matlab\test\mfileA.m'
    name: 'A1'
    line: 43

```

The Cause Array

In some situations, it can be important to record information about not only the one command that caused execution to stop, but also other exceptions that your code caught. You can save these additional `MException` objects in the `cause` field of the primary exception.

The `cause` field of an `MException` is an optional cell array of related `MException` objects. You must use the following syntax when adding objects to the `cause` cell array:

```
primaryException = addCause(primaryException, secondaryException)
```

This example attempts to assign an array `D` to variable `X`. If the `D` array does not exist, the code attempts to load it from a `MAT`-file and then retries assigning it to `X`. If the load fails, a new `MException` object (`ME3`) is constructed to store the cause of the first two errors (`ME1` and `ME2`):

```

try
    X = D(1:25)
catch ME1
    try
        filename = 'test200';
        load(filename);
        X = D(1:25)
    catch ME2
        ME3 = MException('MATLAB:LoadErr', ...

```

```
        'Unable to load from file %s', filename);
    ME3 = addCause(ME3, ME1);
    ME3 = addCause(ME3, ME2);
end
end
```

There are two exceptions in the cause field of ME3:

```
ME3.cause
ans =
    [1x1 MException]
    [1x1 MException]
```

Examine the cause field of ME3 to see the related errors:

```
ME3.cause{:}
ans =

MException object with properties:

    identifier: 'MATLAB:UndefinedFunction'
    message: 'Undefined function or method 'D' for input
arguments of type 'double'.'
    stack: [0x1 struct]
    cause: {}
ans =

MException object with properties:

    identifier: 'MATLAB:load:couldNotReadFile'
    message: 'Unable to read file test204: No such file or
directory.'
    stack: [0x1 struct]
    cause: {}
```

Methods of the MException Class

There are ten methods that you can use with the MException class. The names of these methods are case-sensitive. See the MATLAB function reference pages for more information.

Method Name	Description
<code>addCause</code>	Append an <code>MException</code> to the <code>cause</code> field of another <code>MException</code> .
<code>disp</code>	Display an <code>MException</code> object.
<code>eq</code>	Compare <code>MException</code> objects for equality.
<code>getReport</code>	Return a formatted message based on the current exception.
<code>isequal</code>	Compare <code>MException</code> objects for equality.
<code>last</code>	Return the last uncaught exception. This is a static method.
<code>ne</code>	Compare <code>MException</code> objects for inequality.
<code>rethrow</code>	Reissue an exception that has previously been caught.
<code>throw</code>	Issue an exception.
<code>throwAsCaller</code>	Issue an exception, but omit the current stack frame from the <code>stack</code> field.

Throwing an Exception

When your program detects a fault that will keep it from completing as expected or will generate erroneous results, you should halt further execution and report the error by throwing an exception. The basic steps to take are

- 1** Detect the error. This is often done with some type of conditional statement, such as an `if` statement that checks the output of the current operation.
- 2** Construct an `MException` object to represent the error. Add a message identifier string and error message string to the object when calling the constructor.
- 3** If there are other exceptions that may have contributed to the current error, you can store the `MException` object for each in the `cause` field of a single `MException` that you intend to throw. Use the `addCause` method for this.
- 4** Use the `throw` or `throwAsCaller` function to have the MATLAB software issue the exception. At this point, MATLAB stores call stack information in the `stack` field of the `MException`, exits the currently running function, and returns control to either the keyboard or an enclosing catch block in a calling function.

This example illustrates throwing an exception using the steps just described:

```
function check_results(resultsArr, dataFile)
minValue = 0.09;    maxValue = 2.14;

% 1) Detect the error.
if any(resultsArr < minValue) || any(resultsArr > maxValue)

    % 2) Construct an MException object to represent the error.
    err = MException('ResultChk:OutOfRange', ...
        'Resulting value is outside expected range');
    fileInfo = dir(dataFile);

    % 3) Store any information contributing to the error.
    if datenum(fileInfo.date) < datenum('Oct09','mmyy')
        errCause = MException('ResultChk:BadInput', ...
            'Input file %s is out of date.', dataFile);
```



```
        err = addCause(err, errCause);
    end

    % 4) Throw the exception to stop execution and display
        an error message.
    throw(err)
end
```

If the program detects the `OutOfRange` condition, the `throw(err)` statement throws an exception at the end. If the `BadInput` condition is also detected, the program also displays this as the cause:

```
resultsArr = [1.63, 2.05, 0.91, 2.16, 1.5, 2.11 0.72];
check_results(resultsArr, 'run33.dat')
```

```
Error using check_results (line 20)
Resulting value is outside expected range
```

```
Caused by:
    Input file run33.dat is out of date.
```

Responding to an Exception

In this section...
“Overview” on page 18-18
“The try-catch Statement” on page 18-18
“Suggestions on How to Handle an Exception” on page 18-20

Overview

As stated earlier, the MATLAB software, by default, terminates the currently running program when an exception is thrown. If you catch the exception in your program, however, you can capture information about what went wrong, and deal with the situation in a way that is appropriate for the particular condition. This requires a try-catch statement.

This section covers the following topics:

The try-catch Statement

When you have statements in your code that could generate undesirable results, put those statements into a try-catch block that catches any errors and handles them appropriately.

A try-catch statement looks something like the following pseudocode. It consists of two parts:

- A try block that includes all lines between the try and catch statements.
- A catch block that includes all lines of code between the catch and end statements.

```
try
    Perform one ...
        or more operations
A catch ME
    Examine error info in exception object ME
    Attempt to figure out what went wrong
    Either attempt to recover, or clean up and abort
```

```
end
```

B Program continues

The program executes the statements in the `try` block. If it encounters an error, it skips any remaining statements in the `try` block and jumps to the start of the `catch` block (shown here as point A). If all operations in the `try` block succeed, then execution skips the `catch` block entirely and goes to the first line following the `end` statement (point B).

Specifying the `try`, `catch`, and `end` commands and also the code of the `try` and `catch` blocks on separate lines is recommended. If you combine any of these components on the same line, separate them with commas:

```
try, surf, catch ME, ME.stack, end
ans =
    file: 'matlabroot\toolbox\matlab\graph3d\surf.m'
    name: 'surf'
    line: 54
```

Note You cannot define nested functions within a `try` or `catch` block.

The Try Block

On execution, your code enters the `try` block and executes each statement as if it were part of the regular program. If no errors are encountered, MATLAB skips the `catch` block entirely and continues execution following the `end` statement. If any of the `try` statements fail, MATLAB immediately exits the `try` block, leaving any remaining statements in that block unexecuted, and enters the `catch` block.

The Catch Block

The `catch` command marks the start of a `catch` block and provides access to a data structure that contains information about what caused the exception. This is shown as the variable `ME` in the preceding pseudocode. This data structure is an object of the MATLAB `MException` class. When an exception occurs, MATLAB constructs an instance of this class and returns it in the `catch` statement that handles that error.

You are not required to specify any argument with the `catch` statement. If you do not need any of the information or methods provided by the `MException` object, just specify the `catch` keyword alone.

The `MException` object is constructed by internal code in the program that fails. The object has properties that contain information about the error that can be useful in determining what happened and how to proceed. The `MException` object also provides access to methods that enable you to respond to the exception. See the section on “The `MException` Class” on page 18-5 to find out more about the `MException` class.

Having entered the `catch` block, MATLAB executes the statements in sequence. These statements can attempt to

- Attempt to resolve the error.
- Capture more information about the error.
- Switch on information found in the `MException` object and respond appropriately.
- Clean up the environment that was left by the failing code.

The `catch` block often ends with a `rethrow` command. The `rethrow` causes MATLAB to exit the current function, keeping the call stack information as it was when the exception was first thrown. If this function is at the highest level, that is, it was not called by another function, the program terminates. If the failing function was called by another function, it returns to that function. Program execution continues to return to higher level functions, unless any of these calls were made within a higher-level `try` block, in which case the program executes the respective `catch` block.

More information about the `MException` class is provided in the section “Capturing Information About the Error” on page 18-5.

Suggestions on How to Handle an Exception

The following example reads the contents of an image file. The `try` block attempts to open and read the file. If either the `open` or `read` fails, the program catches the resulting exception and saves the `MException` object in the variable `ME1`.

The catch block in the example checks to see if the specified file could not be found. If so, the program allows for the possibility that a common variation of the filename extension (e.g., jpeg instead of jpg) was used by retrying the operation with a modified extension. This is done using a try-catch statement nested within the original try-catch.

```
function d_in = read_image(filename)
[path name ext] = fileparts(filename);
try
    fid = fopen(filename, 'r');
    d_in = fread(fid);
catch ME1
    % Get last segment of the error message identifier.
    idSegLast = regexp(ME1.identifier, '(?<=:\)\w+$', 'match');

    % Did the read fail because the file could not be found?
    if strcmp(idSegLast, 'InvalidFid') && ...
        ~exist(filename, 'file')

        % Yes. Try modifying the filename extension.
        switch ext
        case '.jpg'      % Change jpg to jpeg
            filename = strrep(filename, '.jpg', '.jpeg')
        case '.jpeg'    % Change jpeg to jpg
            filename = strrep(filename, '.jpeg', '.jpg')
        case '.tif'     % Change tif to tiff
            filename = strrep(filename, '.tif', '.tiff')
        case '.tiff'    % Change tiff to tif
            filename = strrep(filename, '.tiff', '.tif')
        otherwise
            fprintf('File %s not found\n', filename);
            rethrow(ME1);
        end

        % Try again, with modified filenames.
        try
            fid = fopen(filename, 'r');
            d_in = fread(fid);
        catch ME2
            fprintf('Unable to access file %s\n', filename);
```

```
        ME2 = addCause(ME2, ME1);
        rethrow(ME2)
    end
end
end
```

This example illustrates some of the actions that you can take in response to an exception:

- Compare the `identifier` field of the `MException` object against possible causes of the error.
- Use a nested `try-catch` statement to retry the open and read operations using a known variation of the filename extension.
- Display an appropriate message in the case that the file truly does not exist and then rethrow the exception.
- Add the first `MException` object to the `cause` field of the second.
- Rethrow the exception. This stops program execution and displays the error message.

Cleaning up any unwanted results of the error is also advisable. For example, your program may have allocated a significant amount of memory that it no longer needs.

Warnings

In this section...
“Reporting a Warning” on page 18-23
“Identifying the Cause” on page 18-24

Reporting a Warning

Like error, the warning function alerts the user of unexpected conditions detected when running a program. However, warning does not halt the execution of the program. It displays the specified warning message and then continues.

Use warning in your code to generate a warning message during execution. Specify the message string as the input argument to warning. For example,

```
warning('Input must be a string')
```

Warnings also differ from errors in that you can disable any warnings that you do not want to see. You do this by invoking warning with certain control parameters. See “Warning Control” on page 18-25 for more information.

Formatted Message Strings

The warning message string you specify can contain formatting conversion characters, such as those used with the MATLAB `sprintf` function. Make the warning string the first argument, and add any variables used by the conversion as subsequent arguments.

```
warning('formatted_warningmsg', arg1, arg2, ...)
```

For example, if your program cannot process a given parameter, you might report a warning with

```
warning('Ambiguous parameter name, "%s".', param)
```

MATLAB converts special characters like `%d` and `%s` in the warning message string only when you specify more than one input argument with warning. See “Formatted Message Strings” on page 18-23 for information.

Message Identifiers

Use a message identifier argument with `warning` to attach a unique tag to a warning message. MATLAB uses this tag to better identify the source of a warning. The first argument in this example is the message identifier.

```
warning('MATLAB:paramAmbiguous', ...  
        'Ambiguous parameter name, "%s".', param)
```

See “Warning Control Statements” on page 18-27 for more information on how to use identifiers with warnings.

Identifying the Cause

The `lastwarn` function returns a string containing the last warning message issued by MATLAB. Use this to enable your program to identify the cause of a warning that has just been issued. To return the most recent warning message to the variable `warnmsg`, type

```
warnmsg = lastwarn;
```

You can also change the text of the last warning message with a new message or with an empty string as shown here:

```
lastwarn('newwarnmsg'); % Replace last warning with new string  
lastwarn('');          % Replace last warning with empty string
```


Warning Control

In this section...

- “Overview” on page 18-25
- “Warning Statements” on page 18-26
- “Warning Control Statements” on page 18-27
- “Output from Control Statements” on page 18-30
- “Saving and Restoring State” on page 18-32
- “Backtrace and Verbose Modes” on page 18-33

Overview

The MATLAB software gives you the ability to control what happens when a warning is encountered during program execution. Options that are available include

- Display selected warnings.
- Ignore selected warnings.
- Stop in the debugger when a warning is invoked.
- Display the stack trace after a warning is invoked.

Depending on how you set your warning controls, you can have these actions affect all warnings in your code, specific warnings that you select, or just the most recently invoked warning.

Setting up this system of warning control involves several steps.

- 1** Start by determining the scope of the control you need for the warnings generated by your code. Do you want the control operations to affect all the warnings in your code at once, or do you want to be able to control certain warnings separately?
- 2** If the latter is true, you will need to identify those warnings you want to selectively control. This requires going through your code and attaching unique *message identifiers* to each of those warnings. If, on the other hand,

you do not require that fine a granularity of control, the warning statements in your code need no message identifiers.

- 3 When you are ready to run your programs, use the MATLAB warning control statements to exercise the desired controls on all or selected warnings. Include message identifiers in these control statements when selecting specific warnings to act upon.

Warning Statements

The warning statements you put into your code must contain the string to be displayed when the warning is incurred, and may also contain a message identifier. If you are not planning to use warning control or if you do not need to single out certain warnings for control, you need to specify only the message string. Use the syntax shown in “Warnings” on page 18-23. Valid formats are

```
warning('warnmsg')  
warning('formatted_warnmsg', arg1, arg2, ...)
```

Attaching an Identifier to the Warning Statement

If you want to be able to apply control statements to specific warnings, you need to include a message identifier in the warning statements you wish to control. The message identifier must be the first argument in the statement. Valid formats are

```
warning('msg_id', 'warnmsg')  
warning('msg_id', 'formatted_warnmsg', arg1, arg2, ...)
```

See “Message Identifiers” on page 18-8 for information on how to specify the `msg_id` argument.

Note When you specify more than one input argument with `warning`, MATLAB treats the `warnmsg` string as if it were a `formatted_warnmsg`. This is explained in “Formatted Message Strings” on page 18-23.

Warning Control Statements

Once you have the warning statements in your program file and are ready to execute it, you tell MATLAB how to act on these warnings by issuing control statements. These statements place the specified warning(s) into a desired state and have the format

```
warning state msg_id
```

Control statements can return information on the state of selected warnings if you assign the output to a variable, as shown below. See “Output from Control Statements” on page 18-30.

```
s = warning('state', 'msg_id');
```

Warning States

There are three possible values for the `state` argument of a warning control statement.

State	Description
on	Enable the display of selected warning message.
off	Disable the display of selected warning message.
query	Display the current state of selected warning.

Message Identifiers

In addition to the message identifiers already discussed, there are three other identifiers that you can use in control statements only.

Identifier	Description
<i>msg_id string</i>	Set selected warning to the specified state.
all	Set all warnings to the specified state.
last	Set only the last displayed warning to the specified state.

Note MATLAB starts up with all warnings enabled, except for those displayed in response to the command, `warning('query', 'all')`.

When warnings are disabled, the `dbstop if warning` commands have no effect. If warnings are disabled for specific message identifiers, the `dbstop if warning` identifier has no effect for those identifiers.

Retrieving a Message Identifier from a Warning. If you get a warning and you would like to know what the message identifier is for that warning, you can retrieve the identifier from the second output of the `lastwarn` function. The following example generates a warning when it attempts to concatenate two unlike integer types together:

```
warning on all;

A = [int8(150), int16(300)];
Warning: Concatenation with dominant (left-most) integer class
        may overflow other operands on conversion to return class.
```

If you are already aware of the consequences of this command and do not want to see this warning message displayed every time you run your program, you can disable the warning message. To identify the warning to disable, use the following commands to acquire the message identifier:

```
warnStruct = warning('query', 'last');
msgid_integerCat = warnStruct.identifier
msgid_integerCat =
    MATLAB:concatenation:integerInteraction
```

Once you have the identifier, you can use it to disable this one particular message:

```
warning('off', msgid_integerCat);
```

Try the command again

```
A = [int8(150), int16(300)]
A =
    127    127
```

Turn the message back on again, if you need to, as shown here:

```
warning('on', msgid_intcatwarn);  
MATLAB:nonScalarConditional
```

Enabling and Disabling Selected Warnings. Enable just the `actionNotTaken` warning from Simulink by first turning off all warnings and then setting just that warning to on.

```
warning off all  
warning on Simulink:actionNotTaken
```

Next, use `query` to determine the current state of all warnings. It reports that you have set all warnings to off, with the exception of `Simulink:actionNotTaken`.

```
warning query all  
The default warning state is 'off'. Warnings not set to the  
default are
```

```
State Warning Identifier  
  
on Simulink:actionNotTaken
```

Enabling and Disabling All Warnings. You can enable or disable all warnings using the `'all'` identifier. Using the `query` option shows the result:

```
warning on all
```

```
warning query all  
The default warning state is 'on'. Warnings not set to the  
default are
```

```
State Warning Identifier  
  
off MATLAB:nonScalarConditional
```

Note that, in this case, there is one warning that does not take on the default state when that state is changed from off to on. This is intentional. If so desired, you can force this or any individual warning to either the on or off state by specifying the message identifier in the command:

```
warning on MATLAB:nonScalarConditional
```

```
warning query all
```

```
All warnings have the state 'on'.
```

Disabling the Most Recent Warning. Evaluating `inv` on zero displays a warning message. Turn off the most recently invoked warning with `warning off last`.

```
inv(0)
```

```
Warning: Matrix is singular to working precision.
```

```
ans =
```

```
Inf
```

```
warning off last
```

```
inv(0)
```

```
% No warning is displayed this time
```

```
ans =
```

```
Inf
```

Output from Control Statements

The `warning` function, when used in a control statement, returns a MATLAB structure array containing the previous state of the selected warning(s). Use the following syntax to return this information in structure array `s`:

```
s = warning('state', 'msg_id');
```

You must type the command using the MATLAB function format; parentheses and quotation marks are required.

Note MATLAB does not display warning output if you do not assign the output to a variable.

The next example turns off `InconsistentDataType` warnings for the `MATLAB:odearguments` component, and returns the identifier and previous state in a 1-by-1 structure array.

```
MATLAB:odearguments:InconsistentDataType
```

```
s = warning('off','MATLAB:odearguments:InconsistentDataType')
s =
    identifier: 'MATLAB:odearguments:InconsistentDataType'
    state: 'on'
```

You can use output variables with any type of warning control statement. If you just want to collect the information but do not want to change state, simply perform a query on the warning(s). MATLAB returns the current state of those warnings selected by the message identifier.

```
s = warning('query', 'msg_id');
```

If you want to change state, but save the former state so you can restore it later, use the return structure array to save that state. The following example does an implicit query, returning state information in `s`, and then turns on all warnings.

```
s = warning('on', 'all');
```

See “Saving and Restoring State” on page 18-32, for more information on restoring the former state of warnings.

Output Structure Array

Each element of the structure array returned by `warning` contains two fields.

Field Name	Description
<code>identifier</code>	Message identifier string, 'all', or 'last'
<code>state</code>	State of warning(s) prior to invoking this control statement

If you query for the state of just one warning, using a message identifier or 'last' in the command, MATLAB returns a one-element structure array. The `identifier` field contains the selected message identifier, and the `state` field holds the current state of that warning:

```
s = warning('query','last')
s =
    identifier: 'MATLAB:odearguments:InconsistentDataType'
    state: 'on'
```

If you query for the state of all warnings, using 'all' in the command, MATLAB returns a structure array having one or more elements:

- The first element of the array always represents the default state. (This is the state set by the last warning `on|off all` command.)
- Each other element of the array represents a warning that is in a state different from the default.

```
warning off all
warning on MATLAB:odearguments:InconsistentDataType
warning on MATLAB:rmpath:DirNotFound

s = warning('query', 'all')
s =
    3x1 struct array with fields:
        identifier
        state

s(1)
ans =
    identifier: 'all'
    state: 'off'

s(2)
ans =
    identifier: 'MATLAB:odearguments:InconsistentDataType'
    state: 'on'

s(3)
ans =
    identifier: 'MATLAB:rmpath:DirNotFound'
    state: 'on'
```

Saving and Restoring State

To temporarily change the state of some warnings and then later return to your original settings, save the original state in a structure array and then restore it from that array. You can save and restore the state of all of your warnings or just one that you select with a message identifier.

To save the current warning state, assign the output of a warning control statement, as discussed in “Output from Control Statements” on page 18-30. The following statement saves the current state of all warnings in structure array `s`:

```
s = warning('query', 'all');
```

To restore state from `s`, use the syntax shown below. Note that the MATLAB function format (enclosing arguments in parentheses) is required.

```
warning(s)
```

Example 1 – Performing an Explicit Query

Perform a query of all warnings to save the current state in structure array `s`:

```
s = warning('query', 'all');
```

Then, after doing some work that includes making changes to the state of some warnings, restore the original state of all warnings:

```
warning(s)
```

Example 2 – Performing an Implicit Query

Turn on one particular warning, saving the previous state of this warning in `s`. Remember that this nonquery syntax (where `state` equals `on` or `off`) performs an implicit query prior to setting the new state:

```
s = warning('on', 'Control:parameterNotSymmetric');
```

Restore the state of that one warning when you are ready, with

```
warning(s)
```

Backtrace and Verbose Modes

In addition to warning messages, there are two *modes* that can be enabled or disabled with a warning control statement. These modes are shown here.

Mode	Description	Default
verbose	Display a message on how to suppress the warning.	off (terse)
backtrace	Display a stack trace after a warning is invoked.	on (enabled)

The syntax for this type of control statement is as follows, where *state*, in this case, can be only on, off, or query:

```
warning state mode
```

Note that there is no need to include a message identifier with this type of control statement. All enabled warnings are affected by the this type of control statement.

Note You cannot save and restore the current state of the `backtrace` or `verbose` modes as you can with other states.

Example 1 – Enabling Verbose Warnings

When you enable verbose warnings, MATLAB displays an extra line of information with each warning that tells you how to suppress it:

Turn on all warnings, disable backtrace (if you have just run the previous example), and enable verbose warnings:

```
warning on all
warning off backtrace
warning on verbose
```

Create a function that tests a condition and displays a warning message based on the input:

```
function testArrayMax(arr, max)
exceedMax = find(arr > max);
if any(exceedMax)
    warning('TestEnv:InvalidInput', ...
        'Values in array "%s" exceed the maximum.', ...
```

```
        inputname(1))
end
```

Call the function to find out how to suppress warnings that might be generated by that function. Note the last line displayed here:

```
A = [1287, 5010, 2759];

testArrayMax(A, 5000)
Warning: Values in array "A" exceed the maximum.
(Type "warning off TestEnv:InvalidInput" to suppress this warning.)
```

Use the message identifier `TestEnv:InvalidInput` to disable only this warning, and run the function again. This time the warning message is not displayed:

```
warning off TestEnv:InvalidInput
testArrayMax(A, 5000)
```

Example 2 – Displaying a Stack Trace on a Specific Warning

It can be difficult to locate the source of a warning when it is generated from code buried in several levels of function calls. This example generates a warning within a function that is nested several levels deep within the primary function in file `isValidArray.m`:

```
function isValidArray(A)
max = 5000;
nestFun_1
    function nestFun_1
        nestFun_2
            function nestFun_2
                testArrayMax(A, max);
            end
        end
    end
end
```

After enabling all warnings, run the program. Due to the value of `A(2)`, the function generates a warning:

```
warning on all  
warning off verbose
```

```
A = [1287, 5010, 2759];
```

```
isValidArray(A)  
Warning: Values in array "A" exceed the maximum.
```

In a function of this size, it is not difficult to find the cause of the warning, but in a file of several hundred lines, this could take some time. To simplify the debug process, enable backtrace mode. In this mode, MATLAB reports which function generated the warning (`testArrayMax`), the line number of the attempted operation (line 4), the sequence of function calls that led up to the execution of the function (from `isValidArray` to `nestFun_1` to `nestFun_2` and finally to `testArrayMax`), and the line at which each of these function calls were made (3, 5, 7, and 4):

```
warning on backtrace
```

```
callArrayMax(A)  
Warning: Values in array "A" exceed the maximum.  
> In testArrayMax at 4  
   In isValidArray>nestFun_1/nestFun_2 at 7  
   In isValidArray>nestFun_1 at 5  
   In isValidArray at 3
```

Debugging Errors and Warnings

You can direct the MATLAB software to temporarily stop the execution of an program in the event of a run-time error or warning, at the same time opening a debug window paused at the line that generated the error or warning. This enables you to examine values internal to the program and determine the cause of the error.

Use the `dbstop` function to have MATLAB stop execution and enter debug mode when any function you subsequently run produces a run-time error or warning. There are three types of such breakpoints that you can set.

Command	Description
<code>dbstop if all error</code>	Stop on any error.
<code>dbstop if error</code>	Stop on any error not detected within a try-catch block.
<code>dbstop if warning</code>	Stop on any warning.

In all three cases, the file you are trying to debug must be in a folder that is on the search path or in the current folder.

You cannot resume execution after an error; use `dbquit` to exit from the Debugger. To resume execution after a warning, use `dbcont` or `dbstep`.

Program Scheduling

- “Using a MATLAB Timer Object” on page 19-2
- “Creating Timer Objects” on page 19-5
- “Working with Timer Object Properties” on page 19-7
- “Starting and Stopping Timers” on page 19-10
- “Creating and Executing Callback Functions” on page 19-14
- “Timer Object Execution Modes” on page 19-19
- “Deleting Timer Objects from Memory” on page 19-23
- “Finding Timer Objects in Memory” on page 19-24

Using a MATLAB Timer Object

In this section...
“Overview” on page 19-2
“Example: Displaying a Message” on page 19-3

Overview

The MATLAB software includes a timer object that you can use to schedule the execution of MATLAB commands. This section describes how you can create timer objects, start a timer running, and specify the processing that you want performed when a timer fires. A timer is said to *fire* when the amount of time specified by the timer object elapses and the timer object executes the commands you specify.

To use a timer, perform these steps:

- 1 Create a timer object.

You use the `timer` function to create a timer object. See “Creating Timer Objects” on page 19-5 for more information.

- 2 Specify which MATLAB commands you want executed when the timer fires and control other aspects of timer object behavior.

You use timer object properties to specify this information. To learn about all the properties supported by the timer object, see “Working with Timer Object Properties” on page 19-7. (You can also set timer object properties when you create them, in step 1.)

- 3 Start the timer object.

After you create the timer object, you must start it, using either the `start` or `startat` function. See “Starting and Stopping Timers” on page 19-10 for more information.

- 4 Delete the timer object when you are done with it.

After you are finished using a timer object, you should delete it from memory. See “Deleting Timer Objects from Memory” on page 19-23 for more information.

Note The specified execution time and the actual execution of a timer can vary because timer objects work in the MATLAB single-threaded execution environment. The length of this time lag is dependent on what other processing MATLAB is performing. To force the execution of the callback functions in the event queue, include a call to the `drawnow` function in your code. The `drawnow` function flushes the event queue.

Example: Displaying a Message

The following example sets up a timer object that executes a MATLAB command string after 10 seconds elapse. The example creates a timer object, specifying the values of two timer object properties, `TimerFcn` and `StartDelay`. `TimerFcn` specifies the timer callback function. This is the MATLAB command string or program file that you want to execute when the timer fires. In the example, the timer callback function sets the value of the MATLAB workspace variable `stat` and executes the MATLAB `disp` command. The `StartDelay` property specifies how much time elapses before the timer fires.

After creating the timer object, the example uses the `start` function to start the timer object. (The additional commands in this example are included to illustrate the timer but are not required for timer operation.)

