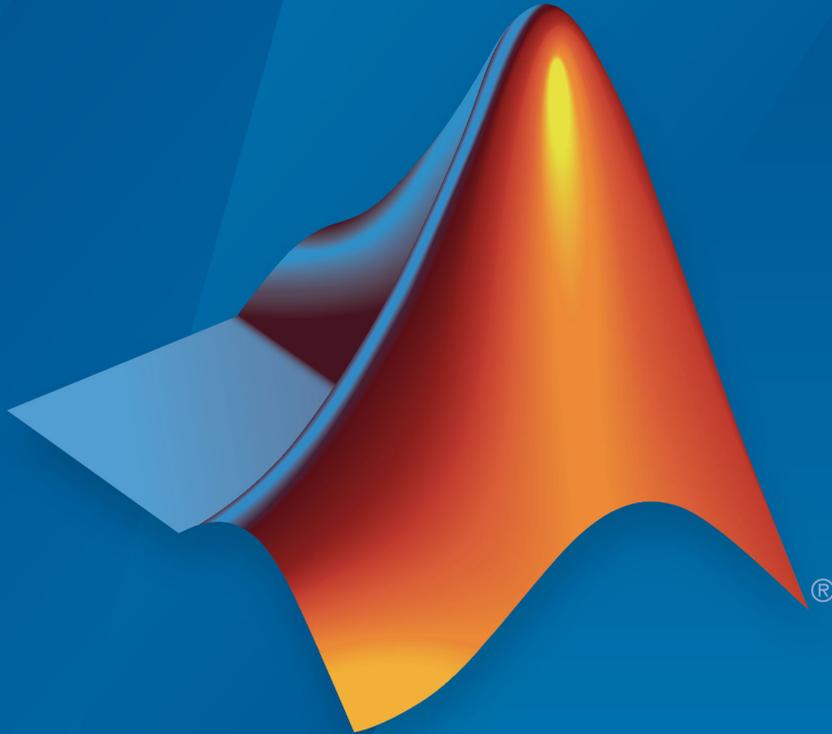


MATLAB®

Programming Fundamentals



MATLAB®

R2018b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

MATLAB Programming Fundamentals

© COPYRIGHT 1984–2018 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)
September 2012	Online only	Revised for Version 8.0 (Release 2012b)
March 2013	Online only	Revised for Version 8.1 (Release 2013a)
September 2013	Online only	Revised for Version 8.2 (Release 2013b)
March 2014	Online only	Revised for Version 8.3 (Release 2014a)
October 2014	Online only	Revised for Version 8.4 (Release 2014b)
March 2015	Online only	Revised for Version 8.5 (Release 2015a)
September 2015	Online only	Revised for Version 8.6 (Release 2015b)
October 2015	Online only	Rereleased for Version 8.5.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 9.0 (Release 2016a)
September 2016	Online only	Revised for Version 9.1 (Release 2016b)
March 2017	Online only	Revised for Version 9.2 (Release 2017a)
September 2017	Online only	Revised for Version 9.3 (Release 2017b)
March 2018	Online only	Revised for Version 9.4 (Release 2018a)
September 2018	Online only	Revised for Version 9.5 (Release 2018b)

Language

1	Syntax Basics	
	Continue Long Statements on Multiple Lines	1-2
	Call Functions	1-3
	Ignore Function Outputs	1-4
	Variable Names	1-5
	Valid Names	1-5
	Conflicts with Function Names	1-5
	Case and Space Sensitivity	1-7
	Command vs. Function Syntax	1-9
	Command and Function Syntaxes	1-9
	Avoid Common Syntax Mistakes	1-10
	How MATLAB Recognizes Command Syntax	1-11
	Common Errors When Calling Functions	1-13
	Conflicting Function and Variable Names	1-13
	Undefined Functions or Variables	1-13

MATLAB Operators and Special Characters	2-2
Arithmetic Operators	2-2
Relational Operators	2-2
Logical Operators	2-3
Special Characters	2-3
String and Character Formatting	2-10
Array vs. Matrix Operations	2-13
Introduction	2-13
Array Operations	2-13
Matrix Operations	2-16
Compatible Array Sizes for Basic Operations	2-19
Inputs with Compatible Sizes	2-19
Inputs with Incompatible Sizes	2-22
Examples	2-23
Array Comparison with Relational Operators	2-25
Array Comparison	2-25
Logic Statements	2-28
Operator Precedence	2-29
Precedence of AND and OR Operators	2-29
Overriding Default Precedence	2-30
Average Similar Data Points Using a Tolerance	2-31
Group Scattered Data Using a Tolerance	2-34
Conditional Statements	2-37
Loop Control Statements	2-39
Regular Expressions	2-41
What Is a Regular Expression?	2-41
Steps for Building Expressions	2-43
Operators and Characters	2-46

Lookahead Assertions in Regular Expressions	2-57
Lookahead Assertions	2-57
Overlapping Matches	2-58
Logical AND Conditions	2-58
Tokens in Regular Expressions	2-60
Introduction	2-60
Multiple Tokens	2-62
Unmatched Tokens	2-62
Tokens in Replacement Text	2-63
Named Capture	2-64
Dynamic Regular Expressions	2-66
Introduction	2-66
Dynamic Match Expressions — (??expr)	2-67
Commands That Modify the Match Expression — (??	
@cmd)	2-68
Commands That Serve a Functional Purpose — (?	
@cmd)	2-69
Commands in Replacement Expressions — \${cmd} ...	2-71
Comma-Separated Lists	2-74
What Is a Comma-Separated List?	2-74
Generating a Comma-Separated List	2-74
Assigning Output from a Comma-Separated List	2-76
Assigning to a Comma-Separated List	2-77
How to Use the Comma-Separated Lists	2-79
Fast Fourier Transform Example	2-81
Alternatives to the eval Function	2-83
Why Avoid the eval Function?	2-83
Variables with Sequential Names	2-83
Files with Sequential Names	2-84
Function Names in Variables	2-85
Field Names in Variables	2-85
Error Handling	2-86

Classes (Data Types)

Overview of MATLAB Classes

3

Fundamental MATLAB Classes	3-2
---	------------

Numeric Classes

4

Integers	4-2
Integer Classes	4-2
Creating Integer Data	4-3
Arithmetic Operations on Integer Classes	4-4
Largest and Smallest Values for Integer Classes	4-5
Floating-Point Numbers	4-7
Double-Precision Floating Point	4-7
Single-Precision Floating Point	4-7
Creating Floating-Point Data	4-8
Arithmetic Operations on Floating-Point Numbers	4-9
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
Complex Numbers	4-17
Creating Complex Numbers	4-17
Infinity and NaN	4-18
Infinity	4-18
NaN	4-18
Identifying Numeric Classes	4-21

Display Format for Numeric Values	4-22
Default Display	4-22
Display Format Examples	4-22
Setting Numeric Format in a Program	4-23

The Logical Class

5

Find Array Elements That Meet a Condition	5-2
Determine if Arrays Are Logical	5-7
Identify Logical Matrix	5-7
Test an Entire Array	5-8
Test Each Array Element	5-8
Summary Table	5-9
Reduce Logical Arrays to Single Value	5-11
Truth Table for Logical Operations	5-14

Characters and Strings

6

Represent Text with Character and String Arrays	6-2
Create Character Arrays	6-6
Create Character Vector	6-6
Create Rectangular Character Array	6-7
Identify Characters	6-9
Work with Space Characters	6-10
Expand Character Arrays	6-11
Create String Arrays	6-12
Cell Arrays of Character Vectors	6-21
Convert to Cell Array of Character Vectors	6-21
Functions for Cell Arrays of Character Vectors	6-22

Analyze Text Data with String Arrays	6-24
Test for Empty Strings and Missing Values	6-31
Formatting Text	6-36
Fields of the Formatting Operator	6-36
Setting Field Width and Precision	6-42
Restrictions on Using Identifiers	6-45
Compare Text	6-47
Search and Replace Text	6-54
Convert from Numeric Values to Character Array	6-61
Function Summary	6-61
Convert Numbers to Character Codes	6-62
Represent Numbers as Text	6-62
Convert to Specific Radix	6-62
Convert from Character Arrays to Numeric Values	6-63
Function Summary	6-63
Convert from Character Code	6-64
Convert Text that Represents Numeric Values	6-64
Convert from Specific Radix	6-65
Frequently Asked Questions About String Arrays	6-66
Why Does Using Command Form With Strings Return An Error?	6-66
Why Do Strings in Cell Arrays Return an Error?	6-67
Why Does length() of String Return 1?	6-68
Why Does isempty("") Return 0?	6-69
Why Does Appending Strings Using Square Brackets Return Multiple Strings?	6-70
Update Your Code to Accept Strings	6-72
What Are String Arrays?	6-72
Recommended Approaches for String Adoption in Old APIs	6-72
How to Adopt String Arrays in Old APIs	6-74
Recommended Approaches for String Adoption in New Code	6-75
How to Maintain Compatibility in New Code	6-77
How to Manually Convert Input Arguments	6-78

How to Check Argument Data Types	6-78
Terminology for Character and String Arrays	6-80
Function Summary	6-82

Dates and Time

7

Represent Dates and Times in MATLAB	7-2
Specify Time Zones	7-6
Convert Date and Time to Julian Date or POSIX Time	7-8
Set Date and Time Display Format	7-12
Formats for Individual Date and Duration Arrays	7-12
datetime Display Format	7-12
duration Display Format	7-13
calendarDuration Display Format	7-14
Default datetime Format	7-15
Generate Sequence of Dates and Time	7-17
Sequence of Datetime or Duration Values Between Endpoints with Step Size	7-17
Add Duration or Calendar Duration to Create Sequence of Dates	7-19
Specify Length and Endpoints of Date or Duration Sequence	7-21
Sequence of Datetime Values Using Calendar Rules	7-22
Share Code and Data Across Locales	7-25
Write Locale-Independent Date and Time Code	7-25
Write Dates in Other Languages	7-26
Read Dates in Other Languages	7-27
Extract or Assign Date and Time Components of Datetime Array	7-28
Combine Date and Time from Separate Variables	7-32

Date and Time Arithmetic	7-34
Compare Dates and Time	7-41
Plot Dates and Durations	7-45
Line Plot with Dates	7-45
Line Plot with Durations	7-47
Scatter Plot with Dates and Durations	7-49
Plots that Support Dates and Durations	7-50
Core Functions Supporting Date and Time Arrays	7-52
Convert Between Datetime Arrays, Numbers, and Text	7-53
Overview	7-53
Convert Between Datetime and Character Vectors	7-54
Convert Between Datetime and String Arrays	7-55
Convert Between Datetime and Date Vectors	7-56
Convert Serial Date Numbers to Datetime	7-57
Convert Datetime Arrays to Numeric Values	7-57
Carryover in Date Vectors and Strings	7-59
Converting Date Vector Returns Unexpected Output ...	7-60

Categorical Arrays

8

Create Categorical Arrays	8-2
Convert Text in Table Variables to Categorical	8-7
Plot Categorical Data	8-12
Compare Categorical Array Elements	8-20
Combine Categorical Arrays	8-23
Combine Categorical Arrays Using Multiplication	8-27

Access Data Using Categorical Arrays	8-30
Select Data By Category	8-30
Common Ways to Access Data Using Categorical Arrays	8-30
Work with Protected Categorical Arrays	8-38
Advantages of Using Categorical Arrays	8-43
Natural Representation of Categorical Data	8-43
Mathematical Ordering for Character Vectors	8-43
Reduce Memory Requirements	8-43
Ordinal Categorical Arrays	8-46
Order of Categories	8-46
How to Create Ordinal Categorical Arrays	8-46
Working with Ordinal Categorical Arrays	8-48
Core Functions Supporting Categorical Arrays	8-50

Tables

9

Create and Work with Tables	9-2
Add and Delete Table Rows	9-13
Add, Delete, and Rearrange Table Variables	9-17
Clean Messy and Missing Data in Tables	9-26
Modify Units, Descriptions, and Table Variable Names .	9-33
Add Custom Properties to Tables and Timetables	9-36
Access Data in a Table	9-42
Ways to Index into a Table	9-42
Create Table from Subset of Larger Table	9-44
Create Array from the Contents of Table	9-47
Calculations on Tables	9-51

Split Data into Groups and Calculate Statistics	9-55
Split Table Data Variables and Apply Functions	9-59
Advantages of Using Tables	9-64
Grouping Variables To Split Data	9-71
Grouping Variables	9-71
Group Definition	9-72
The Split-Apply-Combine Workflow	9-72
Missing Group Values	9-73
Changes to DimensionNames Property in R2016b	9-75

Timetables

10

Create Timetables	10-2
Resample and Aggregate Data in Timetable	10-7
Combine Timetables and Synchronize Their Data	10-11
Retime and Synchronize Timetable Variables Using Different Methods	10-18
Select Timetable Data by Row Time and Variable Type	10-24
Clean Timetable with Missing, Duplicate, or Nonuniform Times	10-31
Using Row Labels in Table and Timetable Operations	10-42

11

Create Structure Array	11-2
Access Data in a Structure Array	11-7
Concatenate Structures	11-11
Generate Field Names from Variables	11-13
Access Data in Nested Structures	11-14
Access Elements of a Nonscalar Struct Array	11-16
Ways to Organize Data in Structure Arrays	11-18
Plane Organization	11-18
Element-by-Element Organization	11-20
Memory Requirements for a Structure Array	11-22

12

What Is a Cell Array?	12-2
Create Cell Array	12-3
Access Data in Cell Array	12-5
Add Cells to Cell Array	12-8
Delete Data from Cell Array	12-10
Combine Cell Arrays	12-12
Pass Contents of Cell Arrays to Functions	12-13
Preallocate Memory for Cell Array	12-18

Cell vs. Struct Arrays	12-19
Multilevel Indexing to Access Parts of Cells	12-25

Function Handles

13

Create Function Handle	13-2
What Is a Function Handle?	13-2
Creating Function Handles	13-2
Anonymous Functions	13-4
Arrays of Function Handles	13-4
Saving and Loading Function Handles	13-5
Pass Function to Another Function	13-6
Call Local Functions Using Function Handles	13-8
Compare Function Handles	13-10

Map Containers

14

Overview of Map Data Structure	14-2
Description of Map Class	14-4
Properties of Map Class	14-4
Methods of Map Class	14-5
Create Map Object	14-6
Construct Empty Map Object	14-6
Construct Initialized Map Object	14-6
Combine Map Objects	14-8
Examine Contents of Map	14-9

Read and Write Using Key Index	14-11
Read From Map	14-11
Add Key/Value Pairs	14-12
Build Map with Concatenation	14-13
Modify Keys and Values in Map	14-16
Remove Keys and Values from Map	14-16
Modify Values	14-16
Modify Keys	14-17
Modify Copy of Map	14-17
Map to Different Value Types	14-19
Map to Structure Array	14-19
Map to Cell Array	14-20

Combining Unlike Classes

15

Valid Combinations of Unlike Classes	15-2
Combining Unlike Integer Types	15-3
Overview	15-3
Example of Combining Unlike Integer Sizes	15-3
Example of Combining Signed with Unsigned	15-4
Combining Integer and Noninteger Data	15-5
Combining Cell Arrays with Non-Cell Arrays	15-6
Empty Matrices	15-7
Concatenation Examples	15-8
Combining Single and Double Types	15-8
Combining Integer and Double Types	15-8
Combining Character and Double Types	15-9
Combining Logical and Double Types	15-9

16

Object Behavior	16-2
Two Copy Behaviors	16-2
Handle Object Copy	16-2
Value Object Copy Behavior	16-2
Handle Object Copy Behavior	16-3
Testing for Handle or Value Class	16-6

Defining Your Own Classes

17

Scripts and Functions

Scripts

18

Create Scripts	18-2
Add Comments to Programs	18-4
Run Code Sections	18-7
Divide Your File into Code Sections	18-7
Evaluate Code Sections	18-7
Navigate Among Code Sections in a File	18-9
Example of Evaluating Code Sections	18-9
Change the Appearance of Code Sections	18-12
Use Code Sections with Control Statements and Functions	18-13
Scripts vs. Functions	18-16

Add Functions to Scripts	18-18
Add Local Functions	18-18
Access Help	18-19
Run Code	18-19
Add and Run Sections in Live Scripts	18-20

Live Scripts and Functions

19

What Is a Live Script or Function?	19-2
Differences with Plain Code Scripts and Functions	19-4
Requirements	19-5
Unsupported Features	19-6
 Create Live Scripts in the Live Editor	 19-7
Create Live Script	19-7
Add Code	19-8
Run Code	19-9
Display Output	19-10
Format Text	19-12
 Run Sections in Live Scripts	 19-15
Divide Your File Into Sections	19-15
Evaluate Sections	19-15
 Debug Code in the Live Editor	 19-17
Show Output	19-17
Debug Using Run to Here	19-18
View Variable Value While Debugging	19-20
Pause a Running File	19-21
End Debugging Session	19-21
Step Into Functions	19-22
Add Breakpoints and Run	19-23
 Modify Figures in Live Scripts	 19-26
Explore Data	19-26
Update Code with Figure Changes	19-28
Add Formatting and Annotations	19-29
Add and Modify Multiple Subplots	19-31
Save and Print Figure	19-36

Format Files in the Live Editor	19-38
Autoformatting	19-40
Insert Equations into the Live Editor	19-44
Insert Equation Interactively	19-44
Insert LaTeX Equation	19-47
Add Interactive Controls to a Live Script	19-56
Create Live Functions	19-60
Create Live Function	19-60
Add Code	19-60
Add Help	19-60
Run Code	19-61
Add Help for Live Functions	19-63
Share Live Scripts and Functions	19-67
Live Code File Format (.mlx)	19-69
Benefits of Live Code File Format	19-69
Source Control	19-69
Introduction to the Live Editor	19-71
Use the Live Editor to Accelerate Exploratory Programming	19-76
Create an Interactive Narrative with the Live Editor ..	19-81
Teach with Live Scripts	19-90

Function Basics

20

Create Functions in Files	20-2
Syntax for Function Definition	20-3
Contents of Functions and Files	20-4
End Statements	20-4

Add Help for Your Program	20-6
Run Functions in the Editor	20-8
Base and Function Workspaces	20-10
Share Data Between Workspaces	20-11
Introduction	20-11
Best Practice: Passing Arguments	20-11
Nested Functions	20-12
Persistent Variables	20-12
Global Variables	20-13
Evaluating in Another Workspace	20-14
Check Variable Scope in Editor	20-16
Use Automatic Function and Variable Highlighting ...	20-16
Example of Using Automatic Function and Variable Highlighting	20-17
Types of Functions	20-20
Local and Nested Functions in a File	20-20
Private Functions in a Subfolder	20-21
Anonymous Functions Without a File	20-21
Anonymous Functions	20-24
What Are Anonymous Functions?	20-24
Variables in the Expression	20-25
Multiple Anonymous Functions	20-26
Functions with No Inputs	20-27
Functions with Multiple Inputs or Outputs	20-27
Arrays of Anonymous Functions	20-28
Local Functions	20-30
Nested Functions	20-32
What Are Nested Functions?	20-32
Requirements for Nested Functions	20-32
Sharing Variables Between Parent and Nested Functions	20-33
Using Handles to Store Function Parameters	20-34
Visibility of Nested Functions	20-37
Variables in Nested and Anonymous Functions	20-39

Private Functions	20-41
Function Precedence Order	20-43

Function Arguments

21

Find Number of Function Arguments	21-2
Support Variable Number of Inputs	21-4
Support Variable Number of Outputs	21-6
Validate Number of Function Arguments	21-8
Argument Checking in Nested Functions	21-11
Ignore Function Inputs	21-13
Check Function Inputs with validateattributes	21-14
Parse Function Inputs	21-17
Input Parser Validation Functions	21-21

Debugging MATLAB Code

22

Debug a MATLAB Program	22-2
Set Breakpoint	22-2
Run File	22-3
Pause a Running File	22-4
Find and Fix a Problem	22-4
Step Through File	22-6
End Debugging Session	22-7

Set Breakpoints	22-8
Standard Breakpoints	22-9
Conditional Breakpoints	22-10
Error Breakpoints	22-11
Breakpoints in Anonymous Functions	22-11
Invalid Breakpoints	22-12
Disable Breakpoints	22-13
Clear Breakpoints	22-13
Examine Values While Debugging	22-15
Select Workspace	22-15
View Variable Value	22-15

Presenting MATLAB Code

23

Options for Presenting Your Code	23-2
Publishing MATLAB Code	23-4
Publishing Markup	23-7
Markup Overview	23-7
Sections and Section Titles	23-10
Text Formatting	23-11
Bulleted and Numbered Lists	23-12
Text and Code Blocks	23-13
External File Content	23-14
External Graphics	23-15
Image Snapshot	23-17
LaTeX Equations	23-18
Hyperlinks	23-20
HTML Markup	23-23
LaTeX Markup	23-24
Output Preferences for Publishing	23-27
How to Edit Publishing Options	23-27
Specify Output File	23-28
Run Code During Publishing	23-29
Manipulate Graphics in Publishing Output	23-31
Save a Publish Setting	23-36

Coding and Productivity Tips

24

Open and Save Files in the Editor	24-2
Open Existing Files	24-2
Save Files	24-3
Check Code for Errors and Warnings	24-6
Automatically Check Code in the Editor — Code Analyzer	24-6
Create a Code Analyzer Message Report	24-10
Adjust Code Analyzer Message Indicators and Messages	24-11
Understand Code Containing Suppressed Messages ..	24-14
Understand the Limitations of Code Analysis	24-16
Enable MATLAB Compiler Deployment Messages	24-18
Improve Code Readability	24-20
Indenting Code	24-20
Right-Side Text Limit Indicator	24-22
Code Folding — Expand and Collapse Code Constructs	24-22
Find and Replace Text in Files	24-27
Find Any Text in the Current File	24-27
Find and Replace Functions or Variables in the Current File	24-27
Automatically Rename All Functions or Variables in a File	24-29
Find and Replace Any Text	24-31
Find Text in Multiple File Names or Files	24-31
Function Alternative for Finding Text	24-31
Perform an Incremental Search in the Editor	24-31
Go To Location in File	24-32
Navigate to a Specific Location	24-32
Set Bookmarks	24-35
Navigate Backward and Forward in Files	24-35

Open a File or Variable from Within a File	24-36
Display Two Parts of a File Simultaneously	24-38
Add Reminders to Files	24-41
Working with TODO/FIXME Reports	24-41
MATLAB Code Analyzer Report	24-44
Running the Code Analyzer Report	24-44
Changing Code Based on Code Analyzer Messages	24-46
Other Ways to Access Code Analyzer Messages	24-47
MATLAB Code Compatibility Report	24-48
Generate the Code Compatibility Report	24-48
Programmatic Use	24-50

Programming Utilities

25

Identify Program Dependencies	25-2
Simple Display of Program File Dependencies	25-2
Detailed Display of Program File Dependencies	25-2
Dependencies Within a Folder	25-3
Protect Your Source Code	25-7
Building a Content Obscured Format with P-Code	25-7
Building a Standalone Executable	25-8
Create Hyperlinks that Run Functions	25-10
Run a Single Function	25-11
Run Multiple Functions	25-11
Provide Command Options	25-12
Include Special Characters	25-12
Create and Share Toolboxes	25-13
Create Toolbox	25-13
Share Toolbox	25-21

Exception Handling in a MATLAB Application	26-2
Overview	26-2
Getting an Exception at the Command Line	26-2
Getting an Exception in Your Program Code	26-3
Generating a New Exception	26-4
Capture Information About Exceptions	26-5
Overview	26-5
The MException Class	26-5
Properties of the MException Class	26-7
Methods of the MException Class	26-13
Throw an Exception	26-15
Respond to an Exception	26-17
Overview	26-17
The try/catch Statement	26-17
Suggestions on How to Handle an Exception	26-19
Clean Up When Functions Complete	26-22
Overview	26-22
Examples of Cleaning Up a Program Upon Exit	26-23
Retrieving Information About the Cleanup Routine	26-25
Using onCleanup Versus try/catch	26-26
onCleanup in Scripts	26-27
Issue Warnings and Errors	26-28
Issue Warnings	26-28
Throw Errors	26-28
Add Run-Time Parameters to Your Warnings and Errors	26-29
Add Identifiers to Warnings and Errors	26-30
Suppress Warnings	26-32
Turn Warnings On and Off	26-33

Restore Warnings	26-35
Disable and Restore a Particular Warning	26-35
Disable and Restore Multiple Warnings	26-36
Change How Warnings Display	26-38
Enable Verbose Warnings	26-38
Display a Stack Trace on a Specific Warning	26-38
Use try/catch to Handle Errors	26-40

Program Scheduling

27

Schedule Command Execution Using Timer	27-2
Overview	27-2
Example: Displaying a Message	27-3
Timer Callback Functions	27-5
Associating Commands with Timer Object Events	27-5
Creating Callback Functions	27-6
Specifying the Value of Callback Function Properties ..	27-8
Handling Timer Queuing Conflicts	27-10
Drop Mode (Default)	27-10
Error Mode	27-11
Queue Mode	27-13

Performance

28

Measure Performance of Your Program	28-2
Overview of Performance Timing Functions	28-2
Time Functions	28-2
Time Portions of Code	28-2
The cputime Function vs. tic/toc and timeit	28-3
Tips for Measuring Performance	28-3

Profile to Improve Performance	28-5
What Is Profiling?	28-5
Profiling Process and Guidelines	28-5
Using the Profiler	28-6
Profile Summary Report	28-8
Profile Detail Report	28-9
Use Profiler to Determine Code Coverage	28-12
Techniques to Improve Performance	28-14
Environment	28-14
Code Structure	28-14
Programming Practices for Performance	28-14
Tips on Specific MATLAB Functions	28-15
Preallocation	28-17
Preallocating a Nondouble Matrix	28-17
Vectorization	28-19
Using Vectorization	28-19
Array Operations	28-20
Logical Array Operations	28-22
Matrix Operations	28-23
Ordering, Setting, and Counting Operations	28-24
Functions Commonly Used in Vectorization	28-26

Memory Usage

29

Strategies for Efficient Use of Memory	29-2
Ways to Reduce the Amount of Memory Required	29-2
Using Appropriate Data Storage	29-4
How to Avoid Fragmenting Memory	29-7
Reclaiming Used Memory	29-8
Resolve “Out of Memory” Errors	29-9
General Suggestions for Reclaiming Memory	29-9
Increase System Swap Space	29-10
Set the Process Limit on Linux Systems	29-10
Disable Java VM on Linux Systems	29-10

Free System Resources on Windows Systems	29-11
How MATLAB Allocates Memory	29-12
Memory Allocation for Arrays	29-12
Data Structures and Memory	29-16
Avoid Unnecessary Copies of Data	29-21
Passing Values to Functions	29-21
Why Pass-by-Value Semantics	29-23
Handle Objects	29-24

Custom Help and Documentation

30

Create Help for Classes	30-2
Help Text from the doc Command	30-2
Custom Help Text	30-3
Check Which Programs Have Help	30-9
Create Help Summary Files — Contents.m	30-12
What Is a Contents.m File?	30-12
Create a Contents.m File	30-12
Check an Existing Contents.m File	30-13
Customize Code Suggestions and Completions	30-15
Function Objects	30-16
Signature Objects	30-16
Argument Objects	30-17
Create Function Signature File	30-22
Multiple Signatures	30-24
Display Custom Documentation	30-27
Overview	30-27
Create HTML Help Files	30-28
Create info.xml File	30-29
Create helptoc.xml File	30-31
Build a Search Database	30-33
Address Validation Errors for info.xml Files	30-34

Display Custom Examples	30-37
How to Display Examples	30-37
Elements of the demos.xml File	30-38

Source Control Interface

31

About MathWorks Source Control Integration	31-3
Classic and Distributed Source Control	31-3
 Select or Disable Source Control System	 31-6
Select Source Control System	31-6
Disable Source Control	31-6
 Create New Repository	 31-7
 Review Changes in Source Control	 31-9
 Mark Files for Addition to Source Control	 31-10
 Resolve Source Control Conflicts	 31-11
Examining and Resolving Conflicts	31-11
Resolve Conflicts	31-11
Merge Text Files	31-12
Extract Conflict Markers	31-13
 Commit Modified Files to Source Control	 31-15
 Revert Changes in Source Control	 31-16
Revert Local Changes	31-16
Revert a File to a Specified Revision	31-16
 Set Up SVN Source Control	 31-17
SVN Source Control Options	31-17
Register Binary Files with SVN	31-18
Standard Repository Structure	31-21
Tag Versions of Files	31-21
Enforce Locking Files Before Editing	31-21
Share a Subversion Repository	31-22

Check Out from SVN Repository	31-24
Retrieve Tagged Version of Repository	31-26
Update SVN File Status and Revision	31-28
Refresh Status of Files	31-28
Update Revisions of Files	31-28
Get SVN File Locks	31-29
Manage SVN Repository Locks	31-29
Set Up Git Source Control	31-30
About Git Source Control	31-30
Install Command-Line Git Client	31-31
Register Binary Files with Git	31-32
Add Git Submodules	31-34
Clone from Git Repository	31-36
Troubleshooting	31-37
Update Git File Status and Revision	31-38
Refresh Status of Files	31-38
Update Revisions of Files	31-38
Branch and Merge with Git	31-40
Create Branch	31-40
Switch Branch	31-42
Compare Branches	31-42
Merge Branches	31-42
Revert to Head	31-43
Delete Branches	31-43
Pull, Push and Fetch Files with Git	31-45
Pull and Push	31-45
Fetch and Merge	31-46
Use Git Stashes	31-47
Move, Rename, or Delete Files Under Source Control	31-49
Customize External Source Control to Use MATLAB for Diff and Merge	31-50
MSSCCI Source Control Interface	31-53