```
t = timer('TimerFcn', 'stat=false; disp(''Timer!'')',...
         'StartDelay',10);
start(t)

stat=true;
while(stat==true)
    disp('.')
    pause(1)
end
```

When you execute this code, it produces this output:

Creating Timer Objects

In this section...

“Creating the Object” on page 19-5

“Naming the Object” on page 19-6

Creating the Object

To use a timer in MATLAB, you must create a timer object. The timer object represents the timer in MATLAB, supporting various properties and functions that control its behavior.

To create a timer object, use the `timer` function. This creates a valid timer object with default values for most properties. The following shows an example of the default timer object and its summary display:

```
t = timer
Timer Object: timer-1

Timer Settings
  ExecutionMode: singleShot
        Period: 1
  BusyMode: drop
        Running: off

Callbacks
  TimerFcn: ''
  ErrorFcn: ''
  StartFcn: ''
  StopFcn: ''
```

MATLAB names the timer object `timer-1`. (See “Naming the Object” on page 19-6 for more information.)

To specify the value of timer object properties after you create it, you can use the `set` function. This example sets the value of the `TimerFcn` property and the `StartDelay` property. For more information about timer object properties, see “Working with Timer Object Properties” on page 19-7.

```
set(t, 'TimerFcn', @(x,y)disp('Hello World!'), 'StartDelay', 5)
```

You can also set timer object properties when you create the timer object by specifying property name and value pairs as arguments to the `timer` function. The following example sets the same properties at object creation time:

```
t = timer('TimerFcn', @(x,y)disp('Hello World!'), 'StartDelay', 5);
```

Always delete timer objects when you are done using them. See “Deleting Timer Objects from Memory” on page 19-23 for more information.

Naming the Object

MATLAB assigns a name to each timer object you create. This name has the form `timer-i`, where *i* is a number representing the total number of timer objects created this session.

For example, the first time you call the `timer` function to create a timer object, MATLAB names the object `timer-1`. If you call the `timer` function again to create another timer object, MATLAB names the object `timer-2`.

MATLAB keeps incrementing the number associated with each timer object it creates, even if you delete the timer objects you already created. For example, if you delete the first two timer objects and create a new object, MATLAB names it `timer-3`, even though the other two timer objects no longer exist in memory. To reset the numeric part of timer object names to 1, execute the `clear classes` command.

Working with Timer Object Properties

In this section...

“Retrieving the Value of Timer Object Properties” on page 19-7

“Setting the Value of Timer Object Properties” on page 19-8

To get information about timer object properties, see the `timer` function reference page.

Retrieving the Value of Timer Object Properties

The timer object supports many properties that provide information about the current state of the timer object and control aspects of its functioning. To retrieve the value of a timer object property, you can use the `get` function or use subscripts (dot notation) to access the field.

The following example uses the `set` function to retrieve the value of the `ExecutionMode` property:

```
t = timer;

tmode = get(t, 'ExecutionMode')

tmode =

singleShot
```

The following example uses dot notation to retrieve the value of the `ExecutionMode` property:

```
tmode = t.ExecutionMode

tmode =

singleShot
```

To view a list of all the properties of a timer object, use the `get` function, specifying the timer object as the only argument:

```
get(t)
    AveragePeriod: NaN
    BusyMode: 'drop'
    ErrorFcn: ''
    ExecutionMode: 'singleShot'
    InstantPeriod: NaN
    Name: 'timer-4'
ObjectVisibility: 'on'
    Period: 1
    Running: 'off'
    StartDelay: 0
    StartFcn: ''
    StopFcn: ''
    Tag: ''
    TasksExecuted: 0
    TasksToExecute: Inf
    TimerFcn: ''
    Type: 'timer'
    UserData: []
```

Setting the Value of Timer Object Properties

To set the value of a timer object property, use the `set` function or subscripted assignment (dot notation). You can also set timer object properties when you create the timer object. For more information, see “Creating Timer Objects” on page 19-5.

The following example uses both methods to assign values to timer object properties. The example creates a timer that, once started, displays a message every second until you stop it with the `stop` command.

- 1 Create a timer object.

```
t = timer;
```

- 2 Assign values to timer object properties using the `set` function.

```
set(t,'ExecutionMode','fixedRate','BusyMode','drop','Period',1);
```

- 3 Assign a value to the timer object `TimerFcn` property using dot notation.

```
t.TimerFcn = @(x,y)disp('Processing...');
```

- 4** Start the timer object. It displays a message at 1-second intervals.

```
start(t)
```

- 5** Stop the timer object.

```
stop(t)
```

- 6** Delete timer objects after you are done using them.

```
delete(t)
```

Viewing a List of All Settable Properties

To view a list of all timer object properties that can have values assigned to them (in contrast to the read-only properties), use the `set` function, specifying the timer object as the only argument.

The display includes the values you can use to set the property if, like the `BusyMode` property, the property accepts an enumerated list of values.

```
t = timer;

set(t)
  BusyMode: [ {drop} | queue | error ]
  ErrorFcn: string -or- function handle -or- cell array
  ExecutionMode: [{singleShot} | fixedSpacing | fixedDelay | fixedRate]
  Name
  ObjectVisibility: [ {on} | off ]
  Period
  StartDelay
  StartFcn: string -or- function handle -or- cell array
  StopFcn: string -or- function handle -or- cell array
  Tag
  TasksToExecute
  TimerFcn: string -or- function handle -or- cell array
  UserData
```

Starting and Stopping Timers

In this section...

“Starting a Timer” on page 19-10

“Starting a Timer at a Specified Time” on page 19-10

“Stopping Timer Objects” on page 19-11

“Blocking the MATLAB Command Line” on page 19-12

Note Because the timer works within the MATLAB single-threaded environment, it cannot guarantee execution times or execution rates.

Starting a Timer

To start a timer object, call the `start` function, specifying the timer object as the only argument. The `start` function starts a timer object running; the amount of time the timer runs is specified in seconds in the `StartDelay` property.

This example creates a timer object that displays a greeting after 5 seconds elapse.

- 1 Create a timer object, specifying values for timer object properties.

```
t = timer('TimerFcn',@(x,y)disp('Hello World!'),'StartDelay', 5);
```

- 2 Start the timer object.

```
start(t)
```

- 3 Delete the timer object after you are finished using it.

```
delete(t);
```

Starting a Timer at a Specified Time

To start a timer object and specify a date and time for the timer to fire, (rather than specifying the number of seconds to elapse), use the `startat` function. This function starts a timer object and allows you to specify the date, hour,

minute, and second when you want to the timer to execute. You specify the time as a MATLAB serial date number or as a specially formatted date text string.

This example creates a timer object that displays a message after an hour has elapsed. The `startat` function starts the timer object running and calculates the value of the `StartDelay` property based on the time you specify.

```
t2=timer('TimerFcn',@(x,y)disp('It has been an hour now'));
startat(t2,now+1/24);
```

Stopping Timer Objects

Once started, the timer object stops running if one of the following conditions apply:

- The timer function callback (`TimerFcn`) has been executed the number of times specified in the `TasksToExecute` property.
- An error occurred while executing a timer function callback (`TimerFcn`).

You can also stop a timer object by using the `stop` function, specifying the timer object as the only argument. The following example illustrates stopping a timer object:

- 1 Create a timer object.

```
t = timer('TimerFcn',@(x,y)disp('Hello World!'), ...
          'StartDelay', 100);
```

- 2 Start it running.

```
start(t)
```

- 3 Check the state of the timer object after starting it.

```
get(t,'Running')
```

```
ans =
```

```
on
```

- 4 Stop the timer using the `stop` command and check the state again. When a timer stops, the value of the `Running` property of the timer object is set to `'off'`.

```
stop(t)

get(t, 'Running')

ans =

off
```

- 5 Delete the timer object when you are finished using it.

```
delete(t)
```

Note The timer object can execute a callback function that you specify when it starts or stops. See “Creating and Executing Callback Functions” on page 19-14.

Blocking the MATLAB Command Line

By default, when you use the `start` or `startat` function to start a timer object, the function returns control to the command line immediately. For some applications, you might prefer to block the command line until the timer fires. To do this, call the `wait` function right after calling the `start` or `startat` function.

- 1 Create a timer object.

```
t = timer('StartDelay', 5, 'TimerFcn', ...
         @(x,y)disp('Hello World!'));
```

- 2 Start the timer object running.

```
start(t)
```

- 3** After the `start` function returns, call the `wait` function immediately. The `wait` function blocks the command line until the timer object fires.

```
wait(t)
```

- 4** Delete the timer object after you are finished using it.

```
delete(t)
```

Creating and Executing Callback Functions

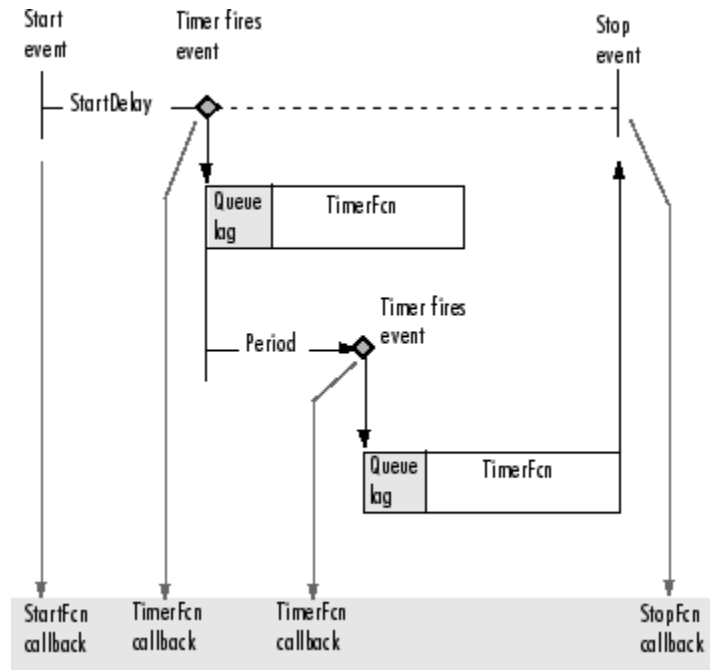
In this section...
“Associating Commands with Timer Object Events” on page 19-14
“Creating Callback Functions” on page 19-15
“Specifying the Value of Callback Function Properties” on page 19-17

Note Callback function execution might be delayed if the callback involves a CPU-intensive task such as updating a figure.

Associating Commands with Timer Object Events

The timer object supports properties that let you specify the MATLAB commands that execute when a timer fires, and for other timer object events, such as starting, stopping, or when an error occurs. These are called *callbacks*. To associate MATLAB commands with a timer object event, set the value of the associated timer object callback property.

The following diagram shows when the events occur during execution of a timer object and give the names of the timer object properties associated with each event. For example, to associate MATLAB commands with a start event, assign a value to the `StartFcn` callback property. Error callbacks can occur at any time.



Timer Object Events and Related Callback Function

Creating Callback Functions

When the time period specified by a timer object elapses, the timer object executes one or more MATLAB functions of your choosing. You can specify the functions directly as the value of the callback property. You can also put the commands in a function file and specify the function as the value of the callback property.

Specifying Callback Functions Directly

This example creates a timer object that displays a greeting after 5 seconds. The example specifies the value of the `TimerFcn` callback property directly, putting the commands in a text string.

```
t = timer('TimerFcn',@(x,y)disp('Hello World!'),'StartDelay',5);
```

Note When you specify the callback commands directly as the value of the callback function property, the commands are evaluated in the MATLAB workspace.

Putting Commands in a Callback Function

Instead of specifying MATLAB commands directly as the value of a callback property, you can put the commands in a MATLAB program file and specify the file as the value of the callback property.

When you create a callback function, the first two arguments must be a handle to the timer object and an event structure. An event structure contains two fields: `Type` and `Data`. The `Type` field contains a text string that identifies the type of event that caused the callback. The value of this field can be any of the following strings: `'StartFcn'`, `'StopFcn'`, `'TimerFcn'`, or `'ErrorFcn'`. The `Data` field contains the time the event occurred.

In addition to these two required input arguments, your callback function can accept application-specific arguments. To receive these input arguments, you must use a cell array when specifying the name of the function as the value of a callback property. For more information, see “Specifying the Value of Callback Function Properties” on page 19-17.

Example: Writing a Callback Function

This example implements a simple callback function that displays the type of event that triggered the callback and the time the callback occurred. To illustrate passing application-specific arguments, the example callback function accepts as an additional argument a text string and includes this text string in the display output. To see this function used with a callback property, see “Specifying the Value of Callback Function Properties” on page 19-17.

```
function my_callback_fcn(obj, event, string_arg)

txt1 = ' event occurred at ';
txt2 = string_arg;

event_type = event.Type;
```

```

event_time = datestr(event.Data.time);

msg = [event_type txt1 event_time];
disp(msg)
disp(txt2)

```

Specifying the Value of Callback Function Properties

You associate a callback function with a specific event by setting the value of the appropriate callback property. You can specify the callback function as a cell array or function handle. If your callback function accepts additional arguments, you must use a cell array.

The following table shows the syntax for several sample callback functions and describes how you call them.

Callback Function Syntax	How to Specify as a Property Value for Object t
<code>function myfile</code>	<code>t.StartFcn = @myfile</code>
<code>function myfile(obj, event)</code>	<code>t.StartFcn = @myfile</code>
<code>function myfile(obj, event, arg1, arg2)</code>	<code>t.StartFcn = {@myfile, 5, 6}</code>

This example illustrates several ways you can specify the value of timer object callback function properties, some with arguments and some without. To see the code of the callback function, `my_callback_fcn`, see “Example: Writing a Callback Function” on page 19-16:

- 1 Create a timer object.

```

t = timer('StartDelay', 4, 'Period', 4, 'TasksToExecute', 2, ...
         'ExecutionMode', 'fixedRate');

```

- 2 Specify the value of the `StartFcn` callback. Note that the example specifies the value in a cell array because the callback function needs to access arguments passed to it:

```

t.StartFcn = {@my_callback_fcn, 'My start message'};

```

- 3 Specify the value of the `StopFcn` callback. Again, the value is specified in a cell array because the callback function needs to access the arguments passed to it:

```
t.StopFcn = { @my_callback_fcn, 'My stop message' };
```

- 4 Specify the value of the `TimerFcn` callback. The example specifies the MATLAB commands in a text string:

```
t.TimerFcn = @(x,y)disp('Hello World!');
```

- 5 Start the timer object:

```
start(t)
```

The example outputs the following.

```
StartFcn event occurred at 10-Mar-2004 17:16:59  
My start message  
Hello World!  
Hello World!  
StopFcn event occurred at 10-Mar-2004 17:16:59  
My stop message
```

- 6 Delete the timer object after you are finished with it.

```
delete(t)
```


Timer Object Execution Modes

In this section...

“Executing a Timer Callback Function Once” on page 19-19

“Executing a Timer Callback Function Multiple Times” on page 19-20

“Handling Callback Function Queuing Conflicts” on page 19-21

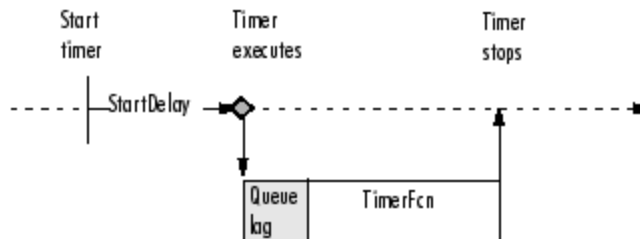
Executing a Timer Callback Function Once

The timer object supports several execution modes that determine how it schedules the timer callback function (`TimerFcn`) for execution. You specify the execution mode by setting the value of the `ExecutionMode` property.

To execute a timer callback function once, set the `ExecutionMode` property to `'singleShot'`. This is the default execution mode. In this mode, the timer object starts the timer and, after the time period specified in the `StartDelay` property elapses, adds the timer callback function (`TimerFcn`) to the MATLAB execution queue. When the timer callback function finishes, the timer stops.

The following figure graphically illustrates the parts of timer callback execution for a `singleShot` execution mode. The shaded area in the figure, labelled `queue lag`, represents the indeterminate amount of time between when the timer adds a timer callback function to the MATLAB execution queue and when the function starts executing. The duration of this lag is dependent on what other processing MATLAB happens to be doing at the time.

`singleShot`



Timer Callback Execution (`singleShot` Execution Mode)

Executing a Timer Callback Function Multiple Times

The timer object supports three multiple-execution modes:

- 'fixedRate'
- 'fixedDelay'
- 'fixedSpacing'

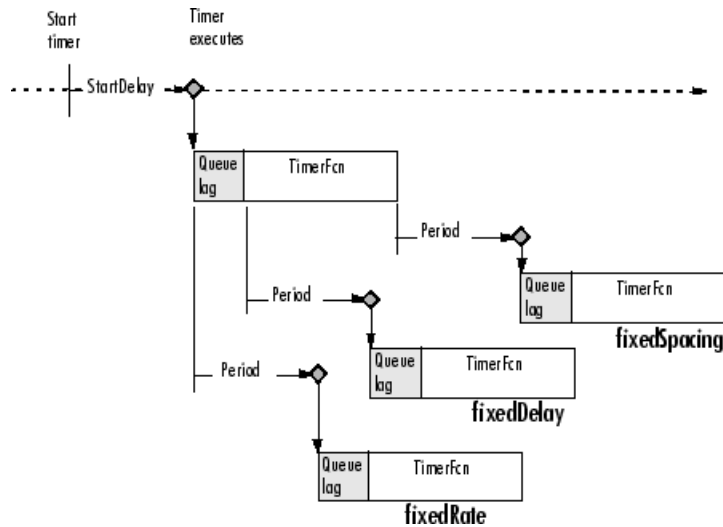
In many ways, these execution modes operate the same:

- The `TasksToExecute` property specifies the number of times you want the timer to execute the timer callback function (`TimerFcn`).
- The `Period` property specifies the amount of time between executions of the timer callback function.
- The `BusyMode` property specifies how the timer object handles queuing of the timer callback function when the previous execution of the callback function has not completed. See “Handling Callback Function Queuing Conflicts” on page 19-21 for more information.

The execution modes differ only in where they start measuring the time period between executions. The following table describes these differences.

Execution Mode	Description
'fixedRate'	Time period between executions begins immediately after the timer callback function is added to the MATLAB execution queue.
'fixedDelay'	Time period between executions begins when the timer function callback actually starts executing, after any time lag due to delays in the MATLAB execution queue.
'fixedSpacing'	Time period between executions begins when the timer callback function finishes executing.

The following figure illustrates the difference between these modes. Note that the amount of time between executions (specified by the `Period` property) remains the same. Only the point at which execution begins is different.



Differences Between Execution Modes

Handling Callback Function Queuing Conflicts

At busy times, in multiple-execution scenarios, the timer may need to add the timer callback function (`TimerFcn`) to the MATLAB execution queue before the previously queued execution of the callback function has completed. You can determine how the timer object handles this scenario by using the `BusyMode` property.

If you specify `'drop'` as the value of the `BusyMode` property, the timer object skips the execution of the timer function callback if the previously scheduled callback function has not already completed.

If you specify `'queue'`, the timer object waits until the currently executing callback function finishes before queuing the next execution of the timer callback function.

Note In 'queue' mode, the timer object tries to make the average time between executions equal the amount of time specified in the `Period` property. If the timer object has to wait longer than the time specified in the `Period` property between executions of the timer function callback, it shortens the time period for subsequent executions to make up the time.

If the `BusyMode` property is set to 'error', the timer object stops and executes the timer object error callback function (`ErrorFcn`), if one is specified.

Deleting Timer Objects from Memory

In this section...

“Deleting One or More Timer Objects” on page 19-23

“Testing the Validity of a Timer Object” on page 19-23

Deleting One or More Timer Objects

When you are finished with a timer object, delete it from memory using the `delete` function:

```
delete(t)
```

When you delete a timer object, workspace variables that referenced the object remain. Deleted timer objects are invalid and cannot be reused. Use the `clear` command to remove workspace variables that reference deleted timer objects.

To remove all timer objects from memory, enter

```
delete(timerfind)
```

For information about the `timerfind` function, see “Finding Timer Objects in Memory” on page 19-24.

Testing the Validity of a Timer Object

To test if a timer object has been deleted, use the `isvalid` function. The `isvalid` function returns logical 0 (false) for deleted timer objects:

```
isvalid(t)  
ans =
```

```
0
```

Finding Timer Objects in Memory

In this section...

“Finding All Timer Objects” on page 19-24

“Finding Invisible Timer Objects” on page 19-24

Finding All Timer Objects

To find all the timer objects that exist in memory, use the `timerfind` function. This function returns an array of timer objects. If you leave off the semicolon, and there are multiple timer objects in the array, `timerfind` displays summary information in a table:

```
t1 = timer;
t2 = timer;
t3 = timer;
t_array = timerfind
```

Timer Object Array

Index:	ExecutionMode:	Period:	TimerFcn:	Name:
1	singleShot	1	''	timer-3
2	singleShot	1	''	timer-4
3	singleShot	1	''	timer-5

Using `timerfind` to determine all the timer objects that exist in memory can be helpful when deleting timer objects.

Finding Invisible Timer Objects

If you set the value of a timer object's `ObjectVisibility` property to 'off', the timer object does not appear in listings of existing timer objects returned by `timerfind`. The `ObjectVisibility` property provides a way for application developers to prevent end-user access to the timer objects created by their application.

Objects that are not visible are still valid. If you have access to the object (for example, from within the file that created it), you can set its properties. To

retrieve a list of all the timer objects in memory, including invisible ones, use the `timerfindall` function.

Performance

- “Analyzing Your Program’s Performance” on page 20-2
- “Techniques for Improving Performance” on page 20-4

Analyzing Your Program's Performance

In this section...
“Overview” on page 20-2
“The Profiler Utility” on page 20-2
“Stopwatch Timer Functions” on page 20-2

Overview

The MATLAB Profiler graphical user interface and the stopwatch timer functions enable you to get back information on how your program is performing and help you identify areas that need improvement. The Profiler can be more useful in measuring relative execution time and in identifying specific performance bottlenecks in your code, while the stopwatch functions tend to be more useful for providing absolute time measurements.

The Profiler Utility

A good first step to speeding up your programs is to find out where the bottlenecks are. This is where you need to concentrate your attention to optimize your code.

The MATLAB software provides the MATLAB Profiler, a graphical user interface that shows you where your program is spending its time during execution. Use the Profiler to help you determine where you can modify your code to make performance improvements.

To start the Profiler, type `profile viewer` or select **Desktop > Profiler** in the MATLAB Command Window. See *Profiling for Improving Performance* in the MATLAB Desktop Tools and Development Environment documentation, and the `profile` function reference page.

Stopwatch Timer Functions

If you just need to get an idea of how long your program (or a portion of it) takes to run, or to compare the speed of different implementations of a program, you can use the stopwatch timer functions, `tic` and `toc`. Invoking

`tic` starts the timer, and the first subsequent `toc` stops it and reports the time elapsed between the two.

Use `tic` and `toc` as shown here:

```
tic
-- run the program section to be timed --
toc
```

Keep in mind that `tic` and `toc` measure overall elapsed time. Make sure that no other applications are running in the background on your system that could affect the timing of your MATLAB programs.

Measuring Smaller Programs

Shorter programs sometimes run too fast to get useful data from `tic` and `toc`. When this is the case, try measuring the program running repeatedly in a loop, and then average to find the time for a single run:

```
tic
  for k = 1:100
    -- run the program --
  end
toc
```

Using `tic` and `toc` Versus the `cputime` Function

Although it is possible to measure performance using the `cputime` function, it is recommended that you use the `tic` and `toc` functions for this purpose exclusively. It has been the general rule for CPU-intensive calculations run on Microsoft Windows machines that the elapsed time using `cputime` and the elapsed time using `tic` and `toc` are close in value, ignoring any first time costs. There are cases however that show a significant difference between these two methods. For example, in the case of a Pentium 4 with hyperthreading running Windows, there can be a significant difference between the values returned by `cputime` versus `tic` and `toc`.

Techniques for Improving Performance

In this section...

“Preallocating Arrays” on page 20-4

“Limiting Size and Complexity” on page 20-5

“Assigning to Variables” on page 20-6

“Using Appropriate Logical Operators” on page 20-7

“Overloading Built-In Functions” on page 20-7

“Functions Are Generally Faster Than Scripts” on page 20-8

“Load and Save Are Faster Than File I/O Functions” on page 20-8

“Vectorizing Loops” on page 20-8

“Avoid Large Background Processes” on page 20-11

Preallocating Arrays

for and while loops that incrementally increase, or *grow*, the size of a data structure each time through the loop can adversely affect performance and memory use. Repeatedly resizing arrays often requires that MATLAB spend extra time looking for larger contiguous blocks of memory and then moving the array into those blocks. You can often improve on code execution time by preallocating the maximum amount of space that would be required for the array ahead of time.

The following code creates a scalar variable `x`, and then gradually increases the size of `x` in a for loop instead of preallocating the required amount of memory at the start:

```
x = 0;
for k = 2:1000
    x(k) = x(k-1) + 5;
end
```

Change the first line to preallocate a 1-by-1000 block of memory for `x` initialized to zero. This time there is no need to repeatedly reallocate memory and move data as more values are assigned to `x` in the loop:

```
x = zeros(1, 1000);
for k = 2:1000
    x(k) = x(k-1) + 5;
end
```

Preallocation Functions

Preallocation makes it unnecessary for MATLAB 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	<code>zeros</code>	<code>y = zeros(1, 100);</code>
Cell	<code>cell</code>	<code>B = cell(2, 3);</code> <code>B{1,3} = 1:3;</code> <code>B{2,2} = 'string';</code>

Preallocating a Nondouble Matrix

When you preallocate a block of memory to hold a matrix of some type other than `double`, avoid using the method

```
A = int8(zeros(100));
```

This statement preallocates a 100-by-100 matrix of `int8` first by creating a full matrix of `doubles`, and then converting each element to `int8`. This costs time and uses memory unnecessarily.

The next statement shows how to do this more efficiently:

```
A = zeros(100, 'int8');
```

Limiting Size and Complexity

Running programs that are unusually large or complex can put a strain on your system's resources. For example, a program that nearly exceeds memory capacity may work some of the time and sometimes not, depending on the commands it uses and on what other applications are running at the time. An example of unnecessary complexity might be having a large number of `if` and `else` statements where `switch` and `case` might be more suitable. This can

also lead to performance and space problems. If you see the following error message displayed, this is likely to be the source of the problem:

```
The input was too complicated or too big for MATLAB to parse
```

If you have a program file that includes thousands of variables or functions, tens of thousands of statements, or hundreds of language keyword pairs (e.g., `if-else`, or `try-catch`), then making some of the changes suggested here is likely to not only boost its performance and reliability, but should make your program code easier to understand and maintain as well.

- Split large script files into smaller ones, having the first file call the second if necessary.
- Take your larger chunks of program code and make separate functions (or subfunctions and nested functions) of them.
- If you have functions or expressions by that seem overly complicated, make smaller and simpler functions or expressions of them. Simpler functions are also more likely to be made into utility functions that you can share with others.

Assigning to Variables

For best performance, keep the following suggestions in mind when assigning values to variables.

Changing a Variable's Data Type or Dimension

Changing the class or array shape of an existing variable slows MATLAB down as it must take extra time to process this. When you need to store data of a different type, it is advisable to create a new variable.

This code changes the type for `X` from `double` to `char`, which has a negative impact on performance:

```
X = 23;
      .
-- other code --
      .
X = 'A';           % X changed from type double to char
      .
```

```
-- other code --
```

Assigning Real and Complex Numbers

Assigning a complex number to a variable that already holds a real number impacts the performance of your program. Similarly, you should not assign a real value to a variable that already holds a complex value.

Using Appropriate Logical Operators

When performing a logical AND or OR operation, you have a choice of two operators of each type.

Operator	Description
&,	Perform logical AND and OR on arrays element by element
&&,	Perform logical AND and OR on scalar values with short-circuiting

In `if` and `while` statements, it is more efficient to use the short-circuiting operators, `&&` for logical AND and `||` for logical OR. This is because these operators often do not have to evaluate the entire logical expression. For example, MATLAB evaluates only the first part of this expression whenever the number of input arguments is less than three:

```
if (nargin >= 3) && (ischar(varargin{3}))
```

See Short-Circuit Operators in the MATLAB documentation for a discussion on short-circuiting with `&&` and `||`.

Overloading Built-In Functions

Overloading MATLAB built-in functions on any of the standard MATLAB data classes can negatively affect performance. For example, if you overload the `plus` function to handle any of the integer classes differently, you may hinder certain optimizations in the MATLAB built-in function code for `plus`, and thus may slow down any programs that make use of this overload.

Functions Are Generally Faster Than Scripts

Your code executes more quickly if it is implemented in a function rather than a script.

Load and Save Are Faster Than File I/O Functions

If you have a choice of whether to use `load` and `save` instead of the low-level MATLAB file I/O routines such as `fread` and `fwrite`, choose the former. `load` and `save` have been optimized to run faster and reduce memory fragmentation.

Vectorizing Loops

The MATLAB software uses a matrix language, which means it is designed for vector and matrix operations. You can often speed up your code by using vectorizing algorithms that take advantage of this design. *Vectorization* means converting `for` and `while` loops to equivalent vector or matrix operations.

Simple Example of Vectorizing

Here is one way to compute the sine of 1001 values ranging from 0 to 10:

```
i = 0;
for t = 0:.01:10
    i = i + 1;
    y(i) = sin(t);
end
```

A vectorized version of the same code is

```
t = 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 scripts that contain the code shown, and then using the `tic` and `toc` functions to measure the performance.

Advanced Example of Vectorizing

`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 array `A`, 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)
```

```
% 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 Create 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,nCols))
```

where `nCols` 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)`, and 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.

Caution While `repmat` can take advantage of vectorization, it can do so at the expense of memory usage. When this is the case, you might find the `bsxfun` function be more appropriate in this respect.

Functions Used in Vectorizing

Some of the most commonly used functions for vectorizing are as follows

Function	Description
<code>all</code>	Test to determine if all elements are nonzero
<code>any</code>	Test for any nonzeros
<code>cumsum</code>	Find cumulative sum

Function	Description
diff	Find differences and approximate derivatives
find	Find indices and values of nonzero elements
ind2sub	Convert from linear index to subscripts
ipermute	Inverse permute dimensions of a multidimensional array
logical	Convert numeric values to logical
meshgrid	Generate X and Y arrays for 3-D plots
ndgrid	Generate arrays for multidimensional functions and interpolation
permute	Rearrange dimensions of a multidimensional array
prod	Find product of array elements
repmat	Replicate and tile an array
reshape	Change the shape of an array
shiftdim	Shift array dimensions
sort	Sort array elements in ascending or descending order
squeeze	Remove singleton dimensions from an array
sub2ind	Convert from subscripts to linear index
sum	Find the sum of array elements

Avoid Large Background Processes

Avoid running large processes in the background at the same time you are executing your program in MATLAB. This frees more CPU time for your MATLAB session.

Memory Usage

- “Memory Allocation” on page 21-2
- “Memory Management Functions” on page 21-12
- “Strategies for Efficient Use of Memory” on page 21-15
- “Resolving “Out of Memory” Errors” on page 21-23

Memory Allocation

In this section...
“Memory Allocation for Arrays” on page 21-2
“Data Structures and Memory” on page 21-6

Memory Allocation for Arrays

The topics below provide information on how the MATLAB software allocates memory when working with arrays and variables. The purpose is to help you use memory more efficiently when writing code. Most of the time, however, you should not need to be concerned with these internal operations as MATLAB handles data storage for you automatically.

- “Creating and Modifying Arrays” on page 21-2
- “Copying Arrays” on page 21-3
- “Array Headers” on page 21-4
- “Function Arguments” on page 21-6

Note Any information on how the MATLAB software handles data internally is subject to change in future releases.

Creating and Modifying Arrays

When you assign a numeric or character array to a variable, MATLAB allocates a contiguous virtual block of memory and stores the array data in that block. MATLAB also stores information about the array data, such as its class and dimensions, in a separate, small block of memory called a *header*.

If you add new elements to an existing array, MATLAB expands the existing array in memory in a way that keeps its storage contiguous. This usually requires finding a new block of memory large enough to hold the expanded array. MATLAB then copies the contents of the array from its original location to this new block in memory, adds the new elements to the array in this block, and frees up the original array location in memory.

If you remove elements from an existing array, MATLAB keeps the memory storage contiguous by removing the deleted elements, and then compacting its storage in the original memory location.

Working with Large Data Sets. If you are working with large data sets, you need to be careful when increasing the size of an array to avoid getting errors caused by insufficient memory. If you expand the array beyond the available contiguous memory of its original location, MATLAB must make a copy of the array and set this copy to the new value. During this operation, there are two copies of the original array in memory. This temporarily doubles the amount of memory required for the array and increases the risk of your program running out of memory during execution. It is better to preallocate sufficient memory for the largest potential size of the array at the start. See “Preallocating Arrays” on page 20-4.

Copying Arrays

Internally, multiple variables can point to the same block of data, thus sharing that array's value. When you copy a variable to another variable (e.g., `B = A`), MATLAB makes a copy of the array reference, but not the array itself. As long as you do not modify the contents of the array, there is no need to store more than one copy of it. If you do modify any elements of the array, MATLAB makes a copy of the array and then modifies that copy.

The following example demonstrates this. Start by creating a simple script `memUsed.m` to display how much memory is currently being used by your MATLAB process. Put these two lines of code in the script:

```
[usr, sys] = memory;  
usr.MemUsedMATLAB
```

Get an initial reading of how much memory is currently being used by your MATLAB process:

```
format short eng;  
memUsed  
ans =  
    295.4977e+006
```

Create a 2000-by-2000 numeric array `A`. This uses about 32MB of memory:

```
A = magic(2000);
memUsed
ans =
    327.6349e+006
```

Make a copy of array A in B. As there is no need at this point to have two copies of the array data, MATLAB only makes a copy of the array reference. This requires no significant additional memory:

```
B = A;
memUsed
ans =
    327.6349e+006
```

Now modify B by making it one half its original size (i.e., set 1000 rows to empty). This requires that MATLAB make a copy of at least the first 1000 rows of the A array, and assign that copy to B:

```
B(1001:2000,:) = [];
format short;    size(B)
ans =
        1000         2000
```

Check the memory used again. Even though B is significantly smaller than it was originally, the amount of memory used by the MATLAB process has increased by about 16 MB (1/2 of the 32 MB originally required for A) because B could no longer remain as just a reference to A:

```
format short eng;    memUsed
ans =
    343.6421e+006
```

Array Headers

When you assign an array to a variable, MATLAB also stores information about the array (such as class and dimensions) in a separate piece of memory called a header. For most arrays, the memory required to store the header is insignificant. There is a small advantage to storing large data sets in a small

number of large arrays as opposed to a large number of small arrays. This is because the former configuration requires fewer array headers.

Structure and Cell Arrays. For structures and cell arrays, MATLAB creates a header not only for each array, but also for each field of the structure and for each cell of a cell array. Because of this, the amount of memory required to store a structure or cell array depends not only on how much data it holds, but also on how it is constructed.

For example, take a scalar structure array **S1** having fields **R**, **G**, and **B**. Each field of size 100-by-50 requires one array header to describe the overall structure, one header for each unique field name, and one header per field for the 1-by-1 structure array. This makes a total of seven array headers for the entire data structure:

```
S1.R(1:100,1:50)
S1.G(1:100,1:50)
S1.B(1:100,1:50)
```

On the other hand, take a 100-by-50 structure array **S2** in which each element has scalar fields **R**, **G**, and **B**. In this case, you need one array header to describe the overall structure, one for each unique field name, and one per field for each of the 5,000 elements of the structure, making a total of 15,004 array headers for the entire data structure:

```
S2(1:100,1:50).R
S2(1:100,1:50).G
S2(1:100,1:50).B
```

Even though **S1** and **S2** contain the same amount of data, **S1** uses significantly less space in memory. Not only is less memory required, but there is a corresponding speed benefit to using the **S1** format, as well.

See “Cell Arrays” and “Structures” under “Data Structures and Memory” on page 21-6.

Memory Usage Reported By the whos Function. The **whos** function displays the amount of memory consumed by any variable. For reasons of simplicity, **whos** reports only the memory used to store the actual data. It does not report storage for the array header, for example.

Function Arguments

MATLAB handles arguments passed in function calls in a similar way. When you pass a variable to a function, you are actually passing a reference to the data that the variable represents. As long as the input data is not modified by the function being called, the variable in the calling function and the variable in the called function point to the same location in memory. If the called function modifies the value of the input data, then MATLAB makes a copy of the original array in a new location in memory, updates that copy with the modified value, and points the input variable in the called function to this new array.

In the example below, function `myfun` modifies the value of the array passed into it. MATLAB makes a copy in memory of the array pointed to by `A`, sets variable `X` as a reference to this new array, and then sets one row of `X` to zero. The array referenced by `A` remains unchanged:

```
A = magic(500);  
myfun(A);
```

```
function myfun(X)  
X(400,:) = 0;
```

If the calling function needs the modified value of the array it passed to `myfun`, you need to return the updated array as an output of the called function, as shown here for variable `A`:

```
A = magic(500);  
A = myfun(A);  
sprintf('The new value of A is %d', A)
```

```
function Y = myfun(X)  
X(400,:) = 0;  
Y = X;
```

Data Structures and Memory

Memory requirements differ for the various types of MATLAB data structures. You may be able to reduce the amount of memory used for these structures by considering how MATLAB stores them.

Numeric Arrays

MATLAB requires 1, 2, 4, or 8 bytes to store 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers, respectively. For floating-point numbers, MATLAB uses 4 or 8 bytes for `single` and `double` types. To conserve memory when working with numeric arrays, MathWorks recommends that you use the smallest integer or floating-point type that will contain your data without overflowing. For more information, see "Numeric Types" in the MATLAB Programming Fundamentals documentation.

Complex Arrays

MATLAB stores complex data as separate real and imaginary parts. If you make a copy of a complex array variable, and then modify only the real or imaginary part of the array, MATLAB creates a new array containing both real and imaginary parts.

Sparse Matrices

It is best to store matrices with values that are mostly zero in sparse format. Sparse matrices can use less memory and may also be faster to manipulate than full matrices. You can convert a full matrix to sparse format using the `sparse` function.

Compare two 1000-by-1000 matrices: `X`, a matrix of doubles with 2/3 of its elements equal to zero; and `Y`, a sparse copy of `X`. The following example shows that the sparse matrix requires approximately half as much memory:

```
whos
  Name      Size      Bytes  Class
  X         1000x1000  8000000  double array
  Y         1000x1000  4004000  double array (sparse)
```

Cell Arrays

In addition to data storage, cell arrays require a certain amount of additional memory to store information describing each cell. This information is recorded in a *header*, and there is one header for each cell of the array. You can determine the amount of memory required for a cell array header by

finding the number of bytes consumed by a 1-by-1 cell that contains no data, as shown below for a 32-bit system:

```
A = {[]};      % Empty cell array
```

```
whos A
  Name      Size      Bytes  Class  Attributes
  A         1x1         60    cell
```

In this case, MATLAB shows the number of bytes required for each header in the cell array on a 32-bit system to be 60. This is the header size that is used in all of the 32-bit examples in this section. For 64-bit systems, the header size is assumed to be 112 bytes in this documentation. You can find the correct header size on a 64-bit system using the method just shown for 32 bits.

To predict the size of an entire cell array, multiply the number you have just derived for the header by the total number of cells in the array, and then add to that the number of bytes required for the data you intend to store in the array:

$$(\text{header_size} \times \text{number_of_cells}) + \text{data}$$

So a 10-by-20 cell array that contains 400 bytes of data would require 22,800 bytes of memory on a 64-bit system:

$$(112 \times 200) + 400 = 22800$$

Note While numeric arrays must be stored in contiguous memory, structures and cell arrays do not.

Example 1 – Memory Allocation for a Cell Array. The following 4-by-1 cell array records the brand name, screen size, price, and on-sale status for three laptop computers:

```
Laptops = {'SuperrrFast 89X', 'ReliablePlus G5', ...
           'UCanA4dIt 140L6'}; ...
           [single(17), single(15.4), single(14.1)]; ...
           [2499.99, 1199.99, 499.99]; ...
```

```
[true, true, false]]];
```

On a 32-bit system, the cell array header alone requires 60 bytes per cell:

4 cells * 60 bytes per cell = 240 bytes for the cell array

Calculate the memory required to contain the data in each of the four cells:

45 characters * 2 bytes per char = 90 bytes

3 doubles * 8 bytes per double = 24 bytes

3 singles * 4 bytes per single = 12 bytes

3 logicals * 1 byte per logical = 3 bytes

90 + 24 + 12 + 3 = 129 bytes for the data

Add the two, and then compare your result with the size returned by MATLAB:

240 + 129 = 369 bytes total

```
whos Laptops
  Name          Size          Bytes  Class  Attributes
  Laptops       4x1             369   cell
```

Structures

```
S.A = [];
B = whos('S');
B.bytes - 60
ans =
    64
```

Compute the memory needed for a structure array as follows:

32-bit systems: fields x ((60 x array elements) + 64) + data

64-bit systems: fields x ((112 x array elements) + 64) + data

On a 64-bit computer system, a 4-by-5 structure `Clients` with fields `Address` and `Phone` uses 4,608 bytes just for the structure:

$$2 \text{ fields} \times ((112 \times 20) + 64) = 2 \times (2240 + 64) = 4608 \text{ bytes}$$

To that sum, you must add the memory required to hold the data assigned to each field. If you assign a 25-character string to `Address` and a 12-character string to `Phone` in each element of the 4-by-5 `Clients` array, you use 1480 bytes for data:

$$(25+12) \text{ characters} \times 2 \text{ bytes per char} \times 20 \text{ elements} = 1480 \text{ bytes}$$

Add the two and you see that the entire structure consumes 6,088 bytes of memory.

Example 1 – Memory Allocation for a Structure Array. Compute the amount of memory that would be required to store the following 6-by-5 structure array having the following four fields on a 32-bit system:

- A: 5-by-8-by-6 signed 8-bit integer array
- B: 1-by-200 single array
- C: 30-by-30 unsigned 16-bit integer array
- D: 1-by-27 character array

Construct the array:

```
A = int8(ones(5,8,6));
B = single(1:500);
C = uint16(magic(30));
D = 'Company Name: MathWorks';

s = struct('f1', A, 'f2', B, 'f3', C, 'f4', D);

for m=1:6
    for n=1:5
        s(m,n)=s(1,1);
    end
end
```

Calculate the amount of memory required for the structure itself, and then for the data it contains:

$$\begin{aligned} \text{structure} &= \text{fields} \times ((60 \times \text{array elements}) + 64) = \\ &4 \times ((60 \times 30) + 64) = 7,456 \text{ bytes} \end{aligned}$$

```
data = (field1 + field2 + field3 + field4) x array elements =  
      (240 + 2000 + 1800 + 54) x 30 = 122,820 bytes
```

Add the two, and then compare your result with the size returned by MATLAB:

Total bytes calculated for structure s: 7,456 + 122,820 = 130,276

```
whos s  
  Name      Size      Bytes  Class  Attributes  
  s         6x5       130036 struct
```

Memory Management Functions

The following functions can help you to manage memory use while running the MATLAB software:

- `memory` displays or returns information about how much memory is available and how much is used by MATLAB. This includes the following:
 - Size of the largest single array MATLAB can create at this time.
 - Total size of the virtual address space available for data.
 - Total amount of memory used by the MATLAB process for both libraries and data.
 - Available and total Virtual Memory for the MATLAB software process.
 - Available system memory, including both physical memory and paging file.
 - Available and the total physical memory (RAM) of the computer.
- `whos` shows how much memory MATLAB currently has allocated for variables in the workspace.
- `pack` saves existing variables to disk, and then reloads them contiguously. This reduces the chances of running into problems due to memory fragmentation.
- `clear` removes variables from memory. One way to increase the amount of available memory is to periodically clear variables from memory that you no longer need.

If you use `pack` and there is still not enough free memory to proceed, you probably need to remove some of the variables you are no longer using from memory. Use `clear` to do this.

- `save selectively` stores variables to the disk. This is a useful technique when you are working with large amounts of data. Save data to the disk periodically, and then use the `clear` function to remove the saved data from memory.
- `load` reloads a data file saved with the `save` function.
- `quit` exits MATLAB and returns all allocated memory to the system. This can be useful on The Open Group UNIX systems, which do not free up

memory allocated to an application (e.g., MATLAB) until the application exits.

You can use the `save` and `load` functions in conjunction with the `quit` command to free memory by:

- 1 Saving any needed variables with the `save` function.
- 2 Quitting MATLAB to free all memory allocated to MATLAB.
- 3 Starting a new MATLAB session and loading the saved variables back into the clean MATLAB workspace.

The `whos` Function

The `whos` command can give you an idea of the memory used by MATLAB variables.