Set Up MSSCCI Source Control	31-54
Create Projects in Source Control System	31-54
Specify Source Control System with MATLAB Software	31-56
Register Source Control Project with MATLAB Software	31-57
Add Files to Source Control	31-59
Check Files In and Out from MSSCCI Source Control	31-61
Check Files Into Source Control	31-61
Check Files Out of Source Control	31-62
Undoing the Checkout	31-63
Additional MSSCCI Source Control Actions	31-64
Getting the Latest Version of Files for Viewing or Compiling	31-64
Removing Files from the Source Control System	31-65
Showing File History	31-66
Comparing the Working Copy of a File to the Latest Version in Source Control	31-67
Viewing Source Control Properties of a File	31-69
Starting the Source Control System	31-69
Access MSSCCI Source Control from Editors	31-71
Troubleshoot MSSCCI Source Control Problems	31-72
Source Control Error: Provider Not Present or Not Installed Properly	31-72
Restriction Against @ Character	31-73
Add to Source Control Is the Only Action Available ...	31-73
More Solutions for Source Control Problems	31-74

Unit Testing

32

Write Test Using Live Script	32-3
Write Script-Based Unit Tests	32-7

Write Script-Based Test Using Local Functions	32-14
Extending Script-Based Tests	32-18
Test Suite Creation	32-18
Test Selection	32-19
Programmatic Access of Test Diagnostics	32-20
Test Runner Customization	32-20
Run Tests in Editor	32-22
Write Function-Based Unit Tests	32-26
Create Test Function	32-26
Run the Tests	32-29
Analyze the Results	32-29
Write Simple Test Case Using Functions	32-31
Write Test Using Setup and Teardown Functions	32-36
Extending Function-Based Tests	32-43
Fixtures for Setup and Teardown Code	32-43
Test Logging and Verbosity	32-44
Test Suite Creation	32-45
Test Selection	32-45
Test Running	32-46
Programmatic Access of Test Diagnostics	32-46
Test Runner Customization	32-47
Author Class-Based Unit Tests in MATLAB	32-48
The Test Class Definition	32-48
The Unit Tests	32-48
Additional Features for Advanced Test Classes	32-50
Write Simple Test Case Using Classes	32-52
Write Setup and Teardown Code Using Classes	32-57
Test Fixtures	32-57
Test Case with Method-Level Setup Code	32-57
Test Case with Class-Level Setup Code	32-58
Types of Qualifications	32-61

Tag Unit Tests	32-64
Tag Tests	32-64
Select and Run Tests	32-65
Write Tests Using Shared Fixtures	32-69
Create Basic Custom Fixture	32-73
Create Advanced Custom Fixture	32-76
Create Basic Parameterized Test	32-83
Create Advanced Parameterized Test	32-89
Use External Parameters in Parameterized Test	32-98
Create Simple Test Suites	32-103
Run Tests for Various Workflows	32-106
Set Up Example Tests	32-106
Run All Tests in Class or Function	32-106
Run Single Test in Class or Function	32-107
Run Test Suites by Name	32-108
Run Test Suites from Test Array	32-108
Run Tests with Customized Test Runner	32-108
Programmatically Access Test Diagnostics	32-110
Add Plugin to Test Runner	32-111
Write Plugins to Extend TestRunner	32-114
Custom Plugins Overview	32-114
Extending Test Level Plugin Methods	32-115
Extending Test Class Level Plugin Methods	32-115
Extending Test Suite Level Plugin Methods	32-116
Create Custom Plugin	32-118
Write Plugin to Save Diagnostic Details	32-124
Plugin to Generate Custom Test Output Format	32-129
Analyze Test Case Results	32-133

Analyze Failed Test Results	32-136
Rerun Failed Tests	32-139
Dynamically Filtered Tests	32-143
Test Methods	32-143
Method Setup and Teardown Code	32-146
Class Setup and Teardown Code	32-148
Create Custom Constraint	32-151
Create Custom Boolean Constraint	32-154
Create Custom Tolerance	32-157
Overview of App Testing Framework	32-163
App Testing	32-163
Gesture Support of UI Componets	32-163
Write a Test for an App	32-164
Write Test for App	32-168
Write Test That Uses App Testing and Mocking	
Frameworks	32-173
Create App	32-173
Test App With Manual Intervention	32-174
Create Fully Automated Test	32-175
Overview of Performance Testing Framework	32-179
Determine Bounds of Measured Code	32-179
Types of Time Experiments	32-180
Write Performance Tests with Measurement	
Boundaries	32-181
Run Performance Tests	32-182
Understand Invalid Test Results	32-182
Test Performance Using Scripts or Functions	32-184
Test Performance Using Classes	32-190
Measure Fast Executing Test Code	32-198
Create Mock Object	32-202

Specify Mock Object Behavior	32-211
Define Mock Method Behavior	32-211
Define Mock Property Behavior	32-213
Define Repeating and Subsequent Behavior	32-214
Summary of Behaviors	32-216
Qualify Mock Object Interaction	32-218
Qualify Mock Method Interaction	32-219
Qualify Mock Property Interaction	32-220
Use Mock Object Constraints	32-221
Summary of Qualifications	32-223

System object Usage and Authoring

33

What Are System Objects?	33-2
Running a System Object	33-3
System Object Functions	33-4
System Objects vs MATLAB Functions	33-6
System Objects vs. MATLAB Functions	33-6
Process Audio Data Using Only MATLAB Functions Code ...	33-6
Process Audio Data Using System Objects	33-7
System Design in MATLAB Using System Objects	33-9
System Design and Simulation in MATLAB	33-9
Create Individual Components	33-10
Configure Components	33-10
Create and Configure Components at the Same Time	33-11
Assemble Components Into System	33-12
Run Your System	33-12
Reconfiguring Objects	33-13
Define Basic System Objects	33-15
Create System Object Class	33-15
Define Algorithm	33-15
Change the Number of Inputs	33-18
Validate Property and Input Values	33-22

Initialize Properties and Setup One-Time Calculations	33-25
Set Property Values at Construction Time	33-28
Reset Algorithm and Release Resources	33-30
Reset Algorithm State	33-30
Release System Object Resources	33-30
Define Property Attributes	33-32
Hide Inactive Properties	33-36
Specify Inactive Property	33-36
Complete Class Definition File with Inactive Properties Method	33-36
Limit Property Values to Finite List	33-38
Create a Whiteboard System object	33-39
Process Tuned Properties	33-43
Define Composite System Objects	33-45
Define Finite Source Objects	33-48
Use the FiniteSource Class and Specify End of the Source .	33-48
Complete Class Definition File with Finite Source	33-48
Save and Load System Object	33-50
Save System Object and Child Object	33-50
Load System Object and Child Object	33-50
Complete Class Definition Files with Save and Load	33-51
Define System Object Information	33-54
Handle Input Specification Changes	33-56
React to Input Specification Changes	33-56
Restrict Input Specification Changes	33-56
Summary of Call Sequence	33-59
Setup Call Sequence	33-59
Running the Object or Step Call Sequence	33-60
Reset Method Call Sequence	33-62
Release Method Call Sequence	33-62

Detailed Call Sequence	33-63
setup Call Sequence	33-63
Running the Object or step Call Sequence	33-64
reset Call Sequence	33-64
release Call Sequence	33-64
Tips for Defining System Objects	33-66
General	33-66
Inputs and Outputs	33-66
Using ~ as an Input Argument in Method Definitions	33-66
Properties	33-67
Text Comparisons	33-67
Simulink	33-68
Code Generation	33-68
Insert System Object Code Using MATLAB Editor	33-69
Define System Objects with Code Insertion	33-69
Create a Temperature Enumeration	33-73
Create Custom Property for Freezing Point	33-75
Add Method to Validate Inputs	33-76
Analyze System Object Code	33-77
View and Navigate System object Code	33-77
Example: Go to StepImpl Method Using Analyzer	33-77
Use Global Variables in System Objects	33-80
System Object Global Variables in MATLAB	33-80
System Object Global Variables in Simulink	33-80

Language

Syntax Basics

- “Continue Long Statements on Multiple Lines” on page 1-2
- “Call Functions” on page 1-3
- “Ignore Function Outputs” on page 1-4
- “Variable Names” on page 1-5
- “Case and Space Sensitivity” on page 1-7
- “Command vs. Function Syntax” on page 1-9
- “Common Errors When Calling Functions” on page 1-13

Continue Long Statements on Multiple Lines

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

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

Build a long character vector by concatenating shorter vectors together:

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

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

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

An ellipsis outside a quoted text 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 inputs in single quotation marks:

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

See Also

Related Examples

- “Ignore Function Outputs” on page 1-4

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 1-5

“Conflicts with Function Names” on page 1-5

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:

- “Unexpected Results When Loading Variables Within a Function”
- “Alternatives to the `eval` Function” on page 2-83

See Also

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

Case and Space Sensitivity

MATLAB code is sensitive to casing, and insensitive to blank spaces except when defining arrays.

Uppercase 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 UNIX®¹ 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 the function name from the rest of the text. Some functions for interfacing to Oracle® 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
```

However, blank spaces act as delimiters in horizontal concatenation. When defining row vectors, you can use spaces and commas interchangeably to separate elements:

```
A = [1, 0 2, 3 3]
```

```
A =
```

```
    1    0    2    3    3
```

Because of this flexibility, check to ensure that MATLAB stores the correct values. For example, the statement `[1 sin (pi) 3]` produces a much different result than `[1 sin(pi) 3]` does.

```
[1 sin (pi) 3]
```

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

```
Error using sin
Not enough input arguments.
```

```
[1 sin(pi) 3]
```

```
ans =
```

```
1.0000 0.0000 3.0000
```

Command vs. Function Syntax

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 character vectors (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. Command syntax always passes inputs as character vectors. To use strings as inputs, use the function syntax. If a character vector contains a space, use the function syntax. For example:

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 character vectors 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> , which is equivalent to <code>round([51 46 52 57 57])</code>	<code>round(3.499)</code>
<code>disp hello world</code>	<code>disp('hello','world')</code>	<code>disp('hello world')</code>
<code>disp "string"</code>	<code>disp('"string"')</code>	<code>disp("string")</code>

Passing Variable Names

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

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

requests information about variable `X` in the example 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 element-wise 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.

Common Errors When Calling Functions

In this section...
“Conflicting Function and Variable Names” on page 1-13
“Undefined Functions or Variables” on page 1-13

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:

- “Conflicts with Function Names” on page 1-5
- “Unexpected Results When Loading Variables Within a Function”
- “Alternatives to the eval Function” on page 2-83

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 the Spelling of the Function Name

One of the most common errors is misspelling 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.

If you misspell a MATLAB function, a suggested function name appears in the Command Window. For example, this command fails because it includes an uppercase letter in the function name:

```
accumArray
```

```
Undefined function or variable 'accumArray'.
```

```
Did you mean:
```

```
>> accumarray
```

Press **Enter** to execute the suggested command or **Esc** to dismiss it.

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`,

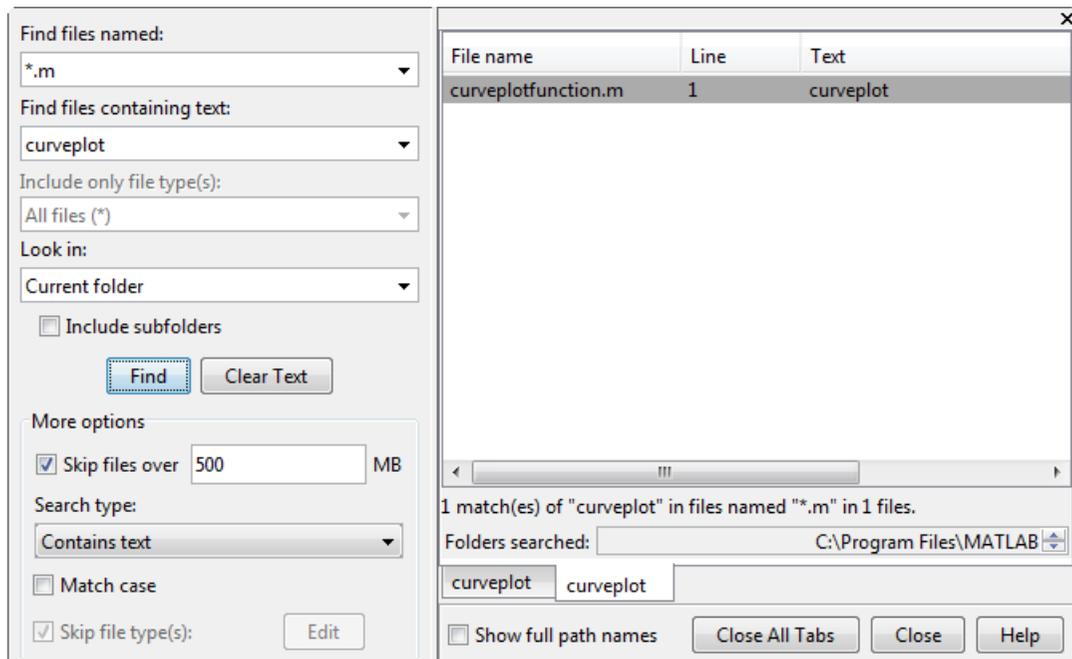
```
function curveplot(xVal, yVal)
    - program code -
```

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 On the **Home** tab, in the **File** section, click  **Find Files**.
- 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, search for the function documentation at <https://www.mathworks.com/help>. The toolbox name appears at the top of the function reference page.

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 On the **Home** tab, in the **Environment** section, click  **Preferences**.
The Preference dialog box appears.
- 2 Under the **MATLAB > General** node, click the **Update Toolbox Path Cache** button.
- 3 On the **Home** tab, in the **Environment** section, click  **Set Path...**.
The Set Path dialog box opens.
- 4 Click **Default**.
A small dialog box opens warning that you will lose your current path settings if you proceed. Click **Yes** if you decide to proceed.

(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.

Program Components

- “MATLAB Operators and Special Characters” on page 2-2
- “Array vs. Matrix Operations” on page 2-13
- “Compatible Array Sizes for Basic Operations” on page 2-19
- “Array Comparison with Relational Operators” on page 2-25
- “Operator Precedence” on page 2-29
- “Average Similar Data Points Using a Tolerance” on page 2-31
- “Group Scattered Data Using a Tolerance” on page 2-34
- “Conditional Statements” on page 2-37
- “Loop Control Statements” on page 2-39
- “Regular Expressions” on page 2-41
- “Lookahead Assertions in Regular Expressions” on page 2-57
- “Tokens in Regular Expressions” on page 2-60
- “Dynamic Regular Expressions” on page 2-66
- “Comma-Separated Lists” on page 2-74
- “Alternatives to the eval Function” on page 2-83

MATLAB Operators and Special Characters

This page contains a comprehensive listing of all MATLAB operators, symbols, and special characters.

Arithmetic Operators

Symbol	Role	More Information
+	Addition	plus
+	Unary plus	uplus
-	Subtraction	minus
-	Unary minus	uminus
.*	Element-wise multiplication	times
*	Matrix multiplication	mtimes
./	Element-wise right division	rdivide
/	Matrix right division	mrdivide
.\	Element-wise left division	ldivide
\	Matrix left division (also known as <i>backslash</i>)	mldivide
.^	Element-wise power	power
^	Matrix power	mpower
.'	Transpose	transpose
'	Complex conjugate transpose	ctranspose

Relational Operators

Symbol	Role	More Information
==	Equal to	eq
~=	Not equal to	ne
>	Greater than	gt

Symbol	Role	More Information
>=	Greater than or equal to	ge
<	Less than	lt
<=	Less than or equal to	le

Logical Operators

Symbol	Role	More Information
&	Logical AND	and
	Logical OR	or
&&	Logical AND (with short-circuiting)	Logical Operators: Short-Circuit &&
	Logical OR (with short-circuiting)	
~	Logical NOT	not

Special Characters

Symbol	Symbol Name	Role	
@	At symbol	Function handle construction and reference	<p>The @ symbol forms anonymous functions</p> <ul style="list-style-type: none"> • “Create Function”

Symbol	Symbol Name	Role	
.	Period or dot	<ul style="list-style-type: none"> • Decimal point • Element-wise operations • Structure field access • Object property or method specifier 	<p>The period character is a MATLAB operator that enables you to access elements of an array or structure.</p> <ul style="list-style-type: none"> • “Array vs. Matrix” • “Structures” • “Access Properties”
...	Dot dot dot or ellipsis	Line continuation	<p>Three or more periods at the end of a line continues the command to the next line. The ellipsis follows the three periods.</p> <p>Note MATLAB interprets the ellipsis as a single line continuation character.</p> <ul style="list-style-type: none"> • “Continue Long Lines”

Symbol	Symbol Name	Role	
,	Comma	Separator	Use commas to separate arguments, and concatenate. <ul style="list-style-type: none">• horzcat

Symbol	Symbol Name	Role	
:	Colon	<ul style="list-style-type: none">• Vector creation• Indexing• For-loop iteration	<p>Use the colon operator for the range of a for loop.</p> <ul style="list-style-type: none">• colon• "Creating, Con"

Symbol	Symbol Name	Role	
;	Semicolon	<ul style="list-style-type: none"> • Signify end of row • Suppress output of code line 	<p>Use semicolons to end a line of code.</p> <ul style="list-style-type: none"> • vertcat
()	Parentheses	<ul style="list-style-type: none"> • Operator precedence • Function argument enclosure • Indexing 	<p>Use parentheses to enclose elements into an array.</p> <ul style="list-style-type: none"> • "Operator Precedence" • "Array Indexing"

Symbol	Symbol Name	Role	
[]	Square brackets	<ul style="list-style-type: none"> • Array construction • Array concatenation • Empty matrix and array element deletion • Multiple output argument assignment 	<p>Square brackets are used for the following:</p> <ul style="list-style-type: none"> • “Creating, Concatenating, and Deleting Array Elements” • horzcat • vertcat
{ }	Curly brackets	Cell array assignment and contents	<p>Use curly braces to create a cell array.</p> <ul style="list-style-type: none"> • “Cell Arrays”
%	Percent	<ul style="list-style-type: none"> • Comment • Conversion specifier 	<p>The percent sign is used for the following:</p> <p>Some functions also use the percent sign.</p> <p>Two percent signs are used for the following:</p> <ul style="list-style-type: none"> • “Add Comment”

Symbol	Symbol Name	Role	
%{ %}	Percent curly bracket	Block comments	<p>The %{ and %} symbols are used to delimit block comments in MATLAB. The lines that immediately follow the opening brace are the lines that immediately follow the closing brace.</p> <p>Note With the exclamation point operator, the lines that immediately follow the opening brace are the lines that immediately follow the closing brace.</p> <ul style="list-style-type: none"> • “Add Comment”
!	Exclamation point	Operating system command	<p>The exclamation point operator is used to execute operating system commands in MATLAB.</p> <ul style="list-style-type: none"> • “Shell Escape Character” <p>Not available in MATLAB 6.5 and earlier.</p>
?	Question mark	Metaclass for MATLAB class	<p>The question mark operator is used to create a metaclass for a MATLAB class. The question mark operator works only with a class.</p> <ul style="list-style-type: none"> • <code>metaclass</code>
' '	Single quotes	Character array constructor	<p>Use single quotes to create a character array in MATLAB.</p> <ul style="list-style-type: none"> • “Represent Text”
""	Double quotes	String constructor	<p>Use double quotes to create a string in MATLAB.</p> <ul style="list-style-type: none"> • “Represent Text”
N/A	Space character	Separator	<p>Use the space character to separate arguments in a function call. In this case, the space character is used to separate the arguments of the <code>sin</code> function.</p>

Symbol	Symbol Name	Role	
~	Tilde	<ul style="list-style-type: none"> • Logical NOT • Argument placeholder 	Use the tilde symbol to ignore functions. <ul style="list-style-type: none"> • not • "Ignore Functions" • "Ignore Functions"
=	Equal sign	Assignment	Use the equal sign to assign a value to a variable B. Note The = character is used to separate two arrays. See ec

String and Character Formatting

Some special characters can only be used in the text of a character vector or string. You can use these special characters to insert new lines or carriage returns, specify folder paths, and more.

Use the special characters in this table to specify a folder path using a character vector or string.

Symbol	Symbol Name	Role	
/ \	Slash and Backslash	File or folder path separation	In addition to the characters separated by spaces, both slash-based systems, and backslash-based systems, use the slash and backslash characters to separate file or folder paths.

Symbol	Symbol Name	Role	
..	Dot dot	Parent folder	Two dots in succession to specify folder <ul style="list-style-type: none"> cd
*	Asterisk	Wildcard character	In addition to being a wildcard character Wildcards are generated by MATLAB matching the character *, which
@	At symbol	Class folder indicator	An @ sign indicates <ul style="list-style-type: none"> "Class and Package"
+	Plus	Package directory indicator	A + sign indicates <ul style="list-style-type: none"> "Packages C"

There are certain special characters that you cannot enter as ordinary text. Instead, you must use unique character sequences to represent them. Use the symbols in this table to format strings and character vectors on their own or in conjunction with formatting functions like `compose`, `sprintf`, and `error`. For more information, see "Formatting Text" on page 6-36.

Symbol	Effect on Text
' '	Single quotation mark
%%	Single percent sign
\\	Single backslash
\a	Alarm
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab

Symbol	Effect on Text
<code>\v</code>	Vertical tab
<code>\xN</code>	Hexadecimal number, N
<code>\N</code>	Octal number, N

See Also

More About

- “Array vs. Matrix Operations” on page 2-13
- “Array Comparison with Relational Operators” on page 2-25
- “Compatible Array Sizes for Basic Operations” on page 2-19
- “Operator Precedence” on page 2-29
- “Find Array Elements That Meet a Condition” on page 5-2
- “Greek Letters and Special Characters in Chart Text”

Array vs. Matrix Operations

In this section...

“Introduction” on page 2-13

“Array Operations” on page 2-13

“Matrix Operations” on page 2-16

Introduction

MATLAB has two different types of arithmetic operations: array operations and matrix operations. You can use these arithmetic operations to perform numeric computations, for example, adding two numbers, raising the elements of an array to a given power, or multiplying two matrices.

Matrix operations follow the rules of linear algebra. By contrast, array operations execute element by element operations and support multidimensional arrays. The period character (.) distinguishes the array operations from the matrix operations. However, since the matrix and array operations are the same for addition and subtraction, the character pairs .+ and .- are unnecessary.

Array Operations

Array operations execute element by element operations on corresponding elements of vectors, matrices, and multidimensional arrays. If the operands have the same size, then each element in the first operand gets matched up with the element in the same location in the second operand. If the operands have compatible sizes, then each input is implicitly expanded as needed to match the size of the other. For more information, see “Compatible Array Sizes for Basic Operations” on page 2-19.

As a simple example, you can add two vectors with the same size.

```
A = [1 1 1]
```

```
A =
```

```
    1    1    1
```

```
B = [1 2 3]
```

```
B =  
    1    2    3
```

```
A+B
```

```
ans =  
    2    3    4
```

If one operand is a scalar and the other is not, then MATLAB implicitly expands the scalar to be the same size as the other operand. For example, you can compute the element-wise product of a scalar and a matrix.

```
A = [1 2 3; 1 2 3]
```

```
A =  
    1    2    3  
    1    2    3
```

```
3.*A
```

```
ans =  
    3    6    9  
    3    6    9
```

Implicit expansion also works if you subtract a 1-by-3 vector from a 3-by-3 matrix because the two sizes are compatible. When you perform the subtraction, the vector is implicitly expanded to become a 3-by-3 matrix.

```
A = [1 1 1; 2 2 2; 3 3 3]
```

```
A =  
    1    1    1  
    2    2    2  
    3    3    3
```

```
m = [2 4 6]
```

```
m =  
    2    4    6
```

```
A - m
```

```
ans =
```

```

-1   -3   -5
 0   -2   -4
 1   -1   -3
```

A row vector and a column vector have compatible sizes. If you add a 1-by-3 vector to a 2-by-1 vector, then each vector implicitly expands into a 2-by-3 matrix before MATLAB executes the element-wise addition.

```
x = [1 2 3]
```

```
x =
```

```

1     2     3
```

```
y = [10; 15]
```

```
y =
```

```

10
15
```

```
x + y
```

```
ans =
```

```

11    12    13
16    17    18
```

If the sizes of the two operands are incompatible, then you get an error.

```
A = [8 1 6; 3 5 7; 4 9 2]
```

```
A =
```

```

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

```
m = [2 4]
```

```
m =
```

```

2     4
```

A - m

Matrix dimensions must agree.

The following table provides a summary of arithmetic array operators in MATLAB. For function-specific information, click the link to the function reference page in the last column.

Operator	Purpose	Description	Reference Page
+	Addition	A+B adds A and B.	plus
+	Unary plus	+A returns A.	uplus
-	Subtraction	A-B subtracts B from A	minus
-	Unary minus	-A negates the elements of A.	uminus
.*	Element-wise multiplication	A.*B is the element-by-element product of A and B.	times
.^	Element-wise power	A.^B is the matrix with elements A(i, j) to the B(i, j) power.	power
./	Right array division	A./B is the matrix with elements A(i, j)/B(i, j).	rdivide
.\	Left array division	A.\B is the matrix with elements B(i, j)/A(i, j).	ldivide
.'	Array transpose	A.' is the array transpose of A. For complex matrices, this does not involve conjugation.	transpose

Matrix Operations

Matrix operations follow the rules of linear algebra and are not compatible with multidimensional arrays. The required size and shape of the inputs in relation to one another depends on the operation. For nonscalar inputs, the matrix operators generally calculate different answers than their array operator counterparts.

For example, if you use the matrix right division operator, /, to divide two matrices, the matrices must have the same number of columns. But if you use the matrix multiplication operator, *, to multiply two matrices, then the matrices must have a common *inner dimension*. That is, the number of columns in the first input must be equal to the number

of rows in the second input. The matrix multiplication operator calculates the product of two matrices with the formula,

$$C(i, j) = \sum_{k=1}^n A(i, k)B(k, j).$$

To see this, you can calculate the product of two matrices.

```
A = [1 3;2 4]
```

```
A =
```

```
     1     3
     2     4
```

```
B = [3 0;1 5]
```

```
B =
```

```
     3     0
     1     5
```

```
A*B
```

```
ans =
```

```
     6    15
    10    20
```

The previous matrix product is not equal to the following element-wise product.

```
A.*B
```

```
ans =
```

```
     3     0
     2    20
```

The following table provides a summary of matrix arithmetic operators in MATLAB. For function-specific information, click the link to the function reference page in the last column.

Operator	Purpose	Description	Reference Page
*	Matrix multiplication	$C = A*B$ is the linear algebraic product of the matrices A and B. The number of columns of A must equal the number of rows of B.	mtimes
\	Matrix left division	$x = A \setminus B$ is the solution to the equation $Ax = B$. Matrices A and B must have the same number of rows.	mldivide
/	Matrix right division	$x = B/A$ is the solution to the equation $xA = B$. Matrices A and B must have the same number of columns. In terms of the left division operator, $B/A = (A' \setminus B')'$.	mrdivide
^	Matrix power	A^B is A to the power B, if B is a scalar. For other values of B, the calculation involves eigenvalues and eigenvectors.	mpower
'	Complex conjugate transpose	A' is the linear algebraic transpose of A. For complex matrices, this is the complex conjugate transpose.	ctranspose

See Also

More About

- “Compatible Array Sizes for Basic Operations” on page 2-19
- “MATLAB Operators and Special Characters” on page 2-2
- “Operator Precedence” on page 2-29

Compatible Array Sizes for Basic Operations

Most binary (two-input) operators and functions in MATLAB support numeric arrays that have *compatible sizes*. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of them is 1. In the simplest cases, two array sizes are compatible if they are exactly the same or if one is a scalar. MATLAB implicitly expands arrays with compatible sizes to be the same size during the execution of the element-wise operation or function.

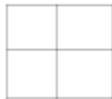
Inputs with Compatible Sizes

2-D Inputs

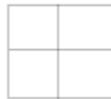
These are some combinations of scalars, vectors, and matrices that have compatible sizes:

- Two inputs which are exactly the same size.

A: 2-by-2



B: 2-by-2

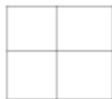


Result: 2-by-2



- One input is a scalar.

A: 2-by-2



B: 1-by-1

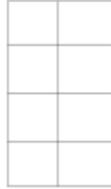


Result: 2-by-2



- One input is a matrix, and the other is a column vector with the same number of rows.

A: 4-by-2



B: 4-by-1



Result: 4-by-2



- One input is a column vector, and the other is a row vector.

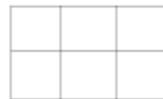
B: 2-by-1



A: 1-by-3



Result: 2-by-3

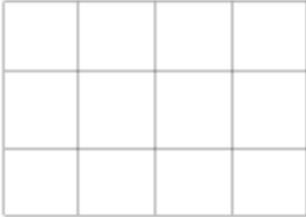


Multidimensional Arrays

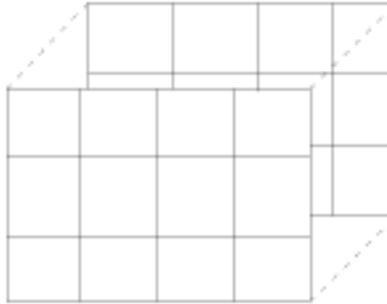
Every array in MATLAB has trailing dimensions of size 1. For multidimensional arrays, this means that a 3-by-4 matrix is the same as a matrix of size 3-by-4-by-1-by-1-by-1. Examples of multidimensional arrays with compatible sizes are:

- One input is a matrix, and the other is a 3-D array with the same number of rows and columns.

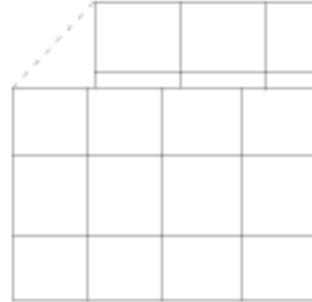
A: 3-by-4



B: 3-by-4-by-2



Result: 3-by-4-by-1

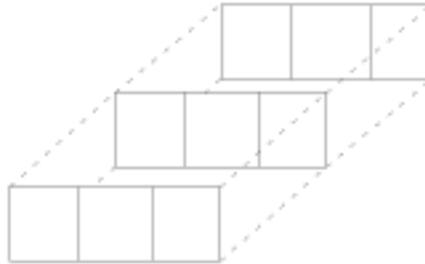


- One input is a matrix, and the other is a 3-D array. The dimensions are all either the same or one of them is 1.