```
A = ones(10,10);
whos
```

Name	Size	Bytes	Class	Attributes
A	10x10	800	double	

Note that `whos` does not include information about

- Memory used by MATLAB (e.g., Sun Java code and plots).
- Memory used for most objects (e.g., time series, custom) .
- Memory for variables not in the calling workspace .
- Shared data copies. `whos` shows bytes used for a shared data copy even when it does not use any memory. This example shows that `whos` reports an array (A) and a shared data copy of that array (B) separately, even though the data exists only once in memory:

Store 400 MB array as A. Memory used = 381MB (715 MB – 334 MB) :

```
memory
Memory used by MATLAB:          334 MB (3.502e+008 bytes)
```

```
A = rand(5e7,1);
```

```
memory
Memory used by MATLAB:          715 MB (7.502e+008 bytes)
```

```
whos
  Name          Size          Bytes  Class  Attributes
  A             50000000x1      400000000  double
```

Create B and point it to A. Note that although whos shows both A and B, there is only one copy of the data in memory. No additional memory is consumed by assigning A to B:

```
B = A;
```

```
memory
Memory used by MATLAB:          715 MB (7.502e+008 bytes)
```

```
whos
  Name          Size          Bytes  Class  Attributes
  A             50000000x1      400000000  double
  B             50000000x1      400000000  double
```

Modifying B(1) copies all of A to B and changes the value of B(1). Memory used = 382MB (1097 MB – 715 MB). Now there are two copies of the data in memory, yet the output of whos does not change:

```
B(1) = 3;
```

```
memory
Memory used by MATLAB:          1097 MB (1.150e+009 bytes)
```

```
whos
  Name          Size          Bytes  Class  Attributes
  A             50000000x1      400000000  double
  B             50000000x1      400000000  double
```

Strategies for Efficient Use of Memory

In this section...

“Ways to Reduce the Amount of Memory Required” on page 21-15

“Using Appropriate Data Storage” on page 21-17

“How to Avoid Fragmenting Memory” on page 21-20

“Reclaiming Used Memory” on page 21-21

Ways to Reduce the Amount of Memory Required

The source of many "out of memory" problems often involves analyzing or processing an existing large set of data such as in a file or a database. This requires bringing all or part of the data set into the MATLAB software process. The following techniques deal with minimizing the required memory during this stage.

Load Only As Much Data As You Need

Only import into MATLAB as much of a large data set as you need for the problem you are trying to solve. This is not usually a problem when importing from sources such as a database, where you can explicitly search for elements matching a query. But this is a common problem with loading large flat text or binary files. Rather than loading the entire file, use the appropriate MATLAB function to load parts of files.

MAT-Files. Load part of a variable by indexing into an object that you create with the `matfile` function.

Text Files. Use the `textscan` function to access parts of a large text file by reading only the selected columns and rows. If you specify the number of rows or a repeat format number with `textscan`, MATLAB calculates the exact amount of memory required beforehand.

Binary Files. You can use low-level binary file I/O functions, such as `fread`, to access parts of any file that has a known format. For binary files of an unknown format, try using memory mapping with the `memmapfile` function.

Image, HDF, Audio, and Video Files. Many of the MATLAB functions that support loading from these types of files allow you to select portions of the data to read. For details, see the function reference pages listed in “Supported File Formats”.

Process Data By Blocks

Consider block processing, that is, processing a large data set one section at a time in a loop. Reducing the size of the largest array in a data set reduces the size of any copies or temporaries needed. You can use this technique in either of two ways:

- For a subset of applications that you can break into separate chunks and process independently.
- For applications that only rely on the state of a previous block, such as filtering.

Avoid Creating Temporary Arrays

Avoid creating large temporary variables, and also make it a practice to clear those temporary variables you do use when they are no longer needed. For example, when you create a large array of zeros, instead of saving to a temporary variable *A*, and then converting *A* to a single:

```
A = zeros(1e6,1);  
As = single(A);
```

use just the one command to do both operations:

```
A = zeros(1e6,1,'single');
```

Using the `repmat` function, array preallocation and `for` loops are other ways to work on `nondouble` data without requiring temporary storage in memory.

Use Nested Functions to Pass Fewer Arguments

When working with large data sets, be aware that MATLAB makes a temporary copy of an input variable if the called function modifies its value. This temporarily doubles the memory required to store the array, which causes MATLAB to generate an error if sufficient memory is not available.

One way to use less memory in this situation is to use nested functions. A nested function shares the workspace of all outer functions, giving the nested function access to data outside of its usual scope. In the example shown here, nested function `setrowval` has direct access to the workspace of the outer function `myfun`, making it unnecessary to pass a copy of the variable in the function call. When `setrowval` modifies the value of `A`, it modifies it in the workspace of the calling function. There is no need to use additional memory to hold a separate array for the function being called, and there also is no need to return the modified value of `A`:

```
function myfun
A = magic(500);

    function setrowval(row, value)
        A(row,:) = value;
    end

setrowval(400, 0);
disp('The new value of A(399:401,1:10) is')
A(399:401,1:10)
end
```

Using Appropriate Data Storage

MATLAB provides you with different sizes of data classes, such as `double` and `uint8`, so you do not need to use large classes to store your smaller segments of data. For example, it takes 7 KB less memory to store 1,000 small unsigned integer values using the `uint8` class than it does with `double`.

Use the Appropriate Numeric Class

The numeric class you should use in MATLAB depends on your intended actions. The default class `double` gives the best precision, but requires 8 bytes per element of memory to store. If you intend to perform complicated math such as linear algebra, you must use a floating-point class such as a `double` or `single`. The `single` class requires only 4 bytes. There are some limitations on what you can do with `singles`, but most MATLAB Math operations are supported.

If you just need to carry out simple arithmetic and you represent the original data as integers, you can use the integer classes in MATLAB. The following is

a list of numeric classes, memory requirements (in bytes), and the supported operations.

Class (Data Type)	Bytes	Supported Operations
single	4	Most math
double	8	All math
logical	1	Logical/conditional operations
int8, uint8	1	Arithmetic and some simple functions
int16, uint16	2	Arithmetic and some simple functions
int32, uint32	4	Arithmetic and some simple functions
int64, int64	8	Arithmetic and some simple functions

Reduce the Amount of Overhead When Storing Data

MATLAB arrays (implemented internally as `mxArrays`) require room to store meta information about the data in memory, such as type, dimensions, and attributes. This takes about 80 bytes per array. This overhead only becomes an issue when you have a large number (e.g., hundreds or thousands) of small `mxArrays` (e.g., scalars). The `whos` command lists the memory used by variables, but does not include this overhead.

Because simple numeric arrays (comprising one `mxArray`) have the least overhead, you should use them wherever possible. When data is too complex to store in a simple array (or matrix), you can use other data structures.

Cell arrays are comprised of separate `mxArrays` for each element. As a result, cell arrays with many small elements have a large overhead.

Structures require a similar amount of overhead per field (see the documentation on “Array Headers” on page 21-4 above). Structures with many fields and small contents have a large overhead and should be avoided. A large array of structures with numeric scalar fields requires much more memory than a structure with fields containing large numeric arrays.

Also note that while MATLAB stores numeric arrays in contiguous memory, this is not the case for structures and cell arrays.

Import Data to the Appropriate MATLAB Class

When reading data from a binary file with `fread`, it is a common error to specify only the class of the data in the file, and not the class of the data MATLAB uses once it is in the workspace. As a result, the default `double` is used even if you are reading only 8-bit values. For example,

```
fid = fopen('large_file_of_uint8s.bin', 'r');
a = fread(fid, 1e3, 'uint8');           % Requires 8k
whos a
  Name      Size      Bytes  Class  Attributes

  a         1000x1      8000  double

a = fread(fid, 1e3, 'uint8=>uint8');    % Requires 1k
whos a
  Name      Size      Bytes  Class  Attributes

  a         1000x1      1000  uint8
```

Make Arrays Sparse When Possible

If your data contains many zeros, consider using sparse arrays, which store only nonzero elements. The example below compares the space required for storage of an array of mainly zeros:

```
A = diag(1e3,1e3);    % Full matrix with ones on the diagonal
As = sparse(A)        % Sparse matrix with only nonzero elements
whos
  Name      Size      Bytes  Class

  A         1001x1001  8016008  double array
  As        1001x1001  4020     double array (sparse)
```

You can see that this array requires only approximately 4 KB to be stored as sparse, but approximately 8 MB as a full matrix. In general, for a sparse double array with `nnz` nonzero elements and `ncol` columns, the memory required is

- $16 * nnz + 8 * ncol + 8$ bytes (on a 64 bit machine)
- $12 * nnz + 4 * ncol + 4$ bytes (on a 32 bit machine)

Note that MATLAB does not support all mathematical operations on sparse arrays.

How to Avoid Fragmenting Memory

MATLAB always uses a contiguous segment of memory to store a numeric array. As you manipulate this data, however, the contiguous block can become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store a new large variable. Increasing fragmentation can use significantly more memory than is necessary.

Preallocate Contiguous Memory When Creating Arrays

In the course of a MATLAB session, memory can become fragmented due to dynamic memory allocation and deallocation. `for` and `while` loops that incrementally increase, or *grow*, the size of a data structure each time through the loop can add to this fragmentation as they have to repeatedly find and allocate larger blocks of memory to store the data.

To make more efficient use of your memory, preallocate a block of memory large enough to hold the matrix at its final size before entering the loop. When you preallocate memory for an array, MATLAB reserves sufficient contiguous space for the entire full-size array at the beginning of the computation. Once you have this space, you can add elements to the array without having to continually allocate new space for it in memory.

For more information on preallocation, see “Preallocating Arrays” on page 20-4.

Allocate Your Larger Arrays First

MATLAB uses a heap method of memory management. It requests memory from the operating system when there is not enough memory available in the heap to store the current variables. It reuses memory as long as the size of the memory segment required is available in the heap.

The following statements can require approximately 4.3 MB of RAM. This is because MATLAB may not be able to reuse the space previously occupied by two 1 MB arrays when allocating space for a 2.3 MB array:

```
a = rand(1e6,1);
```



```
b = rand(1e6,1);  
clear  
c = rand(2.3e6,1);
```

The simplest way to prevent overallocation of memory is to allocate the largest vectors first. These statements require only about 2.0 MB of RAM:

```
c = rand(2.3e6,1);  
clear  
a = rand(1e6,1);  
b = rand(1e6,1);
```

Long-Term Usage (Windows Systems Only)

On 32-bit Microsoft Windows, the workspace of MATLAB can fragment over time due to the fact that the Windows memory manager does not return blocks of certain types and sizes to the operating system. Clearing the MATLAB workspace does not fix this problem. You can minimize the problem by allocating the largest variables first. This cannot address, however, the eventual fragmentation of the workspace that occurs from continual use of MATLAB over many days and weeks, for example. The only solution to this is to save your work and restart MATLAB.

The pack command, which saves all variables to disk and loads them back, does not help with this situation.

Reclaiming Used Memory

One simple way to increase the amount of memory you have available is to clear large arrays that you no longer use.

Save Your Large Data Periodically to Disk

If your program generates very large amounts of data, consider writing the data to disk periodically. After saving that portion of the data, use the `clear` function to remove the variable from memory and continue with the data generation.

Clear Old Variables from Memory When No Longer Needed

When you are working with a very large data set repeatedly or interactively, clear the old variable first to make space for the new variable. Otherwise, MATLAB requires temporary storage of equal size before overriding the variable. For example,

```
a = rand(100e6,1)           % 800 MB array
b = rand(100e6,1)           % New 800 MB array
Error using rand
Out of memory. Type HELP MEMORY for your options.

clear a
a = rand(100e6,1)           % New 800 MB array
```

Resolving “Out of Memory” Errors

In this section...

“General Suggestions for Reclaiming Memory” on page 21-23

“Setting the Process Limit” on page 21-24

“Disabling Java VM on Startup” on page 21-25

“Increasing System Swap Space” on page 21-26

“Using the 3GB Switch on Windows Systems” on page 21-26

“Freeing Up System Resources on Windows Systems” on page 21-27

General Suggestions for Reclaiming Memory

The MATLAB software generates an Out of Memory message whenever it requests a segment of memory from the operating system that is larger than what is currently available. When you see the Out of Memory message, use any of the techniques discussed under “Strategies for Efficient Use of Memory” on page 21-15 to help optimize the available memory. If the Out of Memory message still appears, you can try any of the following:

- Compress data to reduce memory fragmentation.
- If possible, break large matrices into several smaller matrices so that less memory is used at any one time.
- If possible, reduce the size of your data.
- Make sure that there are no external constraints on the memory accessible to MATLAB. (On The Open Group UNIX³ systems, use the `limit` command to check).
- Increase the size of the swap file. We recommend that you configure your system with twice as much swap space as you have RAM. See “Increasing System Swap Space” on page 21-26, below.
- Add more memory to the system.

3. UNIX is a registered trademark of The Open Group in the United States and other countries.

Setting the Process Limit

The platforms and operating systems that MATLAB supports have different memory characteristics and limitations. In particular, the *process limit* is the maximum amount of virtual memory a single process (or application) can address. On 32-bit systems, this is the most important factor limiting data set size. The process limit must be large enough for MATLAB to store all of the data it is to process, any MATLAB program files, the MATLAB executable itself, and additional state information.

Where possible, choose an operating system that maximizes this number, that is, a 64-bit operating system. The following is a list of MATLAB supported operating systems and their process limits.

Operating System	Process Limit
32-bit Microsoft Windows XP, Windows Vista™, Windows 7	2 GB
32-bit Windows XP with 3 GB boot.ini switch or 32-bit Windows Vista or Windows 7 with increaseuserva set (see later)	3 GB
32-bit Linux® (Linux is a registered trademark of Linus Torvalds)	~3 GB
64-bit Windows or Linux running 32-bit MATLAB	≤ 4 GB
64-bit Windows, Apple Macintosh® OS X, or Linux running 64-bit MATLAB	8 TB

To verify the current process limit of MATLAB on Windows systems, use the `memory` function.

```
Maximum possible array:          583 MB (6.111e+008 bytes) *
Memory available for all arrays: 1515 MB (1.588e+009 bytes) **
Memory used by MATLAB:          386 MB (4.050e+008 bytes)
Physical Memory (RAM):          2014 MB (2.112e+009 bytes)
```

* Limited by contiguous virtual address space available.

** Limited by virtual address space available.

When called with one output variable, the `memory` function returns or displays the following values. See the function reference for `memory` to find out how to use it with more than one output.

memory Return Value	Description
<code>MaxPossibleArrayBytes</code>	Size of the largest single array MATLAB can currently create
<code>MemAvailableAllArrays</code>	Total size of the virtual address space available for data
<code>MemUsedMATLAB</code>	Total amount of memory used by the MATLAB process

View the value against the Total entry in the Virtual Memory section. It is shown as 2 GB in the table, which is the default on Windows XP systems. On UNIX systems, see the `ulimit` command to view and set user limits including virtual memory.

Disabling Java VM on Startup

On UNIX systems, you can increase the workspace size by approximately 400 MB if you start MATLAB without the Sun Java VM. To do this, use the command line option `-nojvm` to start MATLAB. This also increases the size of the largest contiguous block (and therefore the largest matrix) by about the same.

Using `-nojvm` comes with a penalty in that you will lose many features that rely on the Java software, including the entire development environment.

Starting MATLAB with the `-nodesktop` option does not save any substantial amount of memory.

Shutting down other applications and services (e.g., using `msconfig` on Windows systems) can help if total system memory is the limiting factor, but usually process limit (on 32-bit machines) is the main limiting factor.

Increasing System Swap Space

The total memory available to applications on your computer is comprised of physical memory (RAM), plus a *page file*, or *swap file*, on disk. The swap file can be very large (e.g., 16 TB on 32-bit Windows, 512 TB on 64-bit Windows). The operating system allocates the virtual memory of each process to physical memory or to the swap file, depending on the needs of the system and other processes.

Most systems allow you to control the size of your swap file. The steps involved depend on the system you are running on.

Note There is no interface for directly controlling the swap space on Mac OS® X systems.

Windows Systems

Use the Windows Control Panel to change the size of the virtual memory paging file on your system. For more information, refer to the Windows help.

Linux Systems

You can change your swap space by using the `mkswap` and `swapon` commands. For more information on the above commands, type `man` followed by the command name at the Linux prompt.

Using the 3GB Switch on Windows Systems

Microsoft Windows XP systems can allocate 3 GB (instead of the default 2 GB) to processes, if you set an appropriate switch in the `boot.ini` file of the system. MathWorks recommends that you only do this with Windows XP SP2 systems or later. This gives an extra 1 GB of virtual memory to MATLAB, not contiguous with the rest of the memory. This enables you to store more data, but not larger arrays, as these are limited by contiguous space. This is mostly beneficial if you have enough RAM (e.g., 3 or 4 GB) to use it.

After setting the switch, confirm the new value of the virtual memory after restarting your computer and using the `memory` function.

```
[userview systemview] = memory;  
  
systemview.VirtualAddressSpace  
ans =  
    Available: 1.6727e+009    % Virtual memory available to MATLAB.  
    Total: 2.1474e+009    % Total virtual memory
```

For more documentation on this option, use the following URL:

<http://support.microsoft.com/kb/291988>

Similarly, on machines running Microsoft Windows Vista and Windows 7, you can achieve the same effect by using the command:

```
BCDEdit /set increaseuserva 3072
```

For more information on this option, go to the following website:

<http://msdn.microsoft.com>

Freeing Up System Resources on Windows Systems

There are no functions implemented to manipulate the way MATLAB handles Microsoft Windows system resources. Windows systems use these resources to track fonts, windows, and screen objects. Resources can be depleted by using multiple figure windows, multiple fonts, or several UI controls. One way to free up system resources is to close all inactive windows. Windows system icons still use resources.

Create Help and Demos

- “Types of Help You Can Create” on page 22-2
- “Add Help for Your Program Files” on page 22-4
- “Add Documentation to the Help Browser” on page 22-12
- “Add Demos to the Help Browser” on page 22-48
- “Address Validation Errors for info.xml Files” on page 22-57

Types of Help You Can Create

You can provide help and demos for the files you create and have them appear formatted like the help and demos MATLAB provides. Including help can be worthwhile for you and others with whom you share your files. As the following table explains, you can provide help in various forms. The table presents guidelines for creating the kinds of help that best suit your program files and the people who need to use them.

Type of Help	Description	See
Help comments	<ul style="list-style-type: none"> • Describe individual program files you create • Provide formatted comments at the start of a MATLAB program file • Display the help comments when you type <code>help filename</code> • Easy to provide 	“Add Help for Your Program Files” on page 22-4
Contents.m file	<ul style="list-style-type: none"> • Describes a collection of program files • Provides a summary file for all files in a folder • Displays the summary when you type <code>help foldername</code> • Can include a version number • Easy to provide • Can be empty to avoid listing folder contents 	“Help Summary for Your Program Files (Contents.m)” on page 22-6
MATLAB class files	<ul style="list-style-type: none"> • Describes classes you create • Provides help in the class definition file, and optionally for class methods, properties and events • View the help by running <code>help classname</code> or <code>doc classname</code> • Easy to provide, but requires object-oriented programming knowledge to create classes 	“Help for Classes You Create” on page 22-7

Type of Help	Description	See
Documentation in the Help browser	<ul style="list-style-type: none">• Supports graphics, images, stylized text, and page formatting. Suited for how-to and conceptual information that helps others run your files• Can include reference pages for functions and blocks• Can include a search database to support searching your documentation• More effort than providing help in program files• Requires the ability to create HTML files and edit XML files	“Add Documentation to the Help Browser” on page 22-12
Demos in the Help browser	<ul style="list-style-type: none">• Suited for explaining how something works, step-by-step. Supports graphics, images, stylized text, and page formatting• Others can view, edit and run your demos• Can be generated from code scripts• View the demos using the Contents pane in the Help browser	“Add Demos to the Help Browser” on page 22-48

Add Help for Your Program Files

In this section...

“Help Within a Program File” on page 22-4

“Help Summary for Your Program Files (Contents.m)” on page 22-6

“Help for Classes You Create” on page 22-7

Help Within a Program File

Help consists of lines of comments at the beginning of a program file. Help that you add to your file displays like the help for MATLAB functions. The first help text line, often called the H1 line, includes the function name and a brief description.

For example, here is an excerpt from the file `hist.m`:

```
function [no,xo] = hist(varargin)
%HIST Histogram.
%   N = HIST(Y) bins the elements of Y into 10 equally spaced containers
%   and returns the number of elements in each container. If Y is a
%   matrix, HIST works down the columns.
%
%   N = HIST(Y,M), where M is a scalar, uses M bins.
%
%   N = HIST(Y,X), where X is a vector, returns the distribution of Y
%   among bins with centers specified by X. The first bin includes
%   data between -inf and the first center and the last bin
%   includes data between the last bin and inf. Note: Use HISTC if
%   it is more natural to specify bin edges instead.
%
%   Class support for inputs Y, X:
%       float: double, single
%
%   See also HISTC, MODE.
```

When you type `help hist`, the help text displays in the Command Window:

```
hist Histogram.
```

`N = hist(Y)` bins the elements of `Y` into 10 equally spaced containers and returns the number of elements in each container. If `Y` is a matrix, `hist` works down the columns.

`N = hist(Y,M)`, where `M` is a scalar, uses `M` bins.

`N = hist(Y,X)`, where `X` is a vector, returns the distribution of `Y` among bins with centers specified by `X`. The first bin includes data between `-inf` and the first center and the last bin includes data between the last bin and `inf`. Note: Use `HISTC` if it is more natural to specify bin edges instead.

Class support for inputs `Y`, `X`:

float: double, single

See also [histc](#), [mode](#).

Overloaded methods:

[fints/hist](#)

[categorical/hist](#)

Reference page in Help browser

[doc hist](#)

Create **See also** links by including function names at the end of your help text on a line that begins with `% See also`. The list of names can include MATLAB functions, toolbox functions, and your own functions. If the function exists on the search path or in the current folder, the `help` command displays each of these function names as a hyperlink to its help. Otherwise, MATLAB prints the function names as they appear in the help text.

End your help text with a blank line (without a `%`).

Tip To make your help easy for readers to follow, be consistent in how you structure it.

You can include hyperlinks (in the form of URLs) to HTML files or Web sites anywhere in your help text. Create hyperlinks by including an HTML

`<A> ` anchor element. Within the anchor, use a `matlab:` statement (pronounced *matlabcolon*) to execute a web command. For example:

```
% For more information, see <a href="matlab:  
% web('http://www.mathworks.com')">the MathWorks Web site</a>.
```

When you are connected to the Internet and click the link, MATLAB opens a Web browser window to display the URL. For more information, see “Display Hyperlinks in the Command Window”.

For information about formatting help in the Editor, see “Add Comments”.

For related information, see “Generating a Summary View of the Help Components in Functions and Scripts”.

Help Summary for Your Program Files (Contents.m)

Provide a summary file for your own collection of program files using the same method as MATLAB. In MATLAB, each folder containing program files includes a file named `Contents.m` (with a capital C) that lists the functions in the folder with a brief description of each. You can view the information in a `Contents.m` file from the command line in several ways:

- `help foldername` displays the text from the `Contents.m` file for that folder. The displayed help has hyperlinks to help for the individual functions.
- `doc foldername` displays the same information in the Help browser.
- `ver toolboxname` displays version information for the specified toolbox, followed by its formatted `Contents.m` file entries.

To create your own `Contents.m` files:

- In the Editor, display a `Contents.m` file provided with MATLAB to see its structure. Most folders in the program tree contain a `Contents.m` file. For an example, see `matlabroot\toolbox\matlab\helptools\Contents.m`.
- Read about “Displaying and Updating a Report on the Contents of a Folder” to learn how to easily create and maintain your `Contents.m` files.
- Provide your own toolbox name, a version, and a date in the first two lines of the `Contents.m` file, which the `ver` function displays:

```
% Toolbox name description
% Version xxx dd-mmm-yyyy
```

Do not include any spaces in the date. Use this format: 12-Mar-2010.

Tip If you do *not* want your users to see a summary of your toolbox functions, place an empty `Contents.m` file in the toolbox folder. An empty `Contents.m` causes `help foldername` to report `No help found for foldername`.

The Upslope Area toolbox example folder contains a `Contents.m` file. For more information, see “Learning to Add Help from Examples” on page 22-13.

You can create a categorical listings of functions for the Help browser by marking up your `Contents.m` file and publishing it to HTML. To learn more, see “Creating Function and Block Category Listings” on page 22-33.

Help for Classes You Create

If you create your own MATLAB classes, you can provide help for the class by including comments in the class definition file:

- Provide help about the class in comment lines directly following the `classdef` statement.
- Add a comment line directly after the constructor method for the class.
- Add comments directly after other methods and next to property definitions.

List the properties and methods of the class within the first block of comments after `classdef`. If you format the list in as described here, MATLAB renders the property and method names you list as hyperlinks to their definitions, which appear later in the same file:

1 To create links from this section to your class properties, add a line:

```
% Classname Properties:
```

where *Classname* is the name from your `classdef`. Be sure to put a colon (`:`) after `Properties`.

- 2 List your property names (with optional same-line descriptions) on the following lines. For example:

```
% prop1 - first property
% prop2 - second property
```

- 3 List the methods; enter the class name followed by **Methods:** (include a colon). Then, list your methods (with optional same-line descriptions) on the following lines, as follows:

```
% MyClass Methods:
% method1 - first method
% method2 - second method
```

View help for your class in the Command Window:

```
help classname
```

To view the same help for the class in the Help browser, run:

```
doc classname
```

Note You do not need to prepare HTML versions of class definition file help. MATLAB generates an HTML page from each class definition automatically and displays it in the Help browser.

For more information about getting help for classes, read the following section. To learn more about how to create class definitions, see “User-Defined Classes”.

Example of Help for a Externally Supplied Class

The following example shows help for a class file, `sads.m`, an example provided with MATLAB documentation. If you create help for your class files, the help should look and work like this example.

Follow these steps to see the help for the example.

- 1 Make sure you can access the examples folder by either:

- Making the folder containing the example file as your current folder:

```
cd(fullfile(matlabroot,'help','techdoc','matlab_env','examples'))
```

- Adding the examples folder to the search path:

```
addpath(fullfile(matlabroot,'help','techdoc','matlab_env','examples'))
```

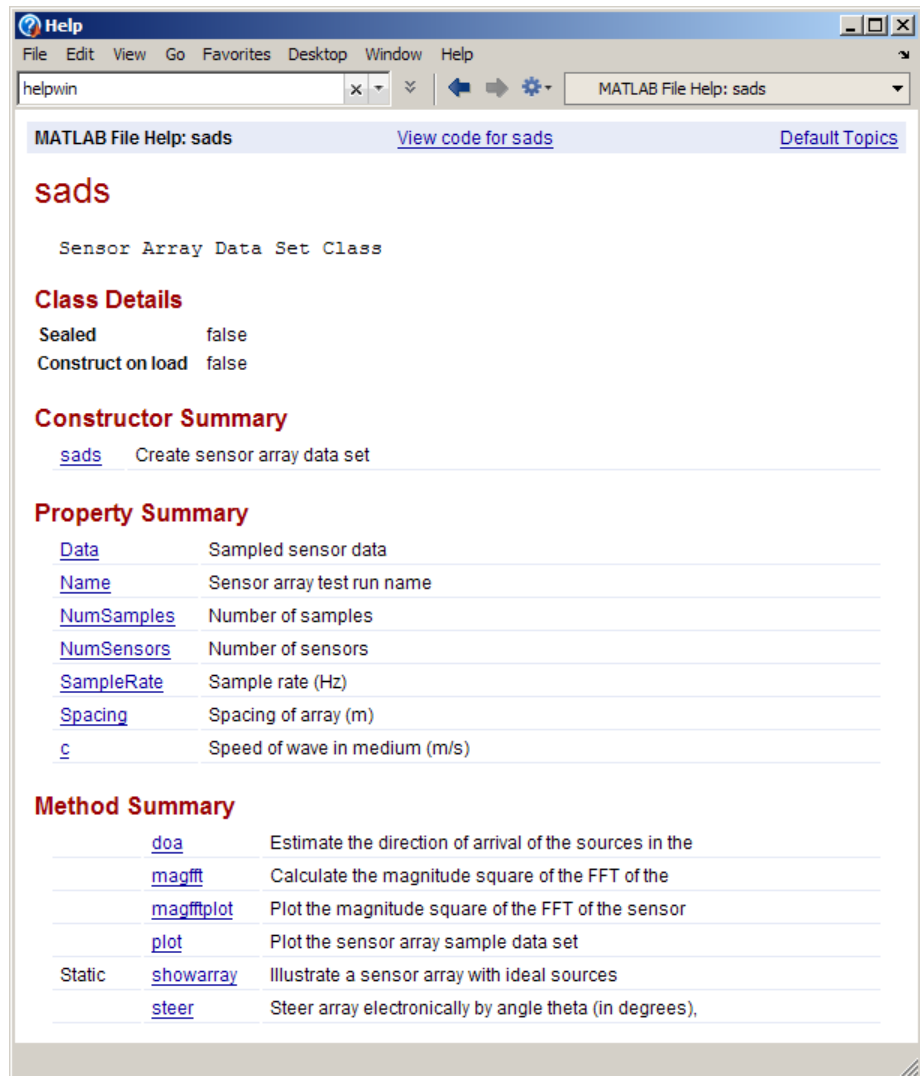
Or, [Click here](#) to add the help examples folder to the search path.

- 2** Open the class file in the Editor to see the help comments.