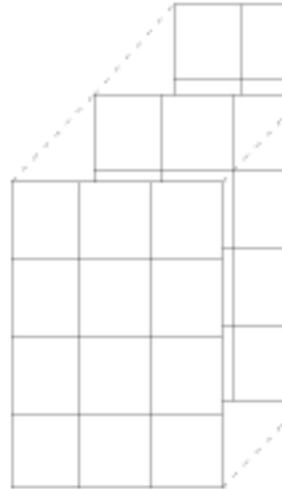
A: 4-by-3



B: 1-by-3-by-3



Result: 4-by-



Empty Arrays

The rules are the same for empty arrays or arrays that have a dimension size of zero. The size of the dimension that is not equal to 1 determines the size of the output. This means that dimensions with a size of zero must be paired with a dimension of size 1 or 0 in the other array, and that the output has a dimension size of 0.

A: 1-by-0
B: 3-by-1
Result: 3-by-0

Inputs with Incompatible Sizes

Incompatible inputs have sizes that cannot be implicitly expanded to be the same size. For example:

- One of the dimension sizes are not equal, and neither is 1.

A: 3-by-2

B: 4-by-2

- Two nonscalar row vectors with lengths that are not the same.

A: 1-by-3

B: 1-by-4

Examples

Subtract Vector from Matrix

To simplify vector-matrix operations, use implicit expansion with dimensional functions such as `sum`, `mean`, `min`, and others.

For example, calculate the mean value of each column in a matrix, then subtract the mean value from each element.

```
A = magic(3)
```

```
A =
```

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

```
C = mean(A)
```

```
C =
```

```
     5     5     5
```

```
A - C
```

```
ans =
```

```
     3    -4     1
    -2     0     2
    -1     4    -3
```

Add Row and Column Vector

Row and column vectors have compatible sizes, and when you perform an operation on them the result is a matrix.

For example, add a row and column vector. The result is the same as `bsxfun(@plus,a,b)`.

```
a = [1 2 3 4]
```

```
ans =
```

```
     1     2     3     4
```

```
b = [5; 6; 7]
```

```
ans =
```

```
     5
```

```
     6
```

```
     7
```

```
a + b
```

```
ans =
```

```
     6     7     8     9
```

```
     7     8     9    10
```

```
     8     9    10    11
```

See Also

`bsxfun`

More About

- “Array vs. Matrix Operations” on page 2-13
- “MATLAB Operators and Special Characters” on page 2-2

Array Comparison with Relational Operators

In this section...

“Array Comparison” on page 2-25

“Logic Statements” on page 2-28

Relational operators compare operands quantitatively, using operators like “less than”, “greater than”, and “not equal to.” The result of a relational comparison is a logical array indicating the locations where the relation is true.

These are the relational operators in MATLAB.

Symbol	Function Equivalent	Description
<	lt	Less than
<=	le	Less than or equal to
>	gt	Greater than
>=	ge	Greater than or equal to
==	eq	Equal to
~=	ne	Not equal to

Array Comparison

Numeric Arrays

The relational operators perform element-wise comparisons between two arrays. The arrays must have compatible sizes to facilitate the operation. Arrays with compatible sizes are implicitly expanded to be the same size during execution of the calculation. In the simplest cases, the two operands are arrays of the same size, or one is a scalar. For more information, see “Compatible Array Sizes for Basic Operations” on page 2-19.

For example, if you compare two matrices of the same size, then the result is a logical matrix of the same size with elements indicating where the relation is true.

```
A = [2 4 6; 8 10 12]
```

```
A =
```

```
      2     4     6
      8    10    12
B = [5 5 5; 9 9 9]
```

```
B =
```

```
      5     5     5
      9     9     9
```

```
A < B
```

```
ans =
```

```
      1     1     0
      1     0     0
```

Similarly, you can compare one of the arrays to a scalar.

```
A > 7
```

```
ans =
```

```
      0     0     0
      1     1     1
```

If you compare a 1-by-N row vector to an M-by-1 column vector, then MATLAB expands each vector into an M-by-N matrix before performing the comparison. The resulting matrix contains the comparison result for each combination of elements in the vectors.

```
A = 1:3
```

```
A =
```

```
      1     2     3
```

```
B = [2; 3]
```

```
B =
```

```
      2
      3
```

```
A >= B
```

```
ans =
```

```

0     1     1
0     0     1

```

Empty Arrays

The relational operators work with arrays for which any dimension has size zero, as long as both arrays have compatible sizes. This means that if one array has a dimension size of zero, then the size of the corresponding dimension in the other array must be 1 or zero, and the size of that dimension in the output is zero.

```

A = ones(3,0);
B = ones(3,1);
A == B

```

```
ans =
```

```
Empty matrix: 3-by-0
```

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 +, -, >, <, &, |, and so on.

To test for empty arrays, use `isempty(A)`.

Complex Numbers

- The operators >, <, >=, and <= use only the real part of the operands in performing comparisons.
- The operators == and ~= test both real and imaginary parts of the operands.

Inf, NaN, NaT, and undefined Element Comparisons

- Inf values are equal to other Inf values.
- NaN values are not equal to any other numeric value, including other NaN values.
- NaT values are not equal to any other datetime value, including other NaT values.
- Undefined categorical elements are not equal to any other categorical value, including other undefined elements.

Logic Statements

Use relational operators in conjunction with the logical operators $A \& B$ (AND), $A | B$ (OR), $\text{xor}(A, B)$ (XOR), and $\sim A$ (NOT), to string together more complex logical statements.

For example, you can locate where negative elements occur in two arrays.

```
A = [2 -1; -3 10]
```

```
A =
```

```
     2     -1  
    -3     10
```

```
B = [0 -2; -3 -1]
```

```
B =
```

```
     0     -2  
    -3     -1
```

```
A<0 & B<0
```

```
ans =
```

```
     0     1  
     1     0
```

For more examples, see “Find Array Elements That Meet a Condition” on page 5-2.

See Also

`eq` | `ge` | `gt` | `le` | `lt` | `ne`

More About

- “Array vs. Matrix Operations” on page 2-13
- “Compatible Array Sizes for Basic Operations” on page 2-19
- “MATLAB Operators and Special Characters” on page 2-2

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 Power with unary minus (. ^ -), unary plus (. ^ +), or logical negation (. ^ ~) as well as matrix power with unary minus (^ -), unary plus (^ +), or logical negation (^ ~).

Note Although most operators work from left to right, the operators (^ -), (. ^ -), (^ +), (. ^ +), (^ ~), and (. ^ ~) work from second from the right to left. It is recommended that you use parentheses to explicitly specify the intended precedence of statements containing these operator combinations.

- 4 Unary plus (+), unary minus (-), logical negation (~)
- 5 Multiplication (. *), right division (. /), left division (. \), matrix multiplication (*), matrix right division (/), matrix left division (\)
- 6 Addition (+), subtraction (-)
- 7 Colon operator (:)
- 8 Less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (==), not equal to (~=)
- 9 Element-wise AND (&)
- 10 Element-wise OR (|)
- 11 Short-circuit AND (&&)
- 12 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
```

See Also

More About

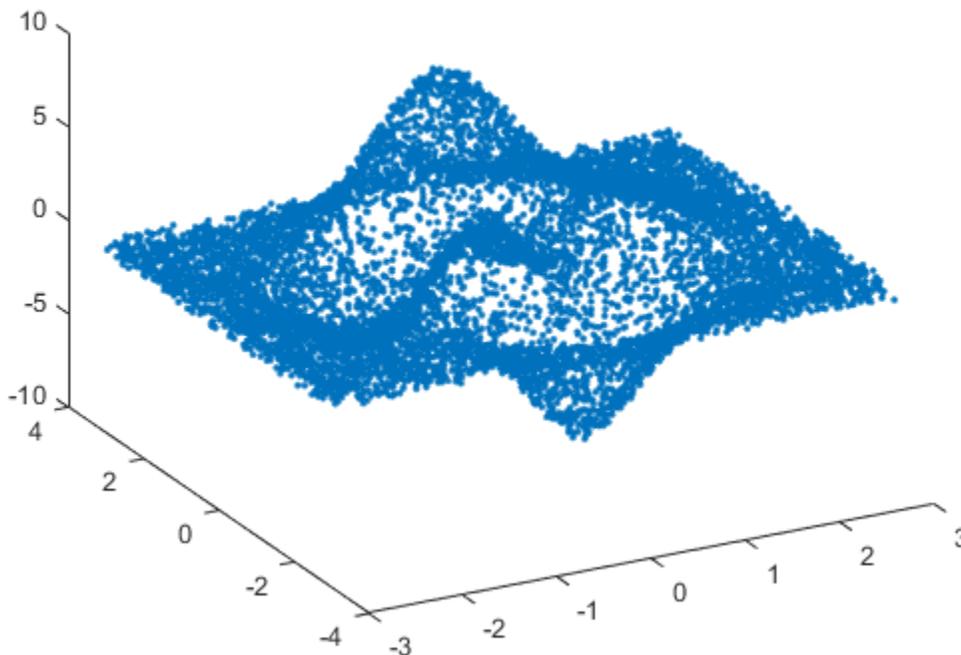
- “Array vs. Matrix Operations” on page 2-13
- “Compatible Array Sizes for Basic Operations” on page 2-19
- “Array Comparison with Relational Operators” on page 2-25
- “MATLAB Operators and Special Characters” on page 2-2

Average Similar Data Points Using a Tolerance

This example shows how to use `uniquetol` to find the average z-coordinate of 3-D points that have similar (within tolerance) x and y coordinates.

Use random points picked from the `peaks` function in the domain $[-3, 3] \times [-3, 3]$ as the data set. Add a small amount of noise to the data.

```
xy = rand(10000,2)*6-3;  
z = peaks(xy(:,1),xy(:,2)) + 0.5-rand(10000,1);  
A = [xy z];  
plot3(A(:,1), A(:,2), A(:,3), 'b.')  
view(-28,32)
```



Find points that have similar x and y coordinates using `uniquetol` with these options:

- Specify `ByRows` as `true`, since the rows of `A` contain the point coordinates.
- Specify `OutputAllIndices` as `true` to return the indices for all points that are within tolerance of each other.
- Specify `DataScale` as `[1 1 Inf]` to use an absolute tolerance for the x and y coordinates, while ignoring the z-coordinate.

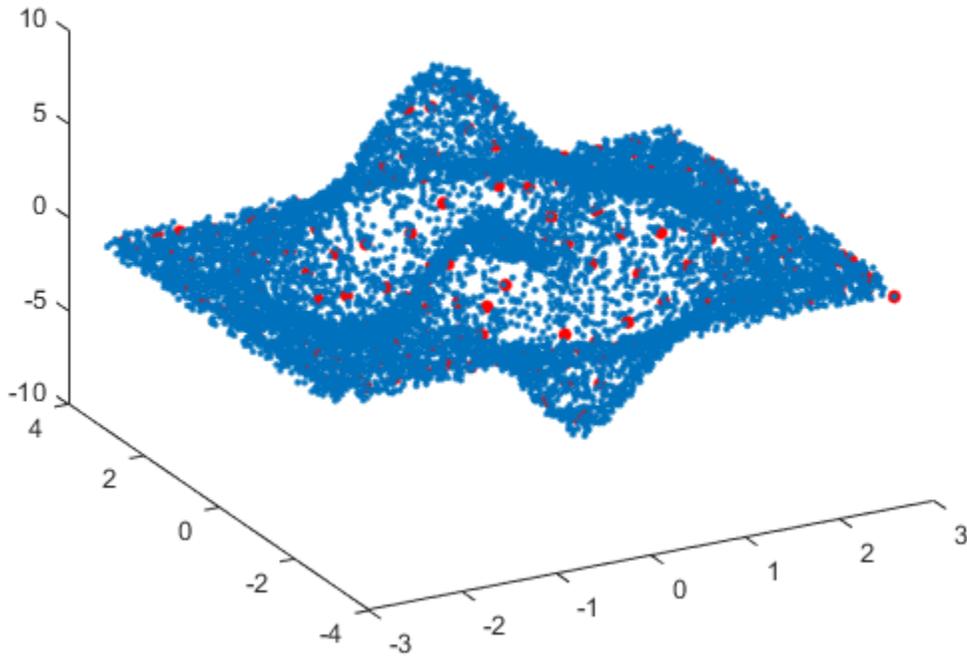
```
DS = [1 1 Inf];  
[C,ia] = uniquetol(A, 0.3, 'ByRows', true, ...  
    'OutputAllIndices', true, 'DataScale', DS);
```

Average each group of points that are within tolerance (including the z-coordinates), producing a reduced data set that still holds the general shape of the original data.

```
for k = 1:length(ia)  
    aveA(k,:) = mean(A(ia{k},:),1);  
end
```

Plot the resulting averaged-out points on top of the original data.

```
hold on  
plot3(aveA(:,1), aveA(:,2), aveA(:,3), '.r', 'MarkerSize', 15)
```



See Also

`uniquetol`

More About

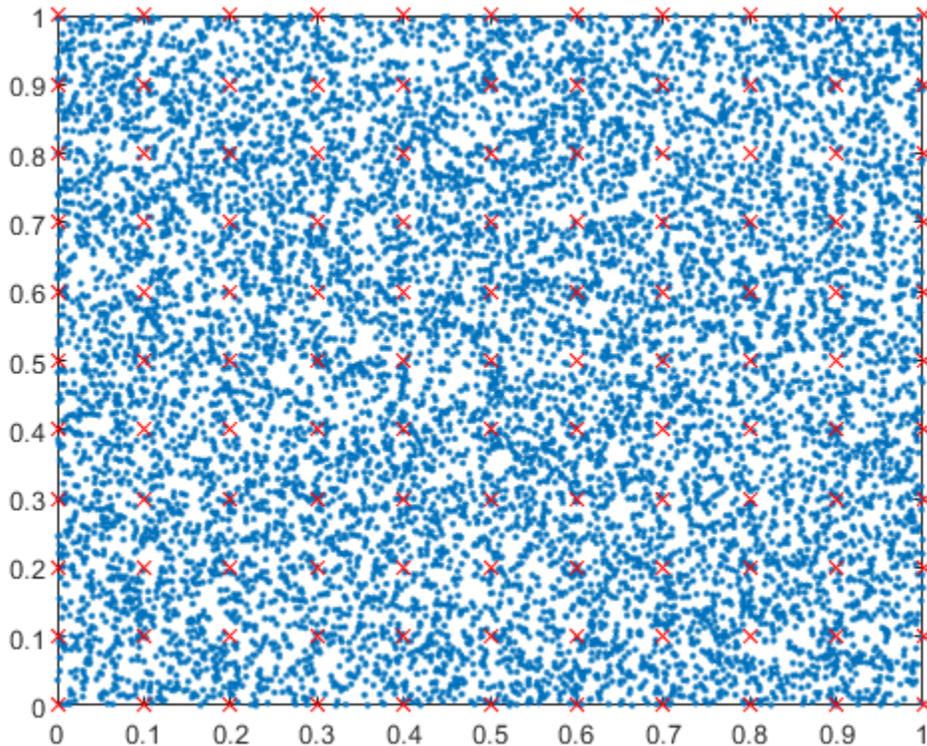
- “Group Scattered Data Using a Tolerance” on page 2-34

Group Scattered Data Using a Tolerance

This example shows how to group scattered data points based on their proximity to points of interest.

Create a set of random 2-D points. Then create and plot a grid of equally spaced points on top of the random data.