```
open('sads.m')
```

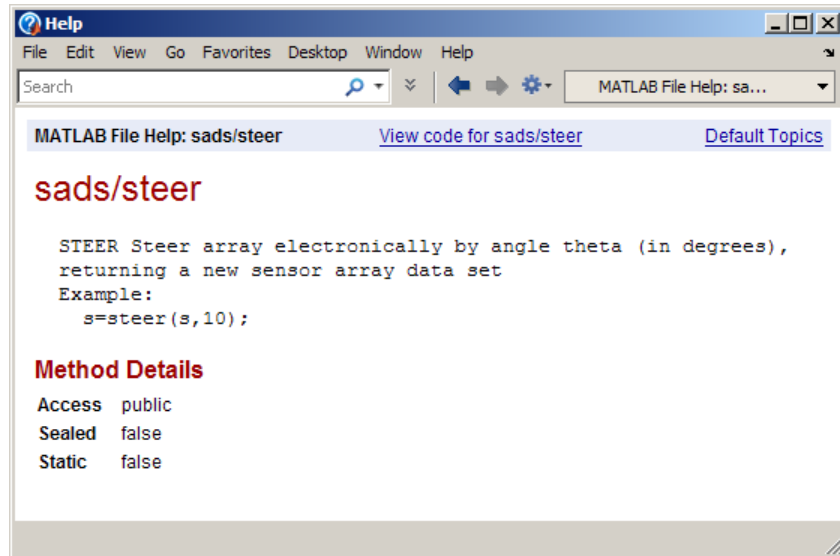
- 3** View help for the `sads` class in the Help browser:

```
doc sads
```



- 4 Access more information by following links on the class help page or by using the `doc` function. For example, to get help for the `steer` method, do one of the following:
- Click the `steer` link under Method Summary.

- Run `doc sads.steer`.



You also can open the `sads.m` file by clicking the **View code for sads** link at the top of the `sads` help page, as described previously.

- 5 Next, view information about `sads` objects another way. Create an instance of `sads`, for example, `sensorArray`, and then open that object in the Variable Editor:

```
loadparameters
```

```
sensorArray=sads(Data, Wavelength,SampleRate,Spacing,Name);
```

```
openvar sensorArray % or double-click sensorArray in the Workspace browser
```

Property	Value	Min	Max
Data	<64x16 complex double>	0.0130...	-1.939...
SampleRate	3.3333e+07	3.3333...	3.3333...
Spacing	1.2500	1.2500	1.2500
Name	'Sensor Array Amplitudes'		
c	3000000000	30000...	30000...
NumSensors	16	16	16
NumSamples	64	64	64

Add Documentation to the Help Browser

In this section...

- “Types of Documentation You Can Provide” on page 22-12
- “Learning to Add Help from Examples” on page 22-13
- “Summary of Creating and Installing HTML Help Files” on page 22-14
- “Organizing Your Documentation” on page 22-15
- “Creating Function Reference Pages” on page 22-28
- “Creating Function and Block Category Listings” on page 22-33
- “Making Your HTML Help Files Searchable” on page 22-40
- “Summary of Workflow for Providing HTML Help Files” on page 22-42

Types of Documentation You Can Provide

MathWorks and third parties provide a rich set of toolboxes, blocksets, and target and link products. Almost all such products come with documentation that displays in the Help browser.

If you create a toolbox that works with MathWorks products—even if it only contains a few functions—you can include with it HTML help files that you and others can access using the Help browser. Providing HTML help files for your toolbox allows you to include figures, diagrams, screen captures, equations, and formatting to make your toolbox more usable.

Consider providing the following types of content in your documentation:

- A start page for your toolbox (called a “roadmap”)
- A quick introduction to your toolbox (“getting started guide”)
- A detailed explanation of using your toolbox (“user guide”)
- Function or block reference pages
- A list of examples, hyperlinked to the documentation set
- Release notes, describing improvements, limitations, known bugs, and so forth

- PDF versions of your HTML files (typically accessed from the roadmap page)

Except for the PDF version of documentation, each of these types of help is a set of one or more HTML pages that you create in the Editor, word processing software, or an HTML authoring environment. Many such applications can also export their source documents as PDF files.

Note You are free to organize and format your help documentation as you choose. However, if you structure your help files similarly to documentation from MathWorks, people who use it will understand where to find specific types of information.

Learning to Add Help from Examples

To learn how to create documentation for the Help browser, refer to examples. This documentation provides two folders that you can copy. These folders contain:

- Template XML files containing required and optional sections, with explanatory comments

You find this folder in

matlabroot/help/techdoc/matlab_env/examples/templates. Always work with *copies* of the files it contains when making modifications. You must edit in your own content to the templates to make them usable.

- A complete toolbox with code you can run (called Upslope Area Toolbox), accompanied by extensive HTML documentation that you can view in the Help browser

Find this example in the folder

matlabroot/help/techdoc/matlab_env/examples/upslope. You can use functions from the toolbox and view the help by adding the folder to the search path. However, if you choose to modify any of the files it contains, copy the entire *upslope* folder to a working folder.

Note Some functionality of Upslope Area Toolbox depends on Image Processing Toolbox™. If you have Image Processing Toolbox installed, [clicking here](#) brings you to its documentation.

The following sections primarily discuss the XML template files for `info.xml` and `helptoc.xml`, showing you how to modify them to create a documentation set. The examples folder provides templates only for XML files, not HTML files. Therefore, to understand how the XML files access HTML documentation files and what those files contain, refer to corresponding files in the example Upslope Area Toolbox folder.

Summary of Creating and Installing HTML Help Files

To add your own documentation to the Help browser, you need to:

- 1** Decide what types of documentation you want to provide and create HTML help files for your toolbox. See “Types of Documentation You Can Provide” on page 22-12.
- 2** Create an `info.xml` and `helptoc.xml` files based on examples. See “Organizing Your Documentation” on page 22-15.
- 3** Optionally create a search database to include your HTML help files in the Help browser search results. See “Making Your HTML Help Files Searchable” on page 22-40.
- 4** Add the HTML files to the Help browser, by editing and incorporating XML and other special files you create. For step-by-step instructions, see “Summary of Workflow for Providing HTML Help Files” on page 22-42.
- 5** Provide the help files to your program users, along with instructions for including the files in the Help browser.

To create HTML help files, use the MATLAB Editor, another text editor, or an HTML editing tool. If you have an XML authoring system, you can develop documentation in that environment and export it as HTML files.

If you use the Editor, enabling syntax highlighting and indenting features will help as you author HTML and XML files. The editor can automatically color syntax for `.htm`, `.html`, and `.xml` source files.

Tip To customize the syntax highlighting and indenting in the Editor, select **File > Preferences > Editor/Debugger > Language**, and choose XML/HTML.

Verify how your HTML files appear in the Help browser. To view an HTML help file that you created, use the `web` function. For example, display an HTML file from the set of examples provided for this topic:

```
web(fullfile(matlabroot, 'help', 'techdoc', 'matlab_env', ...
    'examples', 'upslope', 'html', 'upslope_functions_by_cat.html'))
```

Organizing Your Documentation

After you decide which types of documentation to show in the Help browser, you need to provide HTML and XML files, and link them to work together. The following sections describe how to set up your help documentation.

- “Setting Up a Help Folder” on page 22-16
- “XML Files Required to Add Documentation and Demos” on page 22-18
- “Identifying a Help Folder: the info.xml File” on page 22-18
- “Customizing the info.xml Template File” on page 22-21
- “More About the info.xml File” on page 22-22
- “Creating the Table of Contents File: helptoc.xml” on page 22-24
- “More About the helptoc.xml File” on page 22-27

Note To view the content and organization of the Upslope Area toolbox documentation example, place it on the search path by clicking here, or run this command:

```
addpath(fullfile(matlabroot, 'help', 'techdoc', 'matlab_env', 'examples', 'upslope'))
```

Setting Up a Help Folder

Make a folder to hold HTML and XML files. The folder can contain subfolders to organize HTML and image files. It can also contain MATLAB program files for your toolbox, or you can locate your code files in a different folder. A typical toolbox folder contains the following kinds of elements:

Note Folders and file names that you specify are in italics in this listing. Folders are prefixed with a forward slash (/). On the right side, comments that are in italics are directives for you to follow.

<i>/mytoolbox</i>	Top level folder for toolbox documentation; can also contain your program files.
<i>info.xml</i>	Indicates to MATLAB that this folder contains Help browser documentation, and points to content. <i>Required; must have this file name.</i>
<i>*.m</i>	Program code, data, GUI, and other files for your toolbox. You can also locate program files in a separate folder or a subfolder of this one that you place on the search path. Also include a <code>Contents.m</code> file here summarizing the program files.
<i>*.mat</i>	
<i>*.fig</i>	
<i>...</i>	

<code>/html</code>	Optional subfolder for your HTML documentation content; it can have any name, which must be specified in your <code>info.xml</code> file.
<code>helptoc.xml</code>	Defines hierarchy of help files. <i>Required; must have this file name.</i>
<code>mytoolbox_</code> <code>product_page.html</code>	Roadmap (start page) for your documentation. <i>Use folder name followed by "_product_page.html". Optional but recommended.</i>
<code>getting_started_1.html</code> ... <code>getting_started_n.html</code>	Optional content for getting started guide.
<code>user_guide_1.html</code> ... <code>user_guide_n.html</code>	Content for user guide.
<code>helpfuncbycat.html</code>	Optional functions-by-category listing containing links to function reference HTML files.
<code>release_note_1.html</code> ... <code>release_note_n.html</code>	Release notes files.
<code>/graphics</code>	Optional subfolder for images used in HTML pages; when you use a subfolder, HTML <code><image></code> elements must specify this path to image files.
<code>image_1.png</code> ... <code>image_n.gif</code>	Bitmap graphics files (usually <code>.gif</code> , <code>.png</code> , or <code>.jpg</code>). Do not store custom icons for the TOC here, as they cannot be found.

/reference

Optional subfolder for function or block reference pages; when you use a subfolder, HTML <A> hyperlinks to reference pages must specify this path.

function_1.html

Function/block reference page files.

...

function_n.html

XML Files Required to Add Documentation and Demos

The Help Browser relies on several files coded in XML to recognize and present the contents of documentation and demos. These files always have the same names:

- `info.xml` — Required file that indicates that a folder contains documentation or demos.
- `helptoc.xml` — Required file that provides a structure for presenting the documentation set in the **Contents** pane.
- `demos.xml` — Optional file to add your demos to the **Other Demos** section of the **Contents** pane. See “Add Demos to the Help Browser” on page 22-48.

In addition, you must create and provide the HTML content pages referenced by these files and graphic files for images that they display.

Identifying a Help Folder: the info.xml File

The `info.xml` file specifies the content type, name, and icon to display for your documentation set. It also identifies where to find your HTML help files, and defines items you add to the **Start** button. You must create a file named `info.xml` for each toolbox you document. When you include a file having this name in a folder and then add that folder to the search path, MATLAB adds the documentation for your toolbox to the Help browser **Contents** pane. The folder that `info.xml` identifies as `<help_location>` must contain your HTML documentation and a file named `helptoc.xml`.

The following listing is a template for `info.xml` that you can adapt to describe your toolbox:

```

<productinfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="optional">
  <?xml-stylesheet type="text/xsl"href="optional"?>
  <!-- info.xml file for the mytoolbox toolbox -->
  <!-- Version 1.0 -->
  <!-- Copyright (date) (owner)...>

  <!-- Supply the following six elements in the order specified -->
  <!-- (Required) element; matlabrelease content is not currently used -->
  <matlabrelease>2010a</matlabrelease>
  <!-- (Required) The name element appears in the Contents pane -->
  <name>MyToolbox</name>
  <!-- (Required) The type element identifies your package; pick one: -->
  <!-- matlab, toolbox, simulink, blockset, links_targets -->
  <type>toolbox</type>
  <!-- (Optional) icon file to display in the Start button -->
  <icon>sampleicon.gif</icon>
  <!-- (Required if you supply help) relative path to help (HTML) folder -->
  <help_location>./HTMLfolderName</help_location>
  <!-- (Required if you supply help) icon used in the Help browser TOC -->
  <help_contents_icon>$toolbox/matlab/icons/bookicon.gif</help_contents_icon>

  <!-- - - - - - Start menu - - - - - -->
  <!-- Optional list of entries to display on Start Menu -->
  <!-- Callback items are function calls or commands for toolbox -->
  <!-- Refresh the Start button to see your entries -->
  <!-- Remove this list if you do not want a Start button entry -->
  <list>
    <listitem>
      <!-- The label provides the text for this menu item -->
      <label>MyToolbox Documentation</label>
      <!-- This callback is a command to open your documentation -->
      <callback>
web ./html/mytoolbox_product_page.html -helpbrowser
      </callback>
      <!-- Menu item icon (a toolbox icon from the help browser ) -->
      <icon>$toolbox/matlab/icons/bookicon.gif</icon>
    </listitem>
    <listitem>
      <!-- A menu item label for a opening a GUI -->

```

```

        <label>MyToolbox GUI</label>
        <!-- A command to open a GUI, if any -->
        <callback>mytoolboxgui</callback>
        <!-- The icon for this menu item -->
        <icon>$toolbox/matlab/icons/figureicon.gif</icon>
    </listitem>
    <listitem>
        <!-- A menu item label for a opening a demo -->
        <label>MyToolbox Demo</label>
        <!-- A command to open a demo if any -->
        <callback>mytoolboxdemo</callback>
        <!-- The icon for this menu item -->
        <icon>HelpIcon.DEMOS</icon>
    </listitem>
    <listitem>
        <!-- Include call to open your Web site, if any -->
        <label>MyToolbox Web Site</label>
        <callback>web http://www.mathworks.com -browser</callback>
        <icon>$docroot/techdoc/matlab_env/examples/webicon.gif</icon>
    </listitem>
    <!-- Add listitems for other features of your toolbox... -->
</list>
<!-- - - - - - Start menu - - - - - -->
</productinfo>

```

Note To avoid XML validation errors, include all required elements in the order specified by the template file. If you are not adding a toolbox to the **Start** button menu, omit the lines bracketed by

```

<!-- - - - - - Start menu - - - - - -->

```

For more information, see “Address Validation Errors for info.xml Files” on page 22-57

Replace the contents within the <matlabrelease>, <name>, <type>, and <help_location> elements with appropriate text for your toolbox. The contents of <help_location> is folder name, which usually includes a relative path. Typically, you place the help folder within the folder containing the

`info.xml` file. You can include comments in `info.xml` or any other XML file. For example, you can add copyright and contact information. Lines starting with `<!--` and ending with `-->` contain comments.

When you add a help folder to the Help browser for the first time, take the following actions:

1 Add your toolbox or blockset folder to the search path

Make sure that the folder you are adding is *not* your current folder when you perform this step.

2 Open MATLAB Preferences from the **File** menu.

3 Click **Help**, and then select the **All Products** button.

After MATLAB has identified your folder as a toolbox or blockset and displayed it in the Help browser, you can remove products from the Help browser you do not need to show.

Customizing the `info.xml` Template File

To specify and structure your own documentation content, copy, edit, and save the template file, as follows:

1 In the Editor, open the XML template. You can either:

- Copy the preceding listing and paste it into a new blank document.
- Copy the `info_template.xml` template example file to your current folder:

```
copyfile(fullfile(matlabroot,'help','techdoc','matlab_env', ...
    'examples','templates','info_template.xml'),pwd), ...
fileattrib('info_template.xml','+w')
```

or [click here](#) to copy the template. Then, open the copy in the Editor.

2 Save the file as `info.xml` in your toolbox folder. Saving as a `.xml` file enables Editor syntax highlighting.

- 3** Replace italicized text in the listing with your own content.
- 4** If you are not adding any items to the **Start** menu, delete the - - Start menu - - section. If you want to customize the Start menu, you must modify the `listitem` elements. For instructions, see “Add Your Own Toolboxes to the Start Button”.
- 5** Resave the `info.xml` file when you finish making changes.

More About the `info.xml` File

The `info.xml` file adds the HTML help files to the Help browser and items to the **Start** menu. The following table describes the example `info.xml` file provided as a toolbox template. The source file is `matlabroot/help/techdoc/matlab_env/examples/templates/info_template.xml`.

XML Tag	Description	Value in Template	Notes
<code><matlabrelease></code>	Release of MATLAB	R2010a	Required. Not currently parsed, but indicates when you added help files.
<code><name></code>	Title of toolbox	mytoolbox	Required. The name of your toolbox that appears in the Help browser Contents pane.
<code><type></code>	Determines the toolbox location in the Help browser Contents	toolbox	Required. Allowable values: <code>matlab</code> , <code>toolbox</code> , <code>simulink</code> , <code>blockset</code> , <code>links_targets</code> , <code>other</code> . The Upslope Area toolbox example appears with other toolboxes. The entry has the orange book icon used for toolboxes.
<code><icon></code>	Icon for your toolbox help in the Start button	sampleicon.gif	If you add your toolbox to the Start button options and include a help entry there, specify an icon image file. For more information, see the <code><list><listitem></code> description.

XML Tag	Description	Value in Template	Notes
<help_location>	Location of help files	./HTMLfolderName	Name of subfolder containing helptoc.xml and HTML help files you provide for your toolbox. If not a subfolder, specify the path to help_location relative to the info.xml file. If you provide HTML help files for multiple toolboxes, each help_location must be a different folder.
<help_contents_icon>	Icon to display in Help browser Contents pane	\$toolbox/matlab/icons/bookicon.gif	Required if you supply HTML help files.
<list> <listitem> ...	Entries for Start button	various	If you also want your toolbox to appear as a Start button option, add at least one listitem. For details, see “Add Your Own Toolboxes to the Start Button”.

When you set up an XML file, make sure that:

- You include all required entries;
- The entries are in the same order as in the preceding list and in the template;
- File and folder names in the XML exactly match the names of your files and folders and use upper and lower case letters identically.

For examples, look at the info.xml file for any MathWorks product. To view one of these files:

- 1 Select **Start > Desktop Tools > View Start Button Configuration Files** files.

- 2** From the resulting Start Button Configuration Files dialog box, select the product.
- 3** Click **Open** to view the `info.xml` file in the Editor.

Note The `info.xml` files for MathWorks products contain custom constructs and features that externally supplied `info.xml` files cannot implement.

Creating the Table of Contents File: `helptoc.xml`

You must also create a file named `helptoc.xml`. Place this file in the folder containing your HTML documentation files. This folder is designated as `<help_location>` in your `info.xml` file. Within a top-level `<toc>` element, nest `<tocitem>` elements to define the structure of your table of contents. This template for `helptoc.xml` explains its organization:

```
<?xml version='1.0' encoding="utf-8"?>
<toc version="2.0">
<!-- First tocitem specifies top level in Help browser Contents pane -->
<!-- This can be a roadmap page, as shown below, or a content page -->

    <tocitem target="mytoolbox_product_page.html">MyToolbox Toolbox
        <!-- Nest tocitems to create hierarchical entries in Contents-->
        <!-- To include icons, use the following syntax for tocitems: -->
        <!-- <tocitem target="foo.html" image="HelpIcon.NAME"> -->
        <!-- Title-of-Section </tocitem> -->
        <!-- where NAME is one of the following (use capital letters): -->
        <!-- FUNCTION, USER_GUIDE, EXAMPLES, BLOCK, GETTING_STARTED, -->
        <!-- DEMOS, RELEASE_NOTES -->
        <!-- Icon images used for these entries are also stored in -->
        <!-- matlabroot/toolbox/matlab/icons -->
        <!-- A Getting Started Guide usually comes first -->
        <tocitem target="mytbx_gs_top.html" image="HelpIcon.GETTING_STARTED">
            Getting Started with the MyToolbox Toolbox
            <tocitem target="mytbx_reqts_example.html">System Requirements
            </tocitem>
            <tocitem target="mytbx_features_example.html">Features
                <!-- 2nd and lower TOC levels usually have anchor IDs -->
                <tocitem target="mytbx_feature1_example.htm#10187">Feature 1
```



```

        </tocitem>
        <tocitem target="mytbx_feature2_example.htm#10193">Feature 2
        </tocitem>
    </tocitem>
</tocitem>
<!-- User Guide comes next -->
<tocitem target="mytbx_ug_intro.html"
    image="HelpIcon.USER_GUIDE">MyToolbox User Guide
    <tocitem target="mytbx_ch_1.html">Setting Up MyToolbox
    </tocitem>
    <tocitem target="mytbx_ch_2.html">Processing Data
    </tocitem>
    <tocitem target="mytbx_ch_3.html">Verifying MyToolbox outputs
        <tocitem target="mytbx_ch_3a.html">Handling Test Failures
        </tocitem>
    </tocitem>
</tocitem>
<!-- Function reference next -->
<!-- The first file lists all of the functions, categorizing them -->
<tocitem target="function_categories.html">Functions
    <!-- First category, with link to anchor in above page -->
    <tocitem target="function_categories.html#1">First Category
        <!-- Inside category, list its functions alphabetically -->
        <tocitem target="function_1.html">function_1</tocitem>
        <tocitem target="function_2.html">function_2</tocitem>
        <!-- ... -->
    </tocitem>
    <!-- Second category, with link to anchor in above page -->
    <tocitem target="helpfuncbycat.html#2">Second Category</tocitem>
        <!-- Inside category, list its functions alphabetically -->
        <tocitem target="function_3.html">function_3</tocitem>
        <tocitem target="function_4.html">function_4</tocitem>
        <!-- ... -->
    </tocitem>
    <!-- Third category, with link to anchor in above page -->
        <tocitem target="helpfuncbycat.html#3">Third category</tocitem>
        <!-- ... -->
    </tocitem>
</tocitem>
<!-- Optional List of Examples, with hyperlinks to examples in other files -->

```

```
<tocitem target="mytbx_example.html"
        image="HelpIcon.HelpIcon.EXAMPLES">Mytoolbox Examples
</tocitem>
<!-- Optional link or links to your or other Web sites -->
<tocitem target="http://www.mathworks.com"
        image="$toolbox/matlab/icons/webicon.gif">
MyToolbox Web Site (Example only: goes to mathworks.com)
</tocitem>
</tocitem>
</toc>
```

Be sure that file and path names exactly match those of the files and folders they identify and use upper and lower case letters identically. Your `helptoc.xml` can be shorter or longer than the template. The size of the file depends on the structure of your documentation and how many HTML files it contains.

Most tables of contents have two to four hierarchical levels. Lower levels can either specify subheadings within the top-level HTML file or separate HTML files. A `<tocitem>` can link to subheadings by specifying *anchor IDs* for them. For example, this one,

```
<tocitem target="mytbx_feature1_example.html#107">Feature 1</tocitem>
```

specifies a link to the named anchor `#107` within the file `mytbx_feature1_example.html`. Anchor IDs always start with a pound sign (`#`).

Create anchors for referencing headings or other HTML content with `Any content` elements. If your documentation set includes HTML files that are *not* listed in `helptoc.xml`, at least one file found in the table of contents must contain hyperlinks to them, so that readers can find them. For related information, see “Creating Function and Block Category Listings” on page 22-33.

To customize the `helptoc` template file:

- 1 In the Editor, open the `helptoc` XML template. You can either:
 - Copy the preceding listing and paste it into a new blank document.

- Copy the `helptoc_template.xml` template example file to your current folder:

```
copyfile(fullfile(matlabroot,'help','techdoc','matlab_env', ...
    'examples','templates','helptoc_template.xml'),pwd), ...
fileattrib('helptoc_template.xml','+w')
```





or click [here](#) to copy the template. Then, open the copy in the Editor.




- 2 Save the file as `helptoc.xml` in your toolbox folder. Saving as a `.xml` file enables Editor syntax highlighting.
- 3 Replace italicized text in the listing with your own content.
- 4 Resave the `helptoc.xml` file when you finish making changes.

More About the `helptoc.xml` File

The `info.xml` file inserts your toolbox in the alphabetic listing of toolboxes or blocksets in the **Contents** pane. The `helptoc.xml` defines a hierarchy of entries within it. Each `<tocitem>` entry in the `helptoc.xml` file references one of your HTML help files or anchor IDs within that product entry. The `helptoc_template.xml` file that is provided as an example has the structure most toolboxes use.

You can display icons for your **Contents** pane entries within your toolbox. To use standard MathWorks MathWorks Help browser icons, include any of the following icons as `image` attributes for `<tocitem>` elements.

Icon	Use For	Image Tag String
	Getting Started Guides	<code>HelpIcon.GETTING_STARTED</code>
	User Guides	<code>HelpIcon.USER_GUIDE</code>
	Functions	<code>HelpIcon.FUNCTION</code>
	Blocks	<code>HelpIcon.BLOCK</code>

Icon	Use For	Image Tag String
	Examples	HelpIcon.EXAMPLES
	Release Notes	HelpIcon.RELEASE_NOTES
	Demos	HelpIcon.DEMOS

To make your documentation consistent with MathWorks documentation, organize your table of contents entries in the preceding order .

Include the icons as `image` attributes in top-level TOC entries. If you provide a roadmap page, also include icons for second-level TOC entries under the roadmap. Nest `tocitem` entries for the target chapters or pages within each such TOC entries, for example:

```
<tocitem target="get_start_top.html" image="HelpIcon.GETTING_STARTED">About Mytoolbox
  <tocitem target="get_start_capabilities.html"> Capabilities </tocitem>
  <tocitem> ... </tocitem>
  ...
</tocitem>
```

The markup indicates that you have a file called `get_start1.html` that begins a getting started guide. The HTML pages it contains appear next, coded as nested `tocitem` elements.

Creating Function Reference Pages

Unless you prefer to hide a function from your users, provide an HTML reference page for it. If your program (.m) files contain help text, you already have the content you need to add reference pages to the Help browser. If your program files do not yet include help text, consider adding help as a first step. For details, see “Add Help for Your Program Files” on page 22-4.

You can create a reference page in an HTML authoring environment by importing the help text for a function and formatting the text. For example, you need to remove the percent sign (%) character from the beginning of each line of text, and make sure that spaces separate words. You can then format headings, words, phrases, and examples for HTML display. Finally, you can add `image` attributes to display graphics such as GUIs, diagrams, and

graphic output from your code, and hyperlinks to *See also* items and other related documentation.

Consider creating reference pages from within MATLAB. You can use the capability of MATLAB to publish program scripts directly to HTML documents.

To transform help text from a program file into HTML using the `publish` command:

- 1** Copy the help text into a new file, and remove the code that implements the function.
- 2** Save this file as a MATLAB script (which has no initial function declaration).
- 3** Format the help text using *code cell notation*. For more information, see “Overview of Publishing MATLAB Code”.
- 4** Publish the script as an HTML file.

The following listings illustrate such a transformation for the `upslopeArea.m` function from the example Upslope Area toolbox files to a cell script version of its help text, `upslopeArea_help.m`. Find the original program files in the examples folder `matlabroot/help/techdoc/matlab_env/examples/upslope`. See the subfolder `matlabroot/help/techdoc/matlab_env/examples/upslope/html` for the cell-scripted versions and the HTML generated from publishing those files.

After formatting and saving `upslopeArea_help.m`, the command

```
publish upslopeArea_help.m
```

generates a file named `upslopeArea_help.html` in a subfolder. By default, this folder is named `html`, but you can specify a different name for the folder by configuring the `publish` command, as described in “Specify Output Preferences for Publishing”. (Placing all your reference pages in the same folder simplifies accessing them.)

Original Upslope Area Toolbox Function upslopeArea.m file

```
% upslopeArea Upslope area measurements for a DEM
%
% DESCRIPTION
% A = upslopeArea(E, T) computes the upslope area for each pixel of the
% DEM matrix, E. T is the sparse system of linear equations computed
% by flowMatrix; it represents the distribution of flow from pixel to
% pixel. A contains the upslope area for each corresponding pixel of E.
%
% Note: Connected groups of NaN pixels touching the border are treated as
% having no contribution to flow.
%
% REFERENCE
% Tarboton, "A new method for the determination of flow
% directions and upslope areas in grid digital elevation models," Water
% Resources Research, vol. 33, no. 2, pages 309-319, February 1997.
%
% ALGORITHM NOTES
% The Tarboton paper is not very specific about the handling of plateaus. For
% details of how plateaus are handled in this code, see the algorithm notes for
% the function flowMatrix. In particular, see the subfunction
% plateau_flow_weights in flowMatrix.m.
%
% EXAMPLE
%     s = load('milford_ma_dem');
%     E = s.Zc;
%     R = demFlow(E);
%     T = flowMatrix(E, R);
%     A = upslopeArea(E, T);
%     imshow(log(A), [])
%
% See also demFlow, dependenceMap, fillSinks, flowMatrix, postprocessPlateaus.
%
% Steven L. Eddins
% Copyright 2007-2009 The MathWorks, Inc.

function A = upslopeArea(E, T)

requiresIPT(mfilename);
```

```

% Right-side vector is normally all ones, reflecting an equal contribution
% to water flow originating in each pixel.
rhs = ones(numel(E), 1);

% Connected groups of NaN pixels that touch the border do not contribute
% to water volume.
mask = borderNans(E);
rhs(mask(:)) = 0;

A = T \ rhs;
A = reshape(A, size(E));

```

Upslope Area Toolbox Reference Page Script upslopeArea_help.m file

```

%% upslopeArea
% Upslope area measurements for a DEM
%
%% Description
% |A = upslopeArea(E, T)| computes the upslope area for each pixel of the
% DEM matrix, |E|. |T| is the sparse system of linear equations computed
% by |flowMatrix|; it represents the distribution of flow from pixel to
% pixel. |A| contains the upslope area for each corresponding pixel of |E|.
%
% Note: Connected groups of NaN pixels touching the border are treated as
% having no contribution to flow.
%
%% Reference
% Tarboton, "A new method for the determination of flow
% directions and upslope areas in grid digital elevation models," Water
% Resources Research, vol. 33, no. 2, pages 309-319, February 1997.
%
%% Algorithm notes
% The Tarboton paper is not very specific about the
% handling of plateaus. For details of how plateaus are handled in this
% code, see the algorithm notes for the function |flowMatrix|. In
% particular, see the subfunction |plateau_flow_weights| in |flowMatrix.m|.
%
%% Example

```

```
s = load('milford_ma_dem');
E = s.Zc;
R = demFlow(E);
T = flowMatrix(E, R);
A = upslopeArea(E, T);
imshow(log(A), [])

%% See also
% <demFlow_help.html |demFlow|>, <dependenceMap_help.html |dependenceMap|>,
% <fillSinks_help.html |fillSinks|>, <flowMatrix_help.html |flowMatrix|>,
% <postprocessPlateaus_help.html |postprocessPlateaus|>.

%%
% Copyright 2007-2009 The MathWorks, Inc.
```

As you see, the script file, `upslopeArea_help.m`, does not contain the lines of code that implement the function or comments embedded in that code. However, the file does contain code for the example of using the function and all the help text. The *See also* entries to other toolbox functions are hyperlinks, which you manually edit to use the syntax `<function_name_help.html function_name>`.

Published Upslope Area Toolbox Reference Page upslopeArea_help.html File

When you show the published output file with `web(upslopeArea_help.html)`, the beginning of the reference page resembles the following figure.

upslopeArea

Upslope area measurements for a DEM

Contents

- [Description](#)
- [Reference](#)
- [Algorithm notes](#)
- [Example](#)
- [See also](#)

Description

`A = upslopeArea(E, T)` computes the upslope area for each pixel of the DEM matrix, `E`. `T` is the sparse system of linear equations computed by `flowMatrix`; it represents the distribution of flow from pixel to pixel. `A` contains the upslope area for each corresponding pixel of `E`.

Note: Connected groups of NaN pixels touching the border are treated as having no contribution to flow.

Reference

Tarboton, "A new method for the determination of flow directions and upslope areas in grid digital elevation models," *Water Resources Research*, vol. 33, no. 2, pages 309-319, February 1997.

Algorithm notes

The Tarboton paper is not very specific about the handling of plateaus. For details of how plateaus are handled in this code, see the algorithm notes for the function `flowMatrix`. In particular, see the subfunction `plateau_flow_weights` in `flowMatrix.m`.

Example

```
s = load('milford_ma_dem');
E = s.Zc;
R = demFlow(E);
T = flowMatrix(E, R);
A = upslopeArea(E, T);
imshow(log(A), [])
```

Near the end of the published reference page, a screen capture from `imshow` appears, automatically inserted by `publish`.

Creating Function and Block Category Listings

To make your reference pages more useful, also include a **Functions** (or **Blocks**, for blocksets) entry for them in the **Contents** pane of the Help

browser. Expanding one of these entries can display a list of categories. Each category lists the associated functions (or blocks), along with a brief description of the category and descriptions of each function (or block).

If you supply reference help files, you can provide a classified listing of them. HTML help summaries are similar to `Contents.m` files, but display in the Help browser. If you already have a `Contents.m` file that lists all your public functions, you can use it as the basis for creating a categorical listing in HTML. If you do not have a `Contents.m` file, consider creating one to round out your toolbox. For more information, see “Help Summary for Your Program Files (`Contents.m`)” on page 22-6.

To include a Function-by-Category listing, create an HTML page for it. Use the following example to learn how to edit and mark up your `Contents.m` file, and then publish it to HTML. You can name the output file `helpfuncbycat.html`, as shown, or whatever you prefer. Within `Contents.m`, organize your functions or blocks into categories that you define. Each category begins a new cell. When you publish the file, each category displays as a heading and has an anchor ID (from #1 to #n).

The `helptoc_template.xml` file use category names and anchor IDs in `<tocitem>` elements in its reference section. In the template file, the section for function reference includes links to the categorical listing page, category anchors within it, and individual reference pages.

The following example shows the section of the `helptoc.xml` template file that organizes function reference pages. Publishing `helpfuncbycat.m` created anchor IDs #1, #2, #3, ... in output file `helpfuncbycat.html` to which some `<tocitem>` elements refer:

```
<toc version="2.0">
  <!-- ... -->
  <!-- Function reference next -->
  <!-- The first file lists all of the functions, categorizing them -->
  <tocitem target="function_categories.html">Functions
    <!-- First category, with link to anchor in above page -->
    <tocitem target="function_categories.html#1">First Category
      <!-- Inside category, list its functions alphabetically -->
      <tocitem target="function_1.html">function_1</tocitem>
      <tocitem target="function_2.html">function_2</tocitem>
```

```

        <!-- ... -->
    </tocitem>
    <!-- Second category, with link to anchor in above page -->
    <tocitem target="helpfuncbycat.html#2">Second Category</tocitem>
        <!-- Inside category, list its functions alphabetically -->
        <tocitem target="function_3.html">function_3</tocitem>
        <tocitem target="function_4.html">function_4</tocitem>
        <!-- ... -->
    </tocitem>
    <!-- Third category, with link to anchor in above page -->
    <tocitem target="helpfuncbycat.html#3">Third category</tocitem>
        <!-- ... -->
    </tocitem>
</tocitem>
</toc>

```

Tip Copy the preceding XML code and paste it into an editor. Delete any `<tocitem> ... </tocitem>` lines that you do not need, and replace text italicized in the listing with your own content. Then, paste your code into `helptoc.xml`, replacing the template content section for reference pages displayed here.

Italics in the listing indicate strings you need to replace with your own category, file, function, and anchor names and other text. If you place help files for functions or blocks in a subfolder of the one containing your `helptoc.xml` file, include a relative path in the `target` attribute for each `<tocitem>`. For example, if you place function reference pages in a subfolder called `/reference`, you would specify the target as follows:

```
<tocitem target="./reference/function_1.html">function_1</tocitem>
```

Adding Function Category Listings: Upslope Area Toolbox Example

As mentioned previously, a functions-by-category listing works like a `Contents.m` file. The following example shows how the `Contents.m` file for the Upslope Area toolbox example was marked up and published to create an

HTML page that categorizes the toolbox functions and links each function to its reference documentation.

Note If you perform the following procedure, first copy the Upslope Area toolbox Contents.m file to a working folder so you do not overwrite the file or the files upslope_functions_by_cat.m and upslope_functions_by_cat.html that following the example generates.

1 Edit the original Upslope Area toolbox Contents.m file:

```
% Upslope Area Toolbox
% Version 2.0 09-Dec-2009
%
% Requires Image Processing Toolbox(TM).
%
% Flow Direction.
% demFlow          - Downslope flow direction for a DEM
% facetFlow        - Facet flow direction
% flowMatrix       - Linear equations representing water flow
% pixelFlow        - Downslope flow direction for DEM pixels
%
% Preprocessing and Postprocessing.
% borderNans       - Find NaNs connected to DEM border
% fillSinks        - Fill interior sinks in a DEM
% postprocessPlateaus - Replace upslope areas for plateaus with mean value
%
% Hydrological Applications.
% dependenceMap    - Dependence map for water flow in a DEM
% influenceMap     - Influence map for water flow in a DEM
% upslopeArea      - Upslope area measurements for a DEM
%
% Display.
% visDemFlow       - Visualize flow directions in a DEM
% visMap           - Visualize influence or dependence map for a DEM
%
% Data.
% milford_ma_dem.mat - Sample DEM data provided by USGS and distributed
%                   via Geo Community (geoworld.com), a USGS data
```

```

%           distribution partner. The data set is a 1:24,000-scale
%           raster profile digital elevation model. Download the
%           "Milford" file from the "Digital Elevation Models (DEM)
%           - 24K Middlesex County, Massachusetts, United States"
%           page:
%
%           http://data.geocomm.com/catalog/US/61059/526/group4-3.html
%
% natick_ned* - Sample 1/3 arc-second DEM data for a region in Natick,
%             Massachusetts. Downloaded from the The National Map
%             Seamless Server (http://seamless.usgs.gov/index.php).
%
%
% Steven L. Eddins
% Copyright 2007-2009 The MathWorks, Inc.

```

2 In the Editor, mark up `Contents.m` as follows:

- a** Add a top-level heading, *Functions by Category*.
- b** Format the five categories (*Flow Direction*, *Preprocessing and Postprocessing*, *Hydrological Applications*, *Display*, and *Data*) with double percent signs (%%). Doing so turns the sections into code cells, which become section headings in HTML.
- c** Place an asterisk (*) in front of each function name to mark it as a bullet in HTML.
- d** Format each function name as a hyperlink to its own reference page. In the Upslope example, function reference pages were created by extracting the function help text to files called *functionName_help.m*, which were then published as *functionName_help.html*.

The complete markup of `Contents.m` into a functions-by-category source listing looks like the following example:

```

%% Functions by Category
% Upslope Area Toolbox
% Version 2.0 09-Dec-2009

```

```
%  
% Requires Image Processing Toolbox(TM).  
%  
%% Flow Direction  
% * <demFlow_help.html |demFlow|> - Downslope flow direction for a DEM  
% * <facetFlow_help.html |facetFlow|> - Facet flow direction  
% * <flowMatrix_help.html |flowMatrix|> - Linear equations representing water flow  
% * <pixelFlow_help.html |pixelFlow|> - Downslope flow direction for DEM pixels  
%  
%% Preprocessing and Postprocessing  
% * <borderNans_help.html |borderNans|> - Find NaNs connected to DEM border  
% * <fillSinks_help.html |fillSinks|> - Fill interior sinks in a DEM  
% * <postprocessPlateaus_help.html |postprocessPlateaus|> - Replace upslope areas  
%                               for plateaus with mean value  
%  
%% Hydrological Applications  
% * <dependenceMap_help.html |dependenceMap|> - Dependence map for water flow in a DEM  
% * <influenceMap_help.html |influenceMap|> - Influence map for water flow in a DEM  
% * <upslopeArea_help.html |upslopeArea|> - Upslope area measurements for a DEM  
%  
%% Display  
% * <visDemFlow_help.html |visDemFlow|> - Visualize flow directions in a DEM  
% * <visMap_help.html |visMap|> - Visualize influence or dependence map for a DEM  
%  
%% Data  
% * milford_ma_dem.mat - Sample DEM data provided by USGS and distributed  
% via Geo Community (geoworld.com), a USGS data  
% distribution partner. The data set is a 1:24,000-scale  
% raster profile digital elevation model. Download the  
% "Milford" file from the "Digital Elevation Models (DEM)  
% - 24K Middlesex County, Massachusetts, United States"  
% page at http://data.geocomm.com/catalog/US/61059/526/group4-3.html.  
% * natick_ned* - Sample 1/3 arc-second DEM data for a region in Natick,  
% Massachusetts. Downloaded from the The National Map Seamless Server  
% (http://seamless.usgs.gov/index.php).  
%  
%% Source  
% Steven L. Eddins  
% Copyright 2007-2009 The MathWorks, Inc.
```

- 3** Save your formatted file as `upslope_functions_by_cat.m` in your current folder (in this case, called `helptests`).
- 4** Publish the file, and view the resulting HTML file:

```
publish upslope_functions_by_cat.m
ans =
C:\myfiles\upslope\helptests\upslope_functions_by_cat.html

web(ans)
```

Functions by Category

Upslope Area Toolbox Version 2.0 09-Dec-2009

Requires Image Processing Toolbox™.

Contents

- [Flow Direction](#)
- [Preprocessing and Postprocessing](#)
- [Hydrological Applications](#)
- [Display](#)
- [Data](#)
- [Source](#)

Flow Direction

- [demFlow](#) - Downslope flow direction for a DEM
- [facetFlow](#) - Facet flow direction
- [flowMatrix](#) - Linear equations representing water flow
- [pixelFlow](#) - Downslope flow direction for DEM pixels

Preprocessing and Postprocessing

- [borderNans](#) - Find NaNs connected to DEM border
- [fillSinks](#) - Fill interior sinks in a DEM
- [postprocessPlateaus](#) - Replace upslope areas for plateaus with mean value

Hydrological Applications

- [dependenceMap](#) - Dependence map for water flow in a DEM
- [influenceMap](#) - Influence map for water flow in a DEM
- [upslopeArea](#) - Upslope area measurements for a DEM

Display

- [visDemFlow](#) - Visualize flow directions in a DEM
- [visMap](#) - Visualize influence or dependence map for a DEM

Data

- [milford_ma_dem.mat](#) - Sample DEM data provided by USGS and distributed via GeoWorld (http://www.geoworld.com), a USGS data distribution service. The data set is a 1000x1000 pixel file digitized from a 1:25000 scale "1:25000" file.

Making Your HTML Help Files Searchable

If you want the Help browser to include your documentation in its search results, provide a search database for your HTML help files. MATLAB can create a database for you with one command.

The example uses the `info.xml` file for the Upslope Area toolbox with the `help_location` specified as `C:\myfiles\upslope\html`.

To create the database files:

- 1 If you have not already done so, add the folder containing your `info.xml` file to the search path.

For the example, add the `C:\myfiles\upslope` folder to the path.

- 2 Create a searchable database by running

```
builddocsearchdb('full_path_to_help_location')
```

For the example, assuming your help files are in `C:\myfiles\upslope\html`, run:

```
builddocsearchdb ('C:\myfiles\upslope\html')
```

You must use the functional form when you call `builddocsearchdb` (with the folder location in single quotes inside parentheses).

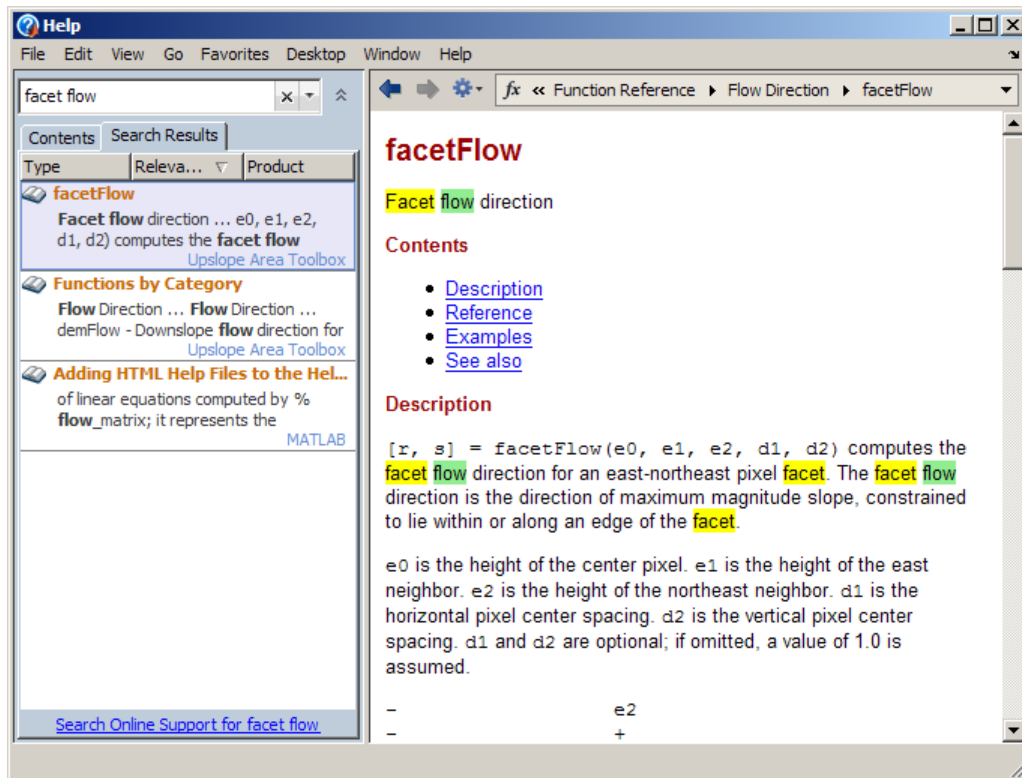
`builddocsearchdb` creates a folder named `helpsearch` in the `help_location` folder. For the example, this command creates the folder `C:\myfiles\upslope\html\helpsearch`.

Each time you run it, `builddocsearchdb` generates three files in `helpsearch`:

- A file called `deletable`.
- A file called `segments`.
- A file having a `cfs` extension with a name that varies.

- 3 To verify that your help files can be searched, use the search field in the Help browser to search for any words in the HTML help files that you provided in the `help_location` folder.

The next figure shows a search of the Upslope Area toolbox and other documentation for the terms `facet` `flow`.



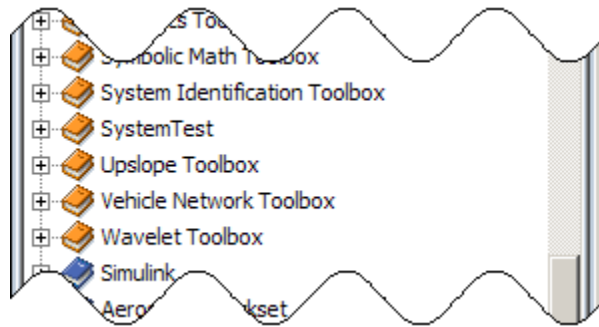
Summary of Workflow for Providing HTML Help Files

To include your HTML help files in the Help browser **Contents** pane, you must create and supply two XML files that the Help browser requires, plus HTML, and image files you develop for your documentation. You must also tell recipients of your software how to install these files. The following steps summarize the steps to take to add your documentation to the Help browser and distribute it to others.

This procedure uses template XML files that you need to customize. To see examples of content and how the files are organized, refer to a complete example of user documentation, the Upslope Area toolbox. To view the Upslope documentation in the Help browser, click here, or run