```
x = rand(10000,2);  
[a,b] = meshgrid(0:0.1:1);  
gridPoints = [a(:), b(:)];  
plot(x(:,1), x(:,2), '.')  
hold on  
plot(gridPoints(:,1), gridPoints(:,2), 'xr', 'Markersize', 6)
```



Use `ismembertol` to locate the data points in `x` that are within tolerance of the grid points in `gridPoints`. Use these options with `ismembertol`:

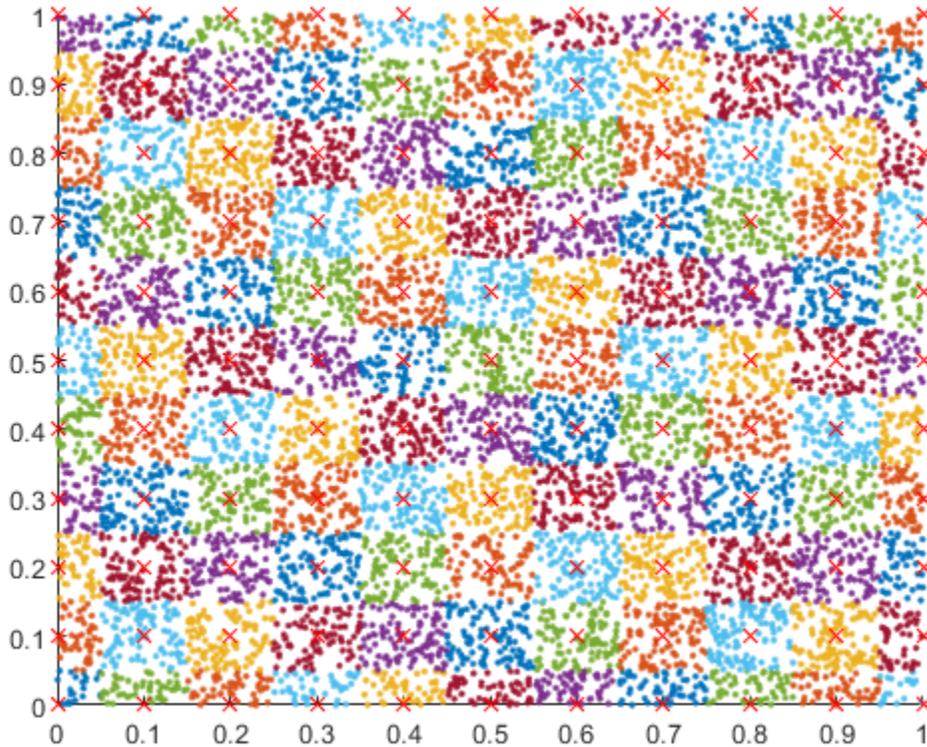
- Specify `ByRows` as `true`, since the point coordinates are in the rows of `x`.
- Specify `OutputAllIndices` as `true` to return all of the indices for rows in `x` that are within tolerance of the corresponding row in `gridPoints`.

```
[LIA,LocB] = ismembertol(gridPoints, x, 0.05, ...
    'ByRows', true, 'OutputAllIndices', true);
```

For each grid point, plot the points in `x` that are within tolerance of that grid point.

```
figure
hold on
```

```
for k = 1:length(LocB)
    plot(x(LocB{k},1), x(LocB{k},2), '.')
end
plot(gridPoints(:,1), gridPoints(:,2), 'xr', 'Markersize', 6)
```



See Also

ismembertol

More About

- “Average Similar Data Points Using a Tolerance” on page 2-31

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
```

See Also

`end` | `if` | `return` | `switch`

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 ~strcmp(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**.

See Also

break | continue | end | for | while

Regular Expressions

In this section...

“What Is a Regular Expression?” on page 2-41

“Steps for Building Expressions” on page 2-43

“Operators and Characters” on page 2-46

What Is a Regular Expression?

A regular expression is a sequence 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 character vector `'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, that is, characters that are alphabetic, numeric, or underscore (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.

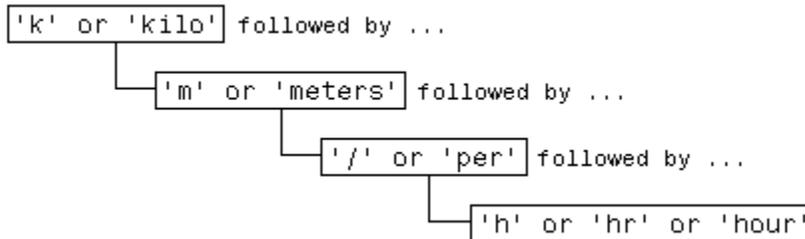
For example, several ways of expressing a metric rate of speed are:

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.
```

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



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 terms 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'
```

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 to the function reference pages.

Function	Description
<code>regexp</code>	Match regular expression.
<code>regexpi</code>	Match regular expression, ignoring case.
<code>regexprep</code>	Replace part of text using regular expression.
<code>regxprtranslate</code>	Translate text into regular expression.

When calling any of the first three functions, pass the text 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.

Steps for Building Expressions

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

- 1 Identify unique patterns in the string on page 2-44

This entails breaking up the text 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 on page 2-44

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

- 3 —Call the appropriate search function on page 2-45

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

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 the text..

```
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 529-882-1759 Fresno, CA jan_stephens@horizon.net
```

Step 1 — Identify Unique Patterns in the Text

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 shows the operators or metacharacters that represent the character pattern.

Description of each segment	Pattern
One or more lowercase letters and underscores	[a-z_]+
@ sign	@
One or more lowercase letters, no underscores	[a-z]+
Dot (period) character	\.
com or net	(com net)

Assembling these patterns into one character vector 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'
```

MATLAB parses a character vector from left to right, “consuming” the vector as it goes. If matching characters are found, `regexp` records the location and resumes parsing the character vector, starting just after the end of the most recent match.

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 argument:

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

Operators and Characters

Regular expressions can contain characters, metacharacters, operators, tokens, and flags that specify patterns to match, as described in these sections:

- “Metacharacters” on page 2-46
- “Character Representation” on page 2-47
- “Quantifiers” on page 2-48
- “Grouping Operators” on page 2-49
- “Anchors” on page 2-50
- “Lookaround Assertions” on page 2-50
- “Logical and Conditional Operators” on page 2-51
- “Token Operators” on page 2-52
- “Dynamic Expressions” on page 2-53
- “Comments” on page 2-55
- “Search Flags” on page 2-55

Metacharacters

Metacharacters represent letters, letter ranges, digits, and space characters. Use them to construct a generalized pattern of characters.

Metacharacter	Description	Example
.	Any single character, including white space	'..ain' matches sequences of five consecutive characters that end with 'ain'.
[c ₁ c ₂ c ₃]	Any character contained within the square brackets. The following characters are treated literally: \$. * + ? and - when not used to indicate a range.	'[rp.]ain' matches 'rain' or 'pain' or '.ain'.
[^c ₁ c ₂ c ₃]	Any character not contained within the square brackets. The following characters are treated literally: \$. * + ? and - when not used to indicate a range.	'[^*rp]ain' matches all four-letter sequences that end in 'ain', except 'rain' and 'pain' and '*ain'. For example, it matches 'gain', 'lain', or 'vain'.

Metacharacter	Description	Example
[c ₁ - c ₂]	Any character in the range of c ₁ through c ₂	'[A-G]' matches a single character in the range of A through G.
\w	Any alphabetic, numeric, or underscore character. For English character sets, \w is equivalent to [a-zA-Z_0-9]	'\w*' identifies a word.
\W	Any character that is not alphabetic, numeric, or underscore. For English character sets, \W is equivalent to [^a-zA-Z_0-9]	'\W*' identifies a term that is not a word.
\s	Any white-space character; equivalent to [\f\n\r\t\v]	'\w*\s' matches words that end with the letter n, followed by a white-space character.
\S	Any non-white-space character; equivalent to [^ \f\n\r\t\v]	'\d\S' matches a numeric digit followed by any non-white-space character.
\d	Any numeric digit; equivalent to [0-9]	'\d*' matches any number of consecutive digits.
\D	Any nondigit character; equivalent to [^0-9]	'\w*\D\>' matches words that do not end with a numeric digit.
\oN or \o{N}	Character of octal value N	'\o{40}' matches the space character, defined by octal 40.
\xN or \x{N}	Character of hexadecimal value N	'\x2C' matches the comma character, defined by hex 2C.

Character Representation

Operator	Description
\a	Alarm (beep)
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab

Operator	Description
<code>\v</code>	Vertical tab
<code>\char</code>	Any character with special meaning in regular expressions that you want to match literally (for example, use <code>\\</code> to match a single backslash)

Quantifiers

Quantifiers specify the number of times a pattern must occur in the matching text.

Quantifier	Matches the expression when it occurs...	Example
<code>expr*</code>	0 or more times consecutively.	' <code>\w*</code> ' matches a word of any length.
<code>expr?</code>	0 times or 1 time.	' <code>\w*(\.\m)?</code> ' matches words that optionally end with the extension <code>.m</code> .
<code>expr+</code>	1 or more times consecutively.	' <code></code> ' matches an <code></code> HTML tag when the file name contains one or more characters.
<code>expr{m,n}</code>	At least <code>m</code> times, but no more than <code>n</code> times consecutively. <code>{0,1}</code> is equivalent to <code>?</code> .	' <code>\S{4,8}</code> ' matches between four and eight non-white-space characters.
<code>expr{m,}</code>	At least <code>m</code> times consecutively. <code>{0,}</code> and <code>{1,}</code> are equivalent to <code>*</code> and <code>+</code> , respectively.	' <code></code> ' matches an <code><a></code> HTML tag when the file name contains one or more characters.
<code>expr{n}</code>	Exactly <code>n</code> times consecutively. Equivalent to <code>{n,n}</code> .	' <code>\d{4}</code> ' matches four consecutive digits.

Quantifiers can appear in three modes, described in the following table. *q* represents any of the quantifiers in the previous table.

Mode	Description	Example
<code>exprq</code>	Greedy expression: match as many characters as possible.	Given the text ' <code><tr><td><p>text</p></td></code> ', the expression ' <code></?t.*></code> ' matches all characters between <code><tr</code> and <code>/td></code> : ' <code><tr><td><p>text</p></td></code> '
<code>exprq?</code>	Lazy expression: match as few characters as necessary.	Given the text ' <code><tr><td><p>text</p></td></code> ', the expression ' <code></?t.*?></code> ' ends each match at the first occurrence of the closing angle bracket (<code>></code>): ' <code><tr></code> ' ' <code><td></code> ' ' <code></td></code> '
<code>exprq+</code>	Possessive expression: match as much as possible, but do not rescan any portions of the text.	Given the text ' <code><tr><td><p>text</p></td></code> ', the expression ' <code></?t.*+></code> ' does not return any matches, because the closing angle bracket is captured using <code>.*</code> , and is not rescanned.

Grouping Operators

Grouping operators allow you to capture tokens, apply one operator to multiple elements, or disable backtracking in a specific group.

Grouping Operator	Description	Example
<code>(expr)</code>	Group elements of the expression and capture tokens.	' <code>Joh?n\s(\w*)</code> ' captures a token that contains the last name of any person with the first name John or Jon.
<code>(?:expr)</code>	Group, but do not capture tokens.	' <code>(?:[aeiou][^aeiou]){2}</code> ' matches two consecutive patterns of a vowel followed by a nonvowel, such as 'anon'. Without grouping, ' <code>[aeiou][^aeiou]{2}</code> ' matches a vowel followed by two nonvowels.

Grouping Operator	Description	Example
(?>expr)	Group atomically. Do not backtrack within the group to complete the match, and do not capture tokens.	'A(?>.*)Z' does not match 'AtoZ', although 'A(?:.*)Z' does. Using the atomic group, Z is captured using .* and is not rescanned.
(expr1 expr2)	Match expression expr1 or expression expr2. If there is a match with expr1, then expr2 is ignored. You can include ?: or ?> after the opening parenthesis to suppress tokens or group atomically.	'(let tel)\w+' matches words that start with let or tel.

Anchors

Anchors in the expression match the beginning or end of a character vector or word.

Anchor	Matches the...	Example
^expr	Beginning of the input text.	'^M\w*' matches a word starting with M at the beginning of the text.
expr\$	End of the input text.	'\w*m\$' matches words ending with m at the end of the text.
\<expr	Beginning of a word.	'\<n\w*' matches any words starting with n.
expr\>	End of a word.	'\w*e\>' matches any words ending with e.

Lookaround Assertions

Lookaround assertions look for patterns that immediately precede or follow the intended match, but are not part of the match.

The pointer remains at the current location, and characters that correspond to the test expression are not captured or discarded. Therefore, lookahead assertions can match overlapping character groups.

Lookaround Assertion	Description	Example
<code>expr(?:=test)</code>	Look ahead for characters that match <code>test</code> .	<code>'\w*(?=ing)'</code> matches terms that are followed by <code>ing</code> , such as <code>'Fly'</code> and <code>'fall'</code> in the input text <code>'Flying, not falling.'</code>
<code>expr(?:!test)</code>	Look ahead for characters that do not match <code>test</code> .	<code>'i(?:!ng)'</code> matches instances of the letter <code>i</code> that are not followed by <code>ng</code> .
<code>(?<=test)expr</code>	Look behind for characters that match <code>test</code> .	<code>'(?:<=re)\w*'</code> matches terms that follow <code>'re'</code> , such as <code>'new'</code> , <code>'use'</code> , and <code>'cycle'</code> in the input text <code>'renew, reuse, recycle'</code>
<code>(?<!test)expr</code>	Look behind for characters that do not match <code>test</code> .	<code>'(?:<!\d)(\d)(?:!\d)'</code> matches single-digit numbers (digits that do not precede or follow other digits).

If you specify a lookahead assertion *before* an expression, the operation is equivalent to a logical AND.

Operation	Description	Example
<code>(?:=test)expr</code>	Match both <code>test</code> and <code>expr</code> .	<code>'(?:=[a-z])[^aeiou]'</code> matches consonants.
<code>(?:!test)expr</code>	Match <code>expr</code> and do not match <code>test</code> .	<code>'(?:![aeiou])[a-z]'</code> matches consonants.

For more information, see “Lookahead Assertions in Regular Expressions” on page 2-57.

Logical and Conditional Operators

Logical and conditional operators allow you to test the state of a given condition, and then use the outcome to determine which pattern, if any, to match next. These operators support logical OR and `if` or `if/else` conditions. (For AND conditions, see “Lookaround Assertions” on page 2-50.)

Conditions can be tokens on page 2-52, lookahead assertions on page 2-50, or dynamic expressions on page 2-53 of the form `(?:@cmd)`. Dynamic expressions must return a logical or numeric value.

Conditional Operator	Description	Example
<code>expr1 expr2</code>	Match expression <code>expr1</code> or expression <code>expr2</code> . If there is a match with <code>expr1</code> , then <code>expr2</code> is ignored.	' <code>(let tel)\w+</code> ' matches words that start with <code>let</code> or <code>tel</code> .
<code>(?(cond)expr)</code>	If condition <code>cond</code> is true, then match <code>expr</code> .	' <code>(?(?@ispc)[A-Z]:\\)</code> ' matches a drive name, such as <code>C:\</code> , when run on a Windows system.
<code>(?(cond)expr1 expr2)</code>	If condition <code>cond</code> is true, then match <code>expr1</code> . Otherwise, match <code>expr2</code> .	' <code>Mr(s?)\..*?(?(1)her his)\w*</code> ' matches text that includes <code>her</code> when the text begins with <code>Mrs</code> , or that includes <code>his</code> when the text begins with <code>Mr</code> .

Token Operators

Tokens are portions of the matched text that you define by enclosing part of the regular expression in parentheses. You can refer to a token by its sequence in the text (an ordinal token), or assign names to tokens for easier code maintenance and readable output.

Ordinal Token Operator	Description	Example
<code>(expr)</code>	Capture in a token the characters that match the enclosed expression.	' <code>Joh?n\s(\w*)</code> ' captures a token that contains the last name of any person with the first name <code>John</code> or <code>Jon</code> .
<code>\N</code>	Match the <code>N</code> th token.	' <code><(\w+).*>.*</\1></code> ' captures tokens for HTML tags, such as <code>'title'</code> from the text <code>'<title>Some text</title>'</code> .
<code>(?(N)expr1 expr2)</code>	If the <code>N</code> th token is found, then match <code>expr1</code> . Otherwise, match <code>expr2</code> .	' <code>Mr(s?)\..*?(?(1)her his)\w*</code> ' matches text that includes <code>her</code> when the text begins with <code>Mrs</code> , or that includes <code>his</code> when the text begins with <code>Mr</code> .

Named Token Operator	Description	Example
(?<name>expr)	Capture in a named token the characters that match the enclosed expression.	' (?<month>\d+) - (?<day>\d+) - (?<yr>\d+) ' creates named tokens for the month, day, and year in an input date of the form mm-dd-yy.
\k<name>	Match the token referred to by name.	' <(?<tag>\w+).*>.*</\k<tag>>' captures tokens for HTML tags, such as 'title' from the text '<title>Some text</title>'.
(?(name)expr1 expr2)	If the named token is found, then match expr1. Otherwise, match expr2.	' Mr(?<sex>s?)\..*?(?(sex)her his) \w*' matches text that includes her when the text begins with Mrs, or that includes his when the text begins with Mr.

Note If an expression has nested parentheses, MATLAB captures tokens that correspond to the outermost set of parentheses. For example, given the search pattern '(and(y|rew))', MATLAB creates a token for 'andrew' but not for 'y' or 'rew'.

For more information, see “Tokens in Regular Expressions” on page 2-60.

Dynamic Expressions

Dynamic expressions allow you to execute a MATLAB command or a regular expression to determine the text to match.

The parentheses that enclose dynamic expressions do *not* create a capturing group.

Operator	Description	Example
(??expr)	Parse expr and include the resulting term in the match expression. When parsed, expr must correspond to a complete, valid regular expression. Dynamic expressions that use the backslash escape character (\) require two backslashes: one for the initial parsing of expr, and one for the complete match.	'^(\d+)((??\w{\$1}))' determines how many characters to match by reading a digit at the beginning of the match. The dynamic expression is enclosed in a second set of parentheses so that the resulting match is captured in a token. For instance, matching '5XXXXX' captures tokens for '5' and 'XXXXX'.
(??@cmd)	Execute the MATLAB command represented by cmd, and include the output returned by the command in the match expression.	'(.{2,}).?(??@fliplr(\$1))' finds palindromes that are at least four characters long, such as 'abba'.
(?@cmd)	Execute the MATLAB command represented by cmd, but discard any output the command returns. (Helpful for diagnosing regular expressions.)	'\w*?(\w)(?@disp(\$1))\1\w*' matches words that include double letters (such as pp), and displays intermediate results.

Within dynamic expressions, use the following operators to define replacement terms.

Replacement Operator	Description
\$& or \$0	Portion of the input text that is currently a match
\$`	Portion of the input text that precedes the current match
\$'	Portion of the input text that follows the current match (use \$'' to represent \$')
\$N	Nth token
\$<name>	Named token
\${cmd}	Output returned when MATLAB executes the command, cmd

For more information, see “Dynamic Regular Expressions” on page 2-66.

Comments

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 text.

Characters	Description	Example
<code>(?#comment)</code>	Insert a comment in the regular expression. The comment text is ignored when matching the input.	<code>'(?# Initial digit)\<\d\w+'</code> includes a comment, and matches words that begin with a number.

Search Flags

Search flags modify the behavior for matching expressions.

Flag	Description
<code>(?-i)</code>	Match letter case (default for <code>regexp</code> and <code>regexprep</code>).
<code>(?i)</code>	Do not match letter case (default for <code>regexpi</code>).
<code>(?s)</code>	Match dot (<code>.</code>) in the pattern with any character (default).
<code>(?-s)</code>	Match dot in the pattern with any character that is not a newline character.
<code>(?-m)</code>	Match the <code>^</code> and <code>\$</code> metacharacters at the beginning and end of text (default).
<code>(?m)</code>	Match the <code>^</code> and <code>\$</code> metacharacters at the beginning and end of a line.
<code>(?-x)</code>	Include space characters and comments when matching (default).
<code>(?x)</code>	Ignore space characters and comments when matching. Use <code>'\ '</code> and <code>'\#'</code> to match space and <code>#</code> characters.

The expression that the flag modifies can appear either after the parentheses, such as