```
addpath(fullfile(matlabroot, 'help', 'techdoc', 'matlab_env', 'examples', 'upslope'))
```

The Upslope Area Toolbox now appears in the Contents pane (toolboxes are alphabetized), as the following graphic displays. The appearance of your contents pane depends on what products you have installed.



Tip Print or bookmark this page of instructions. Then, when you place your own HTML pages in the Help browser, you can view the instructions at the same time.

- 1 Create or choose a folder for storing your help files. You must have write access to the folder. You can use the same folder that contains your toolbox code.

For the Upslope Area toolbox example, name the folder `upslope`.

- 2 Add an `info.xml` file to the folder. This file identifies the folder as one that contains documentation. To add this file, either click here or follow these two steps:

- a Copy

```
copyfile(fullfile(matlabroot,'help','techdoc','matlab_env',
'examples','templates','info_template.xml'),pwd)
```

to the folder.

For example, copy the file to `upslope`.

- b** Verify that the copied `info_template.xml` file is writable. If it is read-only, make it writable with:

```
fileattrib('info_template.xml','+w')
```

- 3** Rename the copy of `info_template.xml` to `info.xml`. The file must have this name.
- 4** Within your current folder, create a new folder to contain files for the Help browser **Contents**. (The `info.xml` points to this folder as `<help_location>`.)

For the example, in `mytoolbox`, create a folder named `html`.

- 5** Make the new folder your current folder.

For the example, `cd html`.

- 6** Add a `helptoc.xml` file to the empty folder. This file organizes the Help browser table of contents for your toolbox. To add this file, either [click here](#) or follow these two steps:

- a** Copy the `helptoc_template.xml` file to your working directory:

```
copyfile(fullfile(matlabroot,'help','techdoc','matlab_env',  
'examples','templates','helptoc_template.xml'),pwd)
```

to the folder.

For example, copy the file to `mytoolbox/html`.

- b** Verify that the copied `helptoc_template.xml` file is writable. If it is read-only, make it writable with:

```
fileattrib('helptoc_template.xml','+w')
```

- 7** Rename the copy of `helptoc_template.xml` to `helptoc.xml`. It must have this name.
- 8** Move the HTML help files you created, as described in “Summary of Creating and Installing HTML Help Files” on page 22-14, to the folder containing your `helptoc.xml` file. Also move to the folder any files that the HTML files reference, such as image files.

The XML files in the `examples/templates` folder have no accompanying HTML files. You can, however, view files for the example Upslope Area toolbox and learn how its `helptoc.xml` file organizes HTML documentation content. For details, see “Organizing Your Documentation” on page 22-15

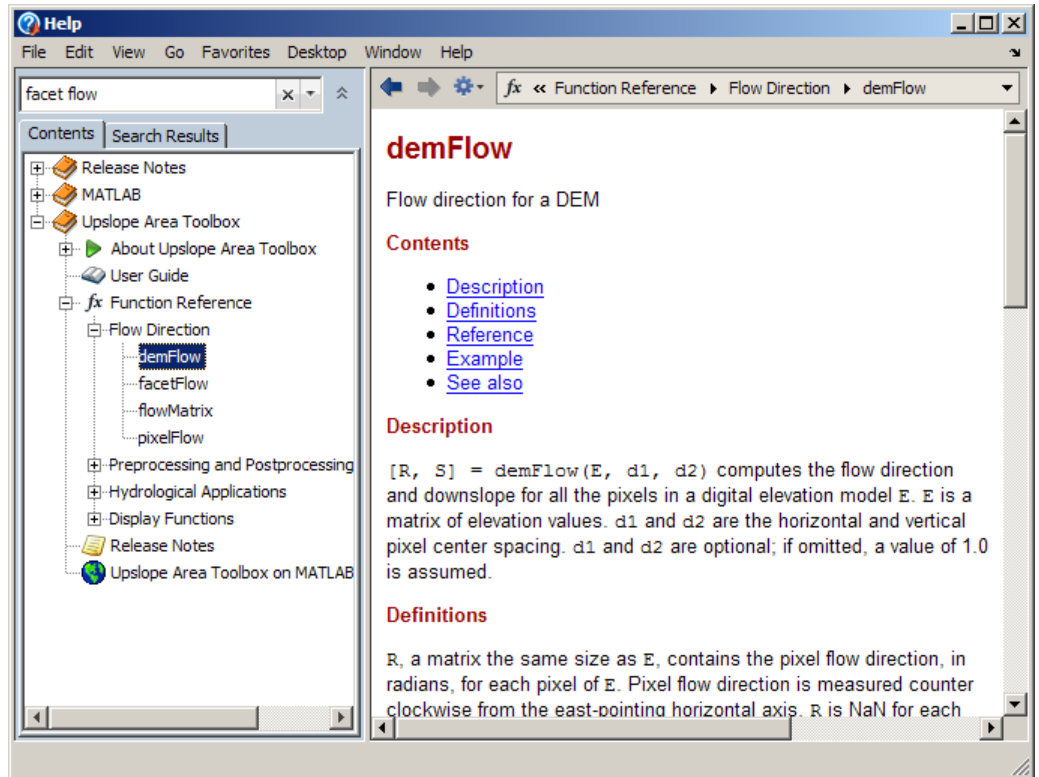
- 9 In the Editor, open, modify, and save your `info.xml`, `helptoc.xml` files.

For details about changes to make, see:

- “More About the `info.xml` File” on page 22-22
 - “More About the `helptoc.xml` File” on page 22-27
 - “Creating Function and Block Category Listings” on page 22-33
- 10 Verify that the Help browser **Filter by Product** preference is set so that your toolbox appears in the display. To set the **Filter by Product** preference:
 - a Access the **Help Preferences** pane by selecting **File > Preferences > Help**.
 - b Under **Filter by Product**, select **All products**.
 - 11 Add the folder to the search path.

For example, add `upslope` to the search path.

- 12 View your HTML help files in the Help browser **Contents** pane.



13 Review the browser display, and verify that there are no errors. MATLAB automatically validates `info.xml` files and reports any problems to the Command Window. For information about addressing the problems, see “Address Validation Errors for `info.xml` Files” on page 22-57.

14 If you provide your documentation to others, make sure that you include all files and folders:

- HTML files.
- Images or other files referenced by the HTML files.
- Your `info.xml` and `helptoc.xml` files.
- Your search database files, if any.

You can use `zip` or `gzip` to create an archive of the folders.

- 15** Instruct recipients of your documentation how to display it. They need to:
 - a** Unzip the archive containing the help files to any disk location they prefer to use, and add the help folder to the search path.
 - b** Verify that your toolbox is selected in the **Filter by Product** Help preferences. Selecting it enables your toolbox to appear in the **Contents** pane of the Help browser.
 - c** If your toolbox still does not appear in the **Contents** pane, suggest removing its folder from the search path and then adding back to the path. The toolbox folder cannot be the current folder during this operation.
- 16** Inform your users which documentation features you support. For example:
 - If you provided search database files, mention that Help browser search results will include your documentation.
 - Alternatively, you can instruct the users to generate a search database with the `builddocsearchdb` function after they set up your files.

Add Demos to the Help Browser

In this section...
“About Creating Demos” on page 22-48
“Providing Demos to Others” on page 22-56

About Creating Demos

You can provide demos for toolboxes you create and make them available in the Help browser. Demos allow you to present the features of your toolbox. Adding your demos to the Help browser is the best way to make them accessible.

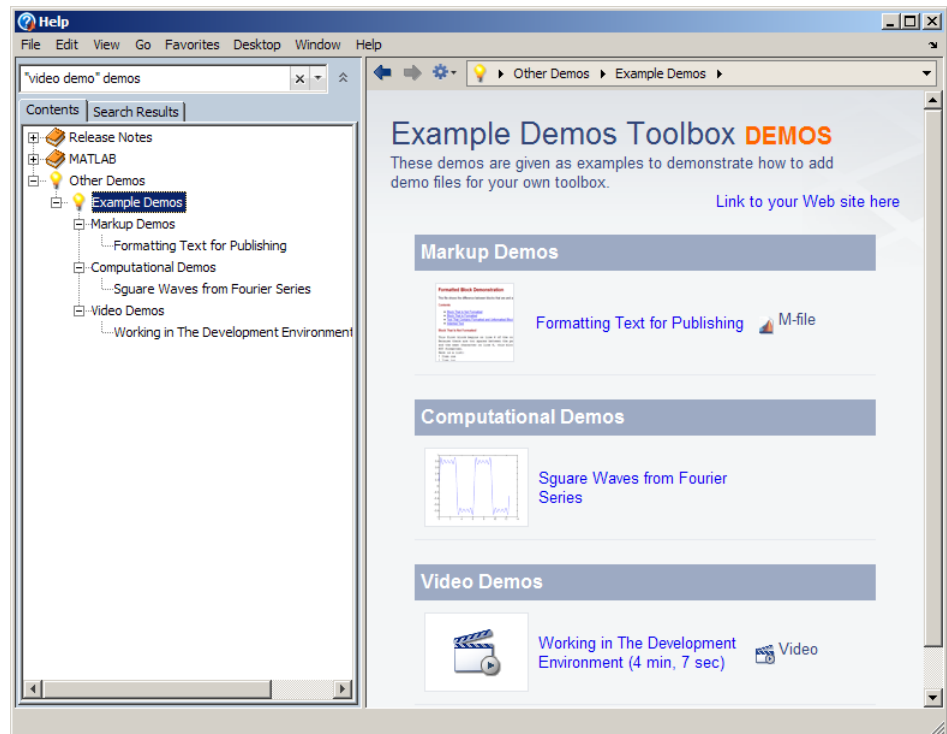
There are no requirements about the types of demos you can provide. However, if you provide the same types of demos that MathWorks products provide, users of your software are already familiar with using them.

This documentation includes an example folder which contains two demos and refers to a third one that comes with MATLAB. [Click here](#) to add this example folder to your search path, or run the following command:

```
addpath(fullfile(matlabroot,'help','techdoc','matlab_env','examples','demo_examples'))
```

The **Contents** pane of the Help browser displays an entry called **Example Demos** under the **Other Demos** entry. As shown in the following figure, within that entry you see three demos:

- Formatting Text for Publishing
- Square Waves from Fourier Series
- A video demo from MATLAB, Working in the Development Environment



To remove the demos you just added, take their folder off the search path:

```
rmpath(fullfile(matlabroot,'help','techdoc','matlab_env','examples','demo_examples'))
```

How to Add Demos

Tip Print or bookmark this page of instructions. Then, when you view your own HTML pages in the Help browser, you can view the instructions at the same time.

After you create a demo, you can access the demo from the Help browser after you perform the following steps:

- 1 Create demos for your toolbox. See “About Creating Demos” on page 22-48.

You can effectively produce MATLAB code demos using the cell-publishing features available in the Editor. Publishing creates an HTML file that includes code, can include figures, describes how to use your code, and enables you to execute the code from the Help browser. For more information, see “Overview of Publishing MATLAB Code”.

- 2 Add the demos files to the Help browser using a special XML file that you create. See “Workflow for Providing Demos” on page 22-50.
- 3 Provide the demo files, along with instructions for including these files in the Help browser. See “Providing Demos to Others” on page 22-56.

The sections that follow refer to a folder of demo examples provided with this documentation. Adapt the contents of that folder to set up your own demos.

Workflow for Providing Demos

To include demos for your toolbox in the Help browser **Contents** pane, you must create and provide a `demos.xml` file and content for your demos. Specify a location to MATLAB where the files will reside:

- 1 Create or choose a folder for storing your demos files. You must have write access to the folder. If you have created a toolbox, the toolbox folder is a good location for storing related demos.

For the example, the folder is `/demo_examples`.

- 2 Create your demo files by publishing code files, constructing a GUI, or another method.

- 3 Put all the demos files you created in the folder.
- 4 Add the folder for your demos files to the search path.

Note The folder cannot be the current folder when you add it to the path or the Help browser will be unable to locate your demos.

- 5 Get the example `demos.xml` file to use as a template for your own file. Click [here](#) to copy that file to your current folder, or
 - a Copy
`matlabroot/help/techdoc/matlab_env/examples/demo_examples/demos.xml`
to the folder for your demos files:

```
copyfile(fullfile(matlabroot, 'help', 'techdoc', 'matlab_env', ...  
            'examples', 'demo_examples', 'demos.xml'), pwd)
```

- b Verify that the copied `demos.xml` file is writable. If it is read-only, make it writable with:

```
fileattrib('demos.xml', '+w')
```

- 6 Edit the content of your copy of `demo.xml`, changing it to describe and point to your demo files. For details, see “More About the `demos.xml` File” on page 22-52.
- 7 View your `demos.xml` file in the Help browser. A new node, `Other Demos`, appears at the bottom of the Help browser **Contents** pane. Expand the node to view the entries you added.
- 8 If the `Other Demos` entry does not appear at the bottom of the Help Navigator, refresh the Help browser. You can refresh in two ways:
 - Right-click on `Demos` in the **Contents** pane, and select **Refresh Demos**. Doing so refreshes all demos on the search path and can take a moment.
 - Remove the folder containing your `demos.xml` file from the search path using `rmpath`. Then, use `addpath` to add your demos folder back on the search path.

Note The Help browser **Filter by Product** preference does not provide an **Other Demos** entry or list the toolbox demos you add. However, the Help browser always shows toolbox demos that you add to the search path.

More About the demos.xml File

Within the `demos.xml` file, the root tag is `<demos>`. This element includes one `<name>`, `<type>`, `<icon>` and `<description>` for the main demo page for your toolbox.

Include a `<demoitem>` for each demo you add. Provide multiple categories of demos by including a `<demosection>` for each category. Put `<demoitem>` entries within that category. If you include *any* categories, then *all* demos must be in categories. In other words, if there is even one `<demosection>`, then all `<demoitem>` tags must be within `<demosection>` tags.

Step 5 of the previous procedure tells how to obtain the example `demos.xml` file. This example contains the following XML code:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Example demos.xml file for adding demos to the Help browser -->
<!-- Your version of this file must be named "demos.xml" -->
<demos>
  <!-- Top-level Demo title in TOC -->
  <name>Example Demos</name>
  <type>toolbox</type>
  <icon>HelpIcon.DEMOS</icon>
  <description>These demos are given as examples to
demonstrate how to add demo files for your own toolbox.
</description>
  <website>
  <a href="http://www.mathworks.com">Link to your Web site here</a>
  </website>
  <!-- First group of demos begins here -->
  <demosection>
    <label>Markup Demos</label>
    <!-- First demo begins here -->
    <demoitem>
```

```

        <!-- How demo is described in the Contents pane -->
        <label>Formatting Text for Publishing</label>
        <!-- Type adds a system icon and this name next to demo item -->
        <type>M-file</type>
        <!-- File to display in the Viewing pane -->
        <file>./html/formatted_block_demo.html</file>
        <!-- Supply optional thumbnail for demo as a .png file -->
        <!-- Name it <demo_name>.png -->
        <!-- for this demo it is ./html/formatted_block_demo.png -->
    </demoitem>
</demosection>
<!-- Second group of demos begins here -->
<demosection>
    <label>Computational Demos</label>
    <demoitem>
        <!-- How demo is described in the Contents pane -->
        <label>Square Waves from Fourier Series</label>
        <!-- Do not add a <type> element if demo is executable -->
        <!-- File to execute for "Run this demo" -->
        <callback>fourier_demo</callback>
        <!-- File to display in the Viewing pane -->
        <file>./html/fourier_demo2.html</file>
    </demoitem>
</demosection>
<!-- Third group of demos begins here -->
<demosection>
    <label>Video Demos</label>
    <demoitem>
        <!-- Type adds a system icon and this name next to demo item -->
        <type>video</type>
        <!-- How demo is described in the Contents pane -->
        <!-- This is an actual MATLAB Flash video demo -->
        <!-- If Flash is installed, it runs in your system browser -->
        <label>Working in The Development Environment (4 min, 7 sec)</label>
        <!-- Command or file to execute for "Run this demo" -->
        <callback>
            playbackdemo('WorkingInTheDevelopmentEnvironment',...
                'toolbox/matlab/web/demos');
        </callback>
    </demoitem>

```

```

        </demosection>
    </demos>

```

Lines starting with `<!--` and ending with `-->` are comments. The code contains two `<demosection>` items, each containing one `<demoitem>`. The first demo consists of HTML documentation only (a `<file>` element). The second one has both HTML documentation and MATLAB code that the reader can execute (a `<callback>` element). When you include a `callback` element, it must contain an executable command. The reader can execute that command by clicking **Run this demo** at the top of the demo page.

The next table describes the `demos.xml` file listed above, and found in the folder `matlabroot/help/techdoc/matlab_env/examples/demo_examples`.

Line	XML Tag	Notes	Value for Example
4	<code><demos></code>	The root element for a <code>demos.xml</code> file.	No value
6	<code><name></code>	Name of your toolbox or collection of demos that displays under Other Demos in the Help browser.	Example Demos
7	<code><type></code>	The product type. Allowable values are <code>matlab</code> , <code>simulink</code> , <code>toolbox</code> , <code>blockset</code> , <code>links_targets</code> , <code>M-file</code> , <code>video</code> , or <code>other</code> .	toolbox
8	<code><icon></code>	Icon for your demo. You can use a standard icon or provide a custom icon by specifying a path to the icon relative to the location of the <code>demos.xml</code> file.	HelpIcon.DEMOS
9 to 11	<code><description></code>	The description that appears in the Help browser viewing pane, on the main page for your demos.	Suggested text: “These demos are given as examples to demonstrate how to add demo files for your own toolbox.”

Line	XML Tag	Notes	Value for Example
12 to 14	<code><website> </website></code>	(Optional) Link to a Web site. For example, MathWorks demos include a link near the top, on the right: Product page at mathworks.com. Can appear anywhere before the <code></demos></code> tag.	<code> Link to your Web site here</code>
16	<code><demosession></code>	(Optional) Begins a category of demos. Each category includes a <code><label></code> , description, and at least one <code><demositem></code> . Use any number of categories.	No value required
17	<code><label></code>	Title shown in Help browser for a <code><demosession></code> .	"Markup Demos"
19	<code><demositem></code>	Use one <code><demositem></code> per demo. Contains <code><label></code> and either a <code><callback></code> or a <code><file></code> tag.	No value required
21	<code><label></code>	Title shown for <code>demositem</code> .	"Formatting Text for Publishing" (1st example <code>demositem</code>)
23	<code><file></code>	Name of HTML file describing the demo, typically produced by publish. Specify a relative path from the location of <code>demos.xml</code> .	<code>./html/formatted_block_demo. html</code> (1st example <code>demositem</code>)
33	<code><callback></code>	Name of an executable file or a MATLAB command. This file runs when you click Run this demo on the demo page.	<code>./html/fourier_demo2. html</code> (for 2nd <code>demositem</code> example)
None	<code><dependency></code>	(Optional) Specifies other products required to run the demo, such as another toolbox. The text must match a product name specified in an <code>info.xml</code> file that is on the search path or in the current folder.	Not included

Supplying Thumbnail Images for Demos. If your demo has an HTML file to describe it, you can include a *thumbnail*, a small image typifying the demo. The `demos.xml` file does not specify thumbnail images directly.

To include a thumbnail, you only need to supply a `.png` image file in the same folder as the HTML file for the demo. Keep the image size to within 96-by-64 pixels (width-by-height). Give the `.png` file the same name as the HTML file. Thus, if the `<file>` element for your demo is `./html/formatted_block_demo.html`, then your thumbnail must be named `formatted_block_demo.png` and reside in the same folder.

When you publish a MATLAB script to HTML with the `publish` command or Editor **File** menu item, you get a `.png` thumbnail file in the correct place with the correct name by default.

Providing Demos to Others

Anyone who wants to use your demos needs the files and instructions for using them:

- 1 Provide recipients with a folder containing:
 - Your demo files
 - All data, images or other files referenced by the demo files
 - Your `demos.xml` file
- 2 Instruct recipients to add the folder containing the demos files to the search path.
- 3 Inform recipients that the toolbox demos appear under **Other Demos**, the last entry in the **Contents** pane.

Address Validation Errors for info.xml Files

In this section...

“About XML File Validation” on page 22-57

“Entities Missing or Out of Order in info.xml” on page 22-57

“Unrelated info.xml File” on page 22-58

“Invalid Constructs in info.xml File” on page 22-58

“Outdated info.xml File for a MathWorks Product” on page 22-58

About XML File Validation

When MATLAB finds an `info.xml` file on the search path or in the current folder, it tries to add information to the Help browser or **Start** button, as specified in the `info.xml` file. MATLAB automatically validates the file against the supported schema. If there is an invalid construct in the `info.xml` file, MATLAB displays an error in the Command Window. The error is typically of the form:

```
Warning: File <yourxmlfile.xml> did not validate.
```

```
...
```

An `info.xml` validation error can occur when you start MATLAB, press the **Start** button, or add folders to the search path.

Following, are the primary causes of an XML file validation error and information to address them:

Entities Missing or Out of Order in info.xml

If you do not list required XML elements in the prescribed order, you receive an XML validation error:

Often, errors result from incorrect ordering of XML tags. Correct the error by updating the `info.xml` file contents to follow the guidelines in the MATLAB help documentation.

The message contains a hyperlink to the page you are now reading. For a description of the elements you need in an `info.xml` file and their required ordering, see “More About the `info.xml` File” on page 22-22.

Unrelated info.xml File

Suppose you have a file named `info.xml` that has nothing to do with the MATLAB Help browser or **Start** button. Because this `info.xml` file is an unrelated file, if the file causes an error, the validation error is irrelevant. In this case, the error is not actually causing any problems, so you can safely ignore it. To prevent the error message from reoccurring, rename the offending `info.xml` file, or ensure that the file is not on the search path or in the current folder.

Invalid Constructs in info.xml File

If the purpose of the `info.xml` file is to add information to the **Start** button or Help browser, correct the reported problem. Use the message in the error to isolate the problem or use any validator. One validator you can use is from the W3C® at <http://www.w3.org/2001/03/webdata/xsv>. For more information about the structure of the `info.xml` file, consult its schema, located at `matlabroot/sys/namespace/info/v1/info.xsd`.

Outdated info.xml File for a MathWorks Product

If you have an `info.xml` file from a different version of MATLAB, that file could contain constructs that are not valid with your version. To identify an `info.xml` file from another version, look at the full path names reported in the error message. The path usually includes a version number, for example, `... \MATLAB\R14\...`. In this situation, the error is not actually causing any problems, so you can safely ignore the error message. To ensure that the error does not reoccur, remove the offending `info.xml` file, or ensure that the file is not on the search path or in the current folder.

Symbols and Numerics

- () symbol
 - for indexing into an array 2-122
 - for specifying function input arguments 2-122
- [] symbol
 - for argument placeholder 2-127
 - for concatenating arrays 2-126
 - for constructing an array 2-126
 - for specifying function return values 2-126
- { } symbol
 - for constructing a cell array 2-118
 - for indexing into a cell array 2-118
- ! symbol
 - for entering a shell escape function 2-121
- % symbol
 - for specifying character conversions 2-122
 - for writing single-line comments 2-122
 - for writing the H1 help line 14-13
- ' symbol
 - for constructing a character array 2-124
- * symbol
 - for filename wildcards 2-115
- , symbol
 - for separating array indices 2-117
 - for separating array row elements 2-117
 - for separating input or output arguments 2-118
 - for separating MATLAB commands 2-118
- . symbol
 - decimal point 2-119
 - for defining a structure field 2-119
 - for specifying object methods 2-119
- : symbol
 - for converting to a column vector 2-117
 - for generating a numeric sequence 2-116
 - for preserving array shape on assignment 2-117
 - for specifying an indexing range 2-117
- ; symbol

- for separating rows of an array 2-123
- for suppressing command output 2-124
- @ symbol
 - for class folders 2-116
 - for constructing function handles 2-115
- .() symbol
 - for creating a dynamic structure field 2-121
- %{ and %} symbols
 - for writing multiple-line comments 2-123
- .. symbol
 - for referring to a parent folder 2-119
- ... symbol
 - for continuing a command line 2-120

A

- accuracy of calculations 2-13
- addition operator 2-2
- and (function equivalent for &) 2-6
- anonymous functions 15-3
 - changing variables 15-9
 - constructing 15-3
 - evaluating variables 15-8
 - in cell arrays 15-6
 - multiple anonymous functions 15-13
 - passing a function to quad 15-12
 - using space characters in 15-6
 - with no input arguments 15-5
- answer, assigned to ans 2-13
- arguments
 - checking 16-14
 - checking number of 16-2
 - function 14-12
 - memory requirements 21-6
 - order in argument list 16-4
 - order of outputs 16-6
 - parsing 16-17
 - passing 1-13
 - passing variable number 16-4
 - to nested functions 16-11

- arithmetic operators 2-2
- array headers
 - memory requirements 21-4
- arrays
 - cell array of strings 6-7
 - copying 21-3
 - of strings 6-3
 - variable names 1-10
- assert
 - formatting strings 6-10
- B**
- backtrace mode
 - warning control 18-33
- base (numeric), converting 6-31
- binary from decimal conversion 6-31
- blank spaces in MATLAB commands 1-12
- blanks
 - finding in string arrays 6-27
- built-in functions 14-45
 - forcing a built-in call 14-46
 - identifying 14-46
- C**
- caching
 - MATLAB folder 14-16
- callback functions
 - creating 19-15
 - specifying 19-17
- calling MATLAB® functions
 - storing as pseudocode 14-7
- capitalization in MATLAB 1-12
- case conversion 6-35
- case sensitivity in MATLAB 1-12
- cell arrays 8-1
 - creating 8-3
 - of strings 6-7
 - comparing strings 6-26
 - functions 6-8
 - preallocating 20-5
 - with anonymous function elements 15-6
- character arrays
 - categorizing characters of 6-27
 - comparing 6-25
 - comparing values on cell arrays 6-26
 - conversion 6-30
 - converting to cell arrays 6-7
 - converting to numeric 6-32
 - creating 6-2
 - delimiting character 6-28
 - evaluating 2-108
 - finding a substring 6-28
 - functions 6-36
 - functions that create 6-35
 - functions that modify 6-35
 - in cell arrays 6-7
 - scalar 6-26
 - searching and replacing 6-28
 - searching or comparing 6-36
 - token 6-28
 - two-dimensional 6-3
 - using relational operators on 6-26
- characters
 - conversion, in format specification
 - string 6-15
 - corresponding ASCII values 6-33
 - finding in string 6-27
- characters and strings 6-2
- classes 3-2
 - cell arrays 8-1
 - cell arrays of strings 6-7
 - combining unlike classes 11-2
 - complex numbers 4-18
 - determining 6-36
 - floating point 4-7
 - double-precision 4-7
 - single-precision 4-8
 - infinity 4-20

- integers 4-3
- logical 5-2
- NaN 4-20
- numeric 4-2
- precedence 11-2
- classes, Map 10-1 10-4
 - methods 10-5
 - properties 10-4
- classes, matlab
 - overview 13-1
- clear 14-35 21-12
- comma-separated lists 2-100
 - assigning output from 2-102
 - assigning to 2-103
 - FFT example 2-106
 - generating from cell array 2-100
 - generating from structure 2-101
 - usage 2-104
 - concatenation 2-105
 - constructing arrays 2-104
 - displaying arrays 2-105
 - function call arguments 2-105
 - function return values 2-106
- command/function duality 1-13
- comments
 - in code 14-14
 - in scripts and functions 14-10
- comparing
 - strings 6-25
- complex arrays
 - memory requirements 21-7
- complex conjugate transpose operator 2-2
- complex number functions 4-27
- complex numbers 4-18
 - creating 4-18
- computational functions
 - in file 14-10
- computer 2-13
- computer type 2-13
- concatenation
 - of strings 17-29
 - of unlike data types 11-2
- containers, Map 10-1
 - concatenating 10-13
 - constructing objects of 10-6
 - examining contents of 10-9
 - mapping to different types 10-18
 - modifying a copy of 10-16
 - modifying keys 10-16
 - modifying values 10-15
 - reading from 10-11
 - removing keys and values 10-15
 - writing to 10-12
- Contents.m file 14-17
- control statements
 - break 2-17
 - case 2-15
 - conditional control 2-15
 - else 2-15
 - elseif 2-15
 - for 2-17
 - if 2-15
 - loop control 2-17
 - otherwise 2-15
 - switch 2-15
 - while 2-17
- conversion characters in format specification
 - string 6-15
- converting
 - cases of strings 6-35
 - dates 2-19
 - numbers 6-30
 - numeric to string 6-30
 - string to numeric 6-32
 - strings 6-30
- converting numeric and string classes 6-37
- converting numeric and string data types 6-37
- converting numeric to string 6-30
- converting string to numeric 6-32
- cos 14-24

- cputime
 - versus tic and toc 20-3
- creating
 - cell array 8-3
 - strings 6-2
 - timer objects 19-5

D

- data organization
 - structure arrays 7-16
- data types 3-2
 - cell arrays 8-1
 - cell arrays of strings 6-7
 - complex numbers 4-18
 - determining 6-36
 - floating point 4-7
 - double-precision 4-7
 - single-precision 4-8
 - infinity 4-20
 - integers 4-3
 - logical 5-2
 - NaN 4-20
 - numeric 4-2
 - precedence 11-2
- date and time functions 2-38
- dates
 - handling and converting 2-19
- debugging
 - errors and warnings 18-37
- decimal representation
 - to binary 6-31
 - to hexadecimal 6-31
- delaying program execution
 - using timers 19-2
- delimiter in string 6-28
- division operators
 - left division 2-2
 - matrix left division 2-2

- matrix right division 2-2
- right division 2-2
- double-precision matrix 3-3 4-2
- duality, command/function 1-13
- dynamic field names in structure arrays 7-11
- dynamic regular expressions 2-80

E

- editor
 - accessing 14-15
 - for creating files 14-15
- element-by-element organization for structures 7-18
- empty arrays
 - and relational operators 2-4
- eps 2-13
- epsilon 2-13
- equal to operator 2-3
- error
 - formatting strings 6-10
- error handling
 - debugging 18-37
- escape characters
 - in format specification string 6-11
- evaluating
 - string containing MATLAB expression 2-108
- examples
 - checking number of function arguments 16-2
 - for 2-17
 - function 14-25
 - script 14-24
 - vectorization 20-8
 - while 2-17
- expressions
 - involving empty arrays 2-4
 - most recent answer 2-13
 - scalar expansion with 2-3
- external program, running from MATLAB 2-113

F

- field names
 - dynamic 7-11
- filenames
 - wildcards 2-115
- files
 - comments 14-14
 - contents 14-10
 - creating with text editor 14-15
 - kinds 14-9
 - naming 14-9
 - overview 14-10
- find function
 - and subscripting 2-8
- finding
 - substring within a string 6-28
- floating point 4-7
- floating point, double-precision 4-7
 - converting to 4-9
 - creating 4-8
 - maximum and minimum values 4-11
- floating point, single-precision 4-8
 - converting to 4-9
 - creating 4-9
 - maximum and minimum values 4-12
- floating-point functions 4-27
- floating-point numbers
 - largest 2-13
 - smallest 2-13
- floating-point relative accuracy 2-13
- flow control
 - break 2-17
 - case 2-15
 - conditional control 2-15
 - else 2-15
 - elseif 2-15
 - for 2-17
 - if 2-15
 - loop control 2-17
 - otherwise 2-15
 - switch 2-15
 - while 2-17
- folders
 - Contents.m file 14-17
 - help for 14-17
 - MATLAB
 - caching 14-16
 - private functions for 15-35
- for
 - example 2-17
- for loop 2-17
- format for numeric values 4-23
- formatting strings 6-10
 - field width 6-17
 - flags 6-18
 - format operator 6-13
 - precision 6-17
 - setting field width 6-20 to 6-21
 - setting precision 6-20 to 6-21
 - subtype 6-16
 - using identifiers 6-22
 - value identifiers 6-20
- fprintf
 - formatting strings 6-10
- function calls
 - memory requirements 21-6
- function definition line
 - for subfunction 15-33
 - in an file 14-10
 - syntax 14-11
- function handles
 - example 9-13
 - for nested functions 15-21
 - maximum name length 9-4
 - naming 9-4
 - operations on 9-27
 - overview of 9-2
- function types
 - overloaded 15-37
- function workspace 14-29

- functions 14-44
 - arguments
 - passing variable number of 16-4
 - body 14-10 14-14
 - built-in 14-45
 - forcing a built-in call 14-46
 - identifying 14-46
 - calling
 - command syntax 1-13
 - function syntax 1-8 1-13
 - passing arguments 1-8 1-13
 - cell arrays of strings 6-8
 - character arrays 6-36
 - clearing from memory 14-35
 - comments 14-10
 - comparing character arrays 6-36
 - complex number 4-27
 - date and time 2-38
 - example 14-25
 - floating-point 4-27
 - identifying 14-44
 - infinity 4-28
 - integer 4-26
 - logical array 5-6
 - modifying character arrays 6-35
 - multiple output arguments 14-12
 - NaN 4-28
 - numeric and string conversion 6-37
 - numeric to string conversion 6-30
 - output formatting 4-28
 - overloaded 14-46
 - primary 15-33
 - searching character arrays 6-36
 - storing as pseudocode 14-7
 - string to numeric conversion 6-32
 - that determine data type 6-36
 - type identification 4-28
 - types of 14-26
 - anonymous 15-3
 - nested 15-16

- overloaded 15-37
- primary 15-15
- private 15-35
- subfunctions 15-33

G

- global variables 14-32
- greater than operator 2-3
- greater than or equal to operator 2-3

H

- H1 line 14-10 14-13
 - and `help` command 14-10
 - and `lookfor` command 14-10
- `help`
 - and H1 line 14-10
 - creating for program files 22-4
 - file 14-13
- help text 14-10
- hexadecimal, converting from decimal 6-31

I

- imaginary unit 2-13
- `Inf` 2-13
- infinity 4-20
 - functions 4-28
 - represented in MATLAB 2-13
- `info.xml` validation errors 22-57
- `inputParser` class 16-17
- integer functions 4-26
- integers 4-3
 - creating 4-4
 - largest system can represent 2-13
 - smallest system can represent 2-13
- `intmax` 2-13
- `intmin` 2-13

K

keywords
 checking for 17-26

L

large data sets
 memory usage in array storage 21-3
 memory usage in function calls 21-16

less than operator 2-3

less than or equal to operator 2-3

load 21-12

local variables 14-29

logical array functions 5-6

logical class 5-2

logical data types 5-2

logical expressions
 and subscripting 2-8

logical operators 2-4
 bit-wise 2-9
 elementwise 2-5
 short-circuit 2-10

lookfor 14-10 14-13
 and H1 line 14-10

loops
 for 2-17
 while 2-17

lowercase usage in MATLAB 1-12

M

Map class 10-1 10-4
 constructing objects of 10-6
 methods 10-5
 properties 10-4

Map objects 10-1
 concatenating 10-13
 constructing 10-6
 examining contents of 10-9
 mapping to different types 10-18

 modifying a copy of 10-16
 modifying keys 10-16
 modifying values 10-15
 reading from 10-11
 removing keys and values 10-15
 writing to 10-12

MATLAB

programming
 files 14-9
 scripts 14-24
 version 2-13

matrices

 constructing a matrix operations
 constructing 1-3
 double-precision 3-3 4-2
 single-precision 3-3 4-2

matrix 1-3**memory**

 making efficient use of 21-2
 management 21-12
 Out of Memory message 21-23

memory requirements

 array headers 21-4
 for array allocation 21-2
 for complex arrays 21-7
 for copying arrays 21-3
 for creating and modifying arrays 21-2
 for handling variables in 21-2
 for numeric arrays 21-7
 for passing arguments 21-6
 for sparse matrices 21-7

message identifiers

 using with warnings 18-27

methods

 determining which is called 14-37

multiplication operators

 matrix multiplication 2-2
 multiplication 2-2

N

- names
 - superseding 15-34
- NaN 2-13 4-20
 - functions 4-28
 - logical operations on 4-21
- nargin
 - checking input arguments 16-2
 - in nested functions 16-11
- nargout
 - in nested functions 16-11
- nested functions 15-16
 - creating 15-16
 - example — creating a function handle 15-27
 - example — function-generating functions 15-29
 - passing optional arguments 16-11
 - separate variable instances 15-25
 - using function handles with 15-21
 - variable scope in 15-19
- newlines in string arrays 6-27
- not (function equivalent for ~) 2-6
- not a number (NaN) 4-20
- not equal to operator 2-3
- Not-a-Number 2-13
- number of arguments 16-2
- numeric arrays
 - memory requirements 21-7
- numeric classes 4-2
 - conversion functions 6-37
 - converting to char 6-30
 - setting display format 4-23
- numeric data types 4-2
 - conversion functions 6-37
 - setting display format 4-23
- numeric to string conversion
 - functions 6-30

O

- objects
 - definitions of 12-2
 - key concepts 12-8
- online help 14-13
- operator precedence 2-11
 - overriding 2-12
- operators
 - addition 2-2
 - arithmetic 2-2
 - categories 2-2
 - colon 2-2
 - complex conjugate transpose 2-2
 - equal to 2-3
 - greater than 2-3
 - greater than or equal to 2-3
 - left division 2-2
 - less than 2-3
 - less than or equal to 2-3
 - logical 2-4
 - bit-wise 2-9
 - elementwise 2-5
 - short-circuit 2-10
 - matrix left division 2-2
 - matrix multiplication 2-2
 - matrix power 2-2
 - matrix right division 2-2
 - multiplication 2-2
 - not equal to 2-3
 - power 2-2
 - relational 2-3
 - right division 2-2
 - subtraction 2-2
 - transpose 2-2
 - unary minus 2-2
 - unary plus 2-2
- optimization
 - preallocation, array 20-4
 - vectorization 20-8
- or (function equivalent for |) 2-6

- organizing data
 - structure arrays 7-16
- Out of Memory message 21-23
- output arguments 14-12
 - order of 16-6
- output formatting functions 4-28
- overloaded functions 14-46 15-37

P

- pack 21-12
- packages
 - use in references 12-11
- parentheses
 - for input arguments 14-12
 - overriding operator precedence with 2-12
- parsing input arguments 16-17
- percent sign (comments) 14-14
- performance
 - analyzing 20-2
- persistent variables 14-32
- pi 2-13
- plane organization for structures 7-16
- polar 14-24
- power operators
 - matrix power 2-2
 - power 2-2
- preallocation
 - arrays 20-4
 - cell array 20-5
- precedence
 - of class 11-2
 - of data types 11-2
 - operator 2-11
 - overriding 2-12
- primary functions 15-15
- private folder 15-35
- private functions 15-35
- program control
 - break 2-17

- case 2-15
- conditional control 2-15
- else 2-15
- elseif 2-15
- for 2-17
- if 2-15
- loop control 2-17
- otherwise 2-15
- switch 2-15
- while 2-17

- program files
 - creating
 - in MATLAB folder 14-16
 - primary function 15-15
 - subfunction 15-33
 - superseding existing names 15-34
- programs
 - running external 2-113
- pseudocode 14-7

Q

- quit 21-12

R

- realmax 2-13
- realmin 2-13
- regexp 2-92
- regexpi 2-92
- regexprep 2-92
- regexptranslate 2-92
- regular expression operators
 - character representation
 - alarm character (\a) 2-57
 - backslash character (\\) 2-57 2-94
 - backspace character (\b) 2-57
 - carriage return character (\r) 2-57
 - dollar sign (\\$) 2-57
 - form feed character (\f) 2-57

- hexadecimal character (`\x`) 2-57
- horizontal tab character (`\t`) 2-57
- literal character (`\char`) 2-57
- new line character (`\n`) 2-57
- octal character (`\o`) 2-57
- vertical tab character (`\v`) 2-57
- character types
 - match alphanumeric character (`\w`) 2-56
 - match any character (period) 2-54
 - match any characters but these (`[^c1c2c3]`) 2-53
 - match any of these characters (`[c1c2c3]`) 2-55
 - match characters in this range (`[c1-c2]`) 2-56
 - match digit character (`\d`) 2-56
 - match nonalphanumeric character (`\W`) 2-54
 - match nondigit character (`\D`) 2-54
 - match nonwhitespace character (`\S`) 2-53
 - match whitespace character (`\s`) 2-56
- conditional operators
 - if condition, match `expr` (`(?(condition)expr)`) 2-78 2-97
- dynamic expressions
 - pattern matching functions 2-84
 - pattern matching scripts 2-85
 - replacement expressions 2-83
 - string replacement functions 2-87
- logical operators
 - atomic group (`(?>expr)`) 2-58
 - comment (`?#expr`) 2-60
 - grouping and capture (`(expr)`) 2-58
 - grouping only (`(?:expr)`) 2-58
 - match exact word (`\<expr\>`) 2-61
 - match `expr1` or `expr2` (`expr1|expr2`) 2-59
 - match if expression begins string (`^expr`) 2-61
 - match if expression begins word (`\<expr`) 2-61
 - match if expression ends string (`expr$`) 2-61
 - match if expression ends word (`expr\>`) 2-61
 - noncapturing group (`(?:expr)`) 2-58
- lookaround operators
 - match `expr1`, if followed by `expr2` (`expr1(?:=expr2)`) 2-63
 - match `expr1`, if not followed by `expr2` (`expr1(?:!expr2)`) 2-64
 - match `expr2`, if not preceded by `expr1` (`expr1(?:!<expr2)`) 2-66
 - match `expr2`, if preceded by `expr1` (`expr1(?:<=expr2)`) 2-65
- operator summary 2-93
- quantifiers
 - lazy quantifier (`quant?`) 2-70
 - match 0 or 1 instance (`expr?`) 2-69
 - match 0 or more instances (`expr*`) 2-69
 - match 1 or more instances (`expr+`) 2-70
 - match at least `m` instances (`expr{m,}`) 2-68
 - match `m` to `n` instances (`expr{m,n}`) 2-70
 - match `n` instances (`expr{n}`) 2-68
- token operators
 - conditional with named token (`((?<name>s1|s2)`) 2-76
 - create named token (`((?<name>expr)`) 2-76
 - create unnamed token (`((expr)`) 2-71
 - give name to token (`((?<name>expr))`) 2-76
 - if token, match `expr1`, else `expr2` (`((?(token)expr1|expr2)`) 2-78
 - match named token (`\k<name>`) 2-76
 - match `N`th token (`\N`) 2-71
 - replace `N`th token (`$N`) 2-72
 - replace `N`th token (`N`) 2-72

- replace with named token
 - (?<name>) 2-76
 - regular expressions
 - character representation 2-57
 - character types 2-53
 - conditional expressions 2-78
 - dynamic expressions 2-80
 - example 2-81
 - functions
 - regex 2-92
 - regexpi 2-92
 - regexprep 2-92
 - regexprtranslate 2-92
 - introduction 2-40
 - logical operators 2-58
 - lookaround operators 2-61
 - used in logical statements 2-67
 - multiple strings
 - matching 2-91
 - quantifiers 2-68
 - lazy 2-70
 - tokens 2-71
 - example 1 2-73
 - example 2 2-73
 - introduction 2-72
 - named capture 2-76
 - operators 2-71
 - use in replacement string 2-76
 - relational operators 2-3
 - empty arrays 2-4
 - strings 6-26
 - replacing substring within string 6-28
- S**
- save 21-12
 - scalar
 - and relational operators 6-26
 - expansion 2-3
 - string 6-26
 - scheduling program execution
 - using timers 19-2
 - scripts 14-9
 - example 14-24
 - executing 14-24
 - search path
 - files on 15-34
 - shell escape functions 2-113
 - short-circuiting
 - in conditional expressions 2-7
 - operators 2-10
 - single-precision matrix 3-3 4-2
 - smallest value system can represent 2-13
 - source code
 - protecting 14-6
 - (space) character
 - for separating array row elements 2-125
 - for separating function return values 2-125
 - spaces in MATLAB commands 1-12
 - sparse matrices
 - memory requirements 21-7
 - sprintf
 - formatting strings 6-10
 - square brackets
 - for output arguments 14-12
 - starting
 - timers 19-10
 - stopping
 - timers 19-10
 - strcmp 6-25
 - string to numeric conversion
 - functions 6-32
 - strings 6-2
 - comparing 6-25
 - converting to numeric 6-32
 - functions to create 6-35
 - searching and replacing 6-28
 - strings, cell arrays of 6-7
 - strings, formatting 6-10
 - escape characters 6-11

- field width 6-17
- flags 6-18
- format operator 6-13
- precision 6-17
- setting field width 6-20 to 6-21
- setting precision 6-20 to 6-21
- subtype 6-16
- using identifiers 6-22
- value identifiers 6-20
- structure arrays
 - data organization 7-16
 - dynamic field names 7-11
 - element-by-element organization 7-18
 - organizing data 7-16
 - example 7-18
 - plane organization 7-16
- structures
 - field names
 - dynamic 7-11
- subfunctions 15-33
 - accessing 15-34
 - creating 15-33
 - debugging 15-34
 - definition line 15-33
- subscripting
 - with logical expression 2-8
 - with the `find` function 2-8
- substring within a string 6-28
- subtraction operator 2-2
- superseding existing filenames 15-34
- symbols 2-114
 - asterisk * 2-115
 - at sign @ 2-115
 - colon : 2-116
 - comma , 2-117
 - curly braces { } 2-118
 - dot . 2-118
 - dot-dot .. 2-119
 - dot-dot-dot ... 2-119
 - dot-parentheses .() 2-121

- exclamation point ! 2-121
- parentheses () 2-121
- percent % 2-122
- percent-brace %{ and %} 2-123
- plus sign + 2-123
- semicolon ; 2-123
- single quotes ' 2-124
- space character 2-125
- square brackets [] 2-126

T

- tabs in string arrays 6-27
- tic and toc
 - versus `cputime` 20-3
- time and date functions 2-38
- timer objects
 - blocking the command line 19-12
 - callback functions 19-14
 - creating 19-5
 - deleting 19-5
 - execution modes 19-19
 - finding all existing timers 19-24
 - naming convention 19-6
 - overview 19-2
 - properties 19-7
 - starting 19-10
 - stopping 19-10
- timers
 - starting and stopping 19-10
 - using 19-2
- tips, programming
 - additional information 17-53
 - command and function syntax 17-3
 - debugging 17-21
 - development environment 17-10
 - evaluating expressions 17-32
 - files and filenames 17-45
 - function arguments 17-15
 - functions 17-12

- help 17-6
- input/output 17-48
- MATLAB path 17-34
- operating system compatibility 17-51
- program control 17-38
- program development 17-18
- save and load 17-42
- starting MATLAB 17-50
- strings 17-29
- variables 17-25
- token in string 6-28
- tokens
 - regular expressions 2-71
- tolerance 2-13
- transpose operator 2-2
- type identification functions 4-28

U

- unary minus operator 2-2
- unary plus operator 2-2
- uppercase usage in MATLAB 1-12

V

- value
 - largest system can represent 2-13
- varargin 16-4
 - in argument list 16-4
 - in nested functions 16-11
- varargout 16-6
 - in argument list 16-4 16-6
 - in nested functions 16-11
 - packing contents 16-6
- variables 1-3
 - global 14-32
 - lifetime of 14-29
 - local 14-29
 - naming 1-10
 - conflict with function names 1-10

- persistent 14-32
- scope 14-29 to 14-30
 - in nested functions 14-31
- storage in memory 21-2
- vector
 - preallocation 20-4
- vectorization 20-8
 - example 20-8
- vectors 1-3
- verbose mode
 - warning control 18-33
- version 2-13
 - obtaining 2-13

W

- warning
 - formatting strings 6-10
- warning control 18-25
 - backtrace, verbose modes 18-33
 - saving and restoring state 18-32
- warning control statements
 - message identifiers 18-27
 - output from 18-30
 - output structure array 18-31
- warnings
 - debugging 18-37
 - identifying 18-24
 - syntax 18-26
 - warning control statements 18-27
 - warning states 18-27
- which 14-37
- while
 - example 2-17
- while loop 2-17
- white space
 - finding in string 6-27
- whos
 - interpreting memory use 21-12
- wildcards, in filenames 2-115

workspace
of individual functions 14-29

X

XML: file validation 22-57