```
(?i)\w*
```

or inside the parentheses and separated from the flag with a colon (`:`), such as

```
(?i:\w*)
```

The latter syntax allows you to change the behavior for part of a larger expression.

See Also

`regex` | `regexpi` | `regexprep` | `regexpttranslate`

More About

- “Lookahead Assertions in Regular Expressions” on page 2-57
- “Tokens in Regular Expressions” on page 2-60
- “Dynamic Regular Expressions” on page 2-66

Lookahead Assertions in Regular Expressions

In this section...

“Lookahead Assertions” on page 2-57

“Overlapping Matches” on page 2-58

“Logical AND Conditions” on page 2-58

Lookahead Assertions

There are two types of lookahead assertions for regular expressions: lookahead and lookbehind. In both cases, the assertion is a condition that must be satisfied to return a match to the expression.

A *lookahead* assertion has the form `(?=test)` and can appear anywhere in a regular expression. MATLAB looks ahead of the current location in the text for the test condition. If MATLAB matches the test condition, it continues processing the rest of the expression to find a match.

For example, look ahead in a character vector specifying a path to find the name of the folder that contains a program file (in this case, `fileread.m`).

```
chr = which('fileread')  
  
chr =  
    matlabroot\toolbox\matlab\iofun\fileread.m  
  
regexp(chr, '\\w+(?=\\w+\\. [mp])', 'match')  
  
ans =  
    'iofun'
```

The match expression, `\\w+`, searches for one or more alphanumeric or underscore characters. Each time `regexp` finds a term that matches this condition, it looks ahead for a backslash (specified with two backslashes, `\\`), followed by a file name (`\\w+`) with an `.m` or `.p` extension (`\\. [mp]`). The `regexp` function returns the match that satisfies the lookahead condition, which is the folder name `iofun`.

Overlapping Matches

Lookahead assertions do not consume any characters in the text. As a result, you can use them to find overlapping character sequences.

For example, use lookahead to find *every* sequence of six nonwhitespace characters in a character vector by matching initial characters that precede five additional characters:

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

The starting indices correspond to these phrases:

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

Without the lookahead operator, MATLAB parses a character vector from left to right, consuming the vector as it goes. If matching characters are found, `regexp` records the location and resumes parsing the character vector from the location of the most recent match. There is no overlapping of characters in this process.

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

The starting indices correspond to these phrases:

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

Logical AND Conditions

Another way to use a lookahead operation is to perform a logical AND between two conditions. This example initially attempts to locate all lowercase consonants in a character array consisting of the first 50 characters of the help for the `normest` function:

```
helptext = help('normest');  
chr = helptext(1:50)
```

```
chr =
  NORMEST Estimate the matrix 2-norm.
  NORMEST(S
```

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

```
c = regexp(chr, '[^aeiou]', 'match')

c =
  Columns 1 through 14

   ' '      'N'      'O'      'R'      'M'      'E'      'S'      'T'      ' '
   'E'      's'      't'      'm'      't'
   ...
```

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(chr, '(?=[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`

See Also

`regexp` | `regexp_i` | `regexprep`

More About

- “Regular Expressions” on page 2-41

Tokens in Regular Expressions

In this section...

“Introduction” on page 2-60
“Multiple Tokens” on page 2-62
“Unmatched Tokens” on page 2-62
“Tokens in Replacement Text” on page 2-63
“Named Capture” on page 2-64

Introduction

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 text. 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.

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 character array, 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 character array. The second part of the expression, `'\1'`, looks for a second instance of the same character immediately following the first:

```
poe = ['While I nodded, nearly napping, ' ...  
      'suddenly there came a tapping,'];  
  
[mat,tok,ext] = regexp(poe, '(\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 `po` are:

```
11 11, 26 26, 35 35, 57 57
```

For another example, 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 angle bracket (`<`) followed by one or more alphabetic, numeric, or underscore characters. The enclosing parentheses capture token characters following the opening angle 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 angle 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'
```

Multiple Tokens

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`.

Unmatched Tokens

For those tokens specified in the regular expression that have no match in the text being evaluated, `regexp` and `regexpi` return an empty character vector (`''`) 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 a character vector specifying the path returned from the MATLAB `tempdir` function. The regular expression `expr` includes six token specifiers, one for each piece of the path. The third specifier `[a-z]+` has no match in the character vector because this part of the path, `Profiles`, begins with an uppercase letter:

```
chr = tempdir
```

```
chr =
  C:\WINNT\Profiles\bascal\LOCALS~1\Temp\
expr = ['([A-Z]:)\(WINNT)\([a-z]+\)?.*\' ...
       '([a-z+)\([A-Z]+~\d)\(Temp)\'];
[tok, ext] = regexp(chr, expr, 'tokens', 'tokenExtents');
```

When a token is not found in the text, `regexp` returns an empty character vector (' ') as the token and a numeric array with the token extent. 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 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 term, `Profiles`, begins in the path. 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
```

Tokens in Replacement Text

When using tokens in replacement text, 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 `regexp` returns the modified text, 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.

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

```
poe = ['While I nodded, nearly napping, ' ...  
      'suddenly there came a tapping,'];  
  
regexp(poe, '(?<anychar>.)\k<anychar>', 'match')  
  
ans =  
    'dd'    'pp'    'dd'    'pp'
```

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

For example, parse different parts of street addresses from several character vectors. A short name is assigned to each token in the expression:

```
chr1 = '134 Main Street, Boulder, CO, 14923';  
chr2 = '26 Walnut Road, Topeka, KA, 25384';  
chr3 = '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(chr1, expr, 'names')  
  
loc1 =  
    adrs: '134 Main Street'  
    city: 'Boulder'
```

```
    state: 'CO'
    zip: '14923'

loc2 = regexp(chr2, expr, 'names')

loc2 =
  adrs: '26 Walnut Road'
  city: 'Topeka'
  state: 'KA'
  zip: '25384'

loc3 = regexp(chr3, expr, 'names')

loc3 =
  adrs: '847 Industrial Drive'
  city: 'Elizabeth'
  state: 'NJ'
  zip: '73548'
```

See Also

[regexp](#) | [regexp_i](#) | [regprep](#)

More About

- “Regular Expressions” on page 2-41

Dynamic Regular Expressions

In this section...

“Introduction” on page 2-66

“Dynamic Match Expressions — (??expr)” on page 2-67

“Commands That Modify the Match Expression — (??@cmd)” on page 2-68

“Commands That Serve a Functional Purpose — (?@cmd)” on page 2-69

“Commands in Replacement Expressions — \${cmd}” on page 2-71
--

Introduction

In a dynamic expression, you can make the pattern that you want `regexp` to match dependent on the content of the input text. In this way, you can more closely match varying input patterns in the text being parsed. You can also use dynamic expressions in replacement terms 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(text, match_expr)
regexpi(text, match_expr)
regexprep(text, match_expr, replace_expr)
```

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($2))}` enables you to base the replacement expression on the input text so that you do not have to change the expression each time. This example uses the dynamic replacement syntax `${cmd}`.

```
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
```

When parsed, a dynamic expression must correspond to a complete, valid regular expression. In addition, dynamic match expressions that use the backslash escape character (`\`) require two backslashes: one for the initial parsing of the expression, and one for the complete match. The parentheses that enclose dynamic expressions do *not* create a capturing group.

There are three forms of dynamic expressions that you can use in match expressions, and one form for replacement expressions, as described in the following sections

Dynamic Match Expressions – (??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:

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

The purpose of this particular command is to locate a series of X characters in each of the character vectors stored in the input cell array. Note however that the number of Xs varies in each character vector. 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 character vector 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 vectors in the cell array. With the first input character vector, `regexp` looks for five X characters; with the second, it looks for eight, and with the third, it looks for just one:

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

Commands That Modify the Match Expression — (??@cmd)

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

For example, use the dynamic expression `(??@fliplr($1))` to locate a palindrome, “Never Odd or Even”, that has been embedded into a larger character vector.

First, create the input string. Make sure that all letters are lowercase, and remove all nonword characters.

```
chr = lower(...  
    'Find the palindrome Never Odd or Even in this string');  
  
chr = regexprep(str, '\W*', '')  
  
chr =  
    findthepalindromeneveroddoreveninthisstring
```

Locate the palindrome within the character vector using the dynamic expression:

```
palchr = regexp(chr, '(.{3,}).?(??@fliplr($1))', 'match')  
  
palchr =  
    'neveroddoreven'
```

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

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

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. This functionality can be useful in diagnosing 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 character vector 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*', 'match')
i
p
p
```

```
ans =  
    'mississippi'
```

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 text quite differently. In this case, MATLAB uses the very first match it can find, and does not even consider the rest of the text:

```
regexp('mississippi', '\w*(\w)(?@disp($1))\1\w*', 'match')
```

```
m  
i  
s
```

```
ans =  
    'mississippi'
```

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 text. 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 text, 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 character vector. The dynamic script (?@if(~isempty(\$&))) serves to omit the empty character vectors 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 text 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 text looking for the letter `g`. At each iteration through the text, `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 text by marking the current location being parsed with a `^` character.

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

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

regexp(chr, expr, 'once');

starting match: [^abcdefghij]
starting match: [a^bcdefghij]
starting match: [ab^cdefghij]
starting match: [abc^defghij]
starting match: [abcd^efghij]
starting match: [abcde^fghij]
starting match: [abcdef^ghij]
```

Commands in Replacement Expressions — `${cmd}`

The `${cmd}` operator modifies the contents of a regular expression replacement pattern, making this pattern adaptable to parameters in the input text 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 `regexp` call shown here, the replacement pattern is `'${convertMe($1,$2)}'`. In this case, the entire replacement pattern is a dynamic expression:

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

The dynamic expression tells MATLAB to execute a function named `convertMe` using the two tokens `(\d+\.?*\d*)` and `(\w+)`, derived from the text being matched, as input arguments in the call to `convertMe`. The replacement pattern requires a dynamic expression because the values of `$1` and `$2` are generated at runtime.

The following example defines the file named `convertMe` that converts measurements from imperial units to metric.

```
function valout = convertMe(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];
end
```

At the command line, call the `convertMe` function from `regexprep`, passing in values for the quantity to be converted and name of the imperial unit:

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

ans =
    This highway is 201.1625 kilometers long

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

```
ans =
    This pitcher holds 1.1828 litres of water

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

See Also

regexp | regexpi | regexprep

More About

- “Regular Expressions” on page 2-41

Comma-Separated Lists

In this section...
“What Is a Comma-Separated List?” on page 2-74
“Generating a Comma-Separated List” on page 2-74
“Assigning Output from a Comma-Separated List” on page 2-76
“Assigning to a Comma-Separated List” on page 2-77
“How to Use the Comma-Separated Lists” on page 2-79
“Fast Fourier Transform Example” on page 2-81

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 uses `deal` to overwrite each element in a comma-separated list. First create a list.

```

c{1} = [31 07];
c{2} = [03 78];
c{:}

ans =

    31     7

```

```
ans =
```

```
3    78
```

Use `deal` to overwrite each element in the list.

```
[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
```

Use `deal` to overwrite the structure fields.

```
[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 2-79
- “Displaying Arrays” on page 2-80
- “Concatenation” on page 2-80
- “Function Call Arguments” on page 2-80
- “Function Return Values” on page 2-81

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 =
```

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

```
ans =
```

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

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
```

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 = cell(2,3);
C{1,1} = 'LineWidth';
C{2,1} = 2;
C{1,2} = 'MarkerEdgeColor';
C{2,2} = 'k';
C{1,3} = 'MarkerFaceColor';
C{2,3} = 'g';
figure
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{:});
end
```

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
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.

Alternatives to the eval Function

In this section...

“Why Avoid the eval Function?” on page 2-83

“Variables with Sequential Names” on page 2-83

“Files with Sequential Names” on page 2-84

“Function Names in Variables” on page 2-85

“Field Names in Variables” on page 2-85

“Error Handling” on page 2-86

Why Avoid the eval Function?

Although the `eval` function is very powerful and flexible, it is 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.
- Concatenated character vectors within an `eval` statement are 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 Cell Array” on page 12-3
- “Create Structure Array” on page 11-2

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 1-9
- “Import or Export 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 character vector. 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 function from a list stored in a cell array:

```
examples = {@odedemo,@sunspots,@fitdemo};
n = input('Select an example (1, 2, or 3): ');
examples{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 11-13.

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.

Classes (Data Types)

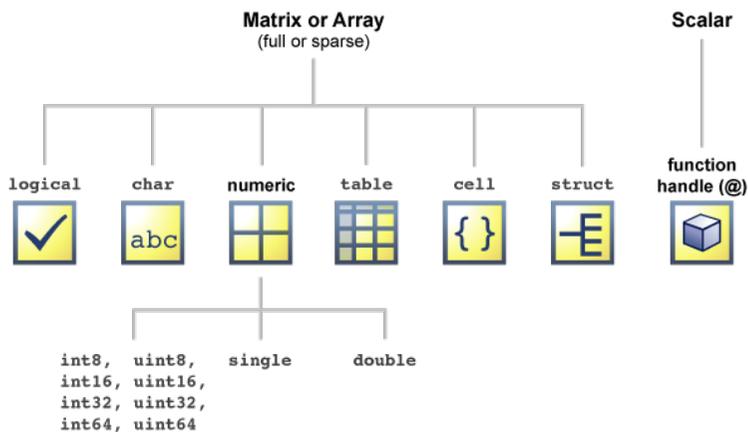
Overview of MATLAB Classes

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. Tables, structures, and cell arrays provide a way to store dissimilar types of data in the same container.

There are 16 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 shown 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.

The following table describes the fundamental classes in more detail.

Class Name	Documentation	Intended Use
<code>double</code> , <code>single</code>	Floating-Point Numbers on page 4-7	<ul style="list-style-type: none"> Required for fractional numeric data. Double on page 4-7 and Single on page 4-7 precision. Use <code>realmin</code> and <code>realmax</code> to show range of values on page 4-11. Two-dimensional arrays can be sparse. Default numeric type in MATLAB.
<code>int8</code> , <code>uint8</code> , <code>int16</code> , <code>uint16</code> , <code>int32</code> , <code>uint32</code> , <code>int64</code> , <code>uint64</code>	Integers on page 4-2	<ul style="list-style-type: none"> Use for signed and unsigned whole numbers. More efficient use of memory. on page 29-4 Use <code>intmin</code> and <code>intmax</code> to show range of values on page 4-5. Choose from 4 sizes (8, 16, 32, and 64 bits).
<code>char</code> , <code>string</code>	"Characters and Strings"	<ul style="list-style-type: none"> Data type for text. Native or Unicode®. Converts to/from numeric. Use with regular expressions on page 2-41. For multiple character arrays, use cell arrays. Starting in R2016b, you also can store text in string arrays. For more information, see <code>string</code>.

Class Name	Documentation	Intended Use
<code>logical</code>	"Logical Operations"	<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.
<code>function_handle</code>	"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.• Use to specify graphics callback functions.• Save to MAT-file and restore later.
<code>table</code>	"Tables"	<ul style="list-style-type: none">• Rectangular container for mixed-type, column-oriented data.• Row and variable names identify contents.• Use the properties of a table to store metadata such as variable units.• Manipulation of elements similar to numeric or logical arrays.• Access data by numeric or named index.• Can select a subset of data and preserve the table container or can extract the data from a table.
<code>struct</code>	"Structures"	<ul style="list-style-type: none">• Fields store arrays of varying classes and sizes.• Access one or all fields/indices in single operation.• Field names identify contents.• Method of passing function arguments.• Use in comma-separated lists on page 2-74.• More memory required for overhead

Class Name	Documentation	Intended Use
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 numeric or logical arrays.• Method of passing function arguments.• Use in comma-separated lists.• More memory required for overhead

See Also

More About

- "Valid Combinations of Unlike Classes" on page 15-2

Numeric Classes

- “Integers” on page 4-2
- “Floating-Point Numbers” on page 4-7
- “Complex Numbers” on page 4-17
- “Infinity and NaN” on page 4-18
- “Identifying Numeric Classes” on page 4-21
- “Display Format for Numeric Values” on page 4-22

Integers

In this section...

“Integer Classes” on page 4-2

“Creating Integer Data” on page 4-3

“Arithmetic Operations on Integer Classes” on page 4-4

“Largest and Smallest Values for Integer Classes” on page 4-5

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
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;  
int16(x)  
ans =
```

```
int16
```

```
325
```

```
x = x + .001;  
int16(x)  
ans =
```

```
int16
```

```
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 =
```

```
int16
```

```
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 =
```

```
int16
1427
```

The integer conversion functions are also useful when converting other classes, such as strings, to integers:

```
str = 'Hello World';
```

```
int8(str)
ans =
```

```
1×11 int8 row vector
72  101  108  108  111  32  87  111  114  108  100
```

If you convert a NaN value into an integer class, the result is a value of 0 in that integer class. For example,

```
int32(NaN)
ans =
```

```
int32
0
```

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 element-wise 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.

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 4-2 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')
ans =
```

```
  int8
```

```
  127
```

```
intmin('int8')
ans =
```

```
  int8
```

```
 -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
```

```
127
```

```
x = int8(-300)
x =
```

```
int8
```

```
-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
```

```
127
```

```
x = int8(-100) * 3
x =
```

```
int8
```

```
-128
```

Floating-Point Numbers

In this section...

“Double-Precision Floating Point” on page 4-7

“Single-Precision Floating Point” on page 4-7

“Creating Floating-Point Data” on page 4-8

“Arithmetic Operations on Floating-Point Numbers” on page 4-9

“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

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 f of the number $1.f$

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)

Bits	Usage
30 to 23	Exponent, biased by 127
22 to 0	Fraction <i>f</i> 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 =
    logical
     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 =
    single
    -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 =

    single

    Inf

-realmax('single') - .0001e+038
ans =

    single

    -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 + eps(5)`. If a double-precision computation returns the answer 5, the result is only accurate to within `eps(5)`.

The value of `eps(x)` depends on `x`. This example shows that, as `x` gets larger, so does `eps(x)`:

```
eps(50)
ans =
    7.105427357601002e-15
```

If you enter `eps` with no input argument, MATLAB returns the value of `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`, `eps(x)` returns the distance between `x` and the next larger single-precision number. For example,

```
x = single(5);
eps(x)

returns

ans =

    single

    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 =
```

```
single  
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 =  
  
    logical  
  
    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 =

    logical

    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.

References

- [1] Moler, Cleve. "Floating Points." *MATLAB News and Notes*. Fall, 1996.
- [2] Moler, Cleve. *Numerical Computing with MATLAB*. Natick, MA: The MathWorks, Inc., 2004.

Complex Numbers

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
```

Infinity and NaN

In this section...
“Infinity” on page 4-18
“NaN” on page 4-18

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.

<code>x = 1/0</code> <code>x =</code> <code>Inf</code>	<code>x = 1.e1000</code> <code>x =</code> <code>Inf</code>
<code>x = exp(1000)</code> <code>x =</code> <code>Inf</code>	<code>x = log(0)</code> <code>x =</code> <code>-Inf</code>

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

You can also create NaNs by:

```
x = NaN;
```

```
whos x
  Name      Size      Bytes  Class
  x         1x1         8      double
```

The NaN function returns one of the IEEE arithmetic representations for NaN as a double scalar value. The exact bit-wise hexadecimal representation of this NaN value is,

```
format hex
x = NaN
x =
    fff8000000000000
```

Always use the `isnan` function to verify that the elements in an array are NaN:

```
isnan(x)
ans =
    1
```

MATLAB preserves the “Not a Number” status of alternate NaN representations and treats all of the different representations of NaN equivalently. However, in some special cases (perhaps due to hardware limitations), MATLAB does not preserve the exact bit pattern of alternate NaN representations throughout an entire calculation, and instead uses the canonical NaN bit pattern defined above.

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
```

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 4-22
“Display Format Examples” on page 4-22
“Setting Numeric Format in a Program” on page 4-23

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);
```


The Logical Class

- “Find Array Elements That Meet a Condition” on page 5-2
- “Determine if Arrays Are Logical” on page 5-7
- “Reduce Logical Arrays to Single Value” on page 5-11
- “Truth Table for Logical Operations” on page 5-14

Find Array Elements That Meet a Condition

This example shows how to filter the elements of an array by applying conditions to the array. For instance, you can examine the even elements in a matrix, find the location of all 0s in a multidimensional array, or replace NaN values in data. You can perform these tasks using a combination of the relational and logical operators. The relational operators ($>$, $<$, $>=$, $<=$, $==$, $\sim=$) impose conditions on the array, and you can apply multiple conditions by connecting them with the logical operators *and*, *or*, and *not*, respectively denoted by the symbols $\&$, $|$, and \sim .

Apply a Single Condition

To apply a single condition, start by creating a 5-by-5 matrix that contains random integers between 1 and 15. Reset the random number generator to the default state for reproducibility.

```
rng default
A = randi(15,5)
```

```
A = 5x5
```

```
    13     2     3     3    10
    14     5    15     7     1
     2     9    15    14    13
    14    15     8    12    15
    10    15    13    15    11
```

Use the relational *less than* operator, $<$, to determine which elements of *A* are less than 9. Store the result in *B*.

```
B = A < 9
```

```
B = 5x5 logical array
```

```
    0     1     1     1     0
    0     1     0     1     1
    1     0     0     0     0
    0     0     1     0     0
    0     0     0     0     0
```

The result is a logical matrix. Each value in *B* represents a logical 1 (*true*) or logical 0 (*false*) state to indicate whether the corresponding element of *A* fulfills the condition *A*

`< 9`. For example, `A(1,1)` is 13, so `B(1,1)` is logical 0 (false). However, `A(1,2)` is 2, so `B(1,2)` is logical 1 (true).

Although `B` contains information about *which* elements in `A` are less than 9, it doesn't tell you what their *values* are. Rather than comparing the two matrices element by element, you can use `B` to index into `A`.

`A(B)`

`ans = 8×1`

```

2
2
5
3
8
3
7
1

```

The result is a column vector of the elements in `A` that are less than 9. Since `B` is a logical matrix, this operation is called **logical indexing**. In this case, the logical array being used as an index is the same size as the other array, but this is not a requirement. For more information, see “Array Indexing”.

Some problems require information about the *locations* of the array elements that meet a condition rather than their actual values. In this example, you can use the `find` function to locate all of the elements in `A` less than 9.

`I = find(A < 9)`

`I = 8×1`

```

3
6
7
11
14
16
17
22

```

The result is a column vector of linear indices. Each index describes the location of an element in A that is less than 9, so in practice $A(I)$ returns the same result as $A(B)$. The difference is that $A(B)$ uses logical indexing, whereas $A(I)$ uses linear indexing.

Apply Multiple Conditions

You can use the logical **and**, **or**, and **not** operators to apply any number of conditions to an array; the number of conditions is not limited to one or two.

First, use the logical **and** operator, denoted **&**, to specify two conditions: the elements must be **less than 9** and **greater than 2**. Specify the conditions as a logical index to view the elements that satisfy both conditions.

```
A(A<9 & A>2)
```

```
ans = 5×1
```

```
5
3
8
3
7
```

The result is a list of the elements in A that satisfy both conditions. Be sure to specify each condition with a separate statement connected by a logical operator. For example, you cannot specify the conditions above by $A(2<A<9)$, since it evaluates to $A(2<A \mid A<9)$.

Next, find the elements in A that are **less than 9** and **even numbered**.

```
A(A<9 & ~mod(A,2))
```

```
ans = 3×1
```

```
2
2
8
```

The result is a list of all even elements in A that are less than 9. The use of the logical NOT operator, **~**, converts the matrix $\text{mod}(A, 2)$ into a logical matrix, with a value of logical 1 (**true**) located where an element is evenly divisible by 2.

Finally, find the elements in A that are **less than 9** and **even numbered** and **not equal to 2**.

```
A(A<9 & ~mod(A,2) & A~=2)
```

```
ans = 8
```

The result, 8, is even, less than 9, and not equal to 2. It is the only element in A that satisfies all three conditions.

Use the `find` function to get the index of the element equal to 8 that satisfies the conditions.

```
find(A<9 & ~mod(A,2) & A~=2)
```

```
ans = 14
```

The result indicates that $A(14) = 8$.

Replace Values That Meet a Condition

Sometimes it is useful to simultaneously change the values of several existing array elements. Use logical indexing with a simple assignment statement to replace the values in an array that meet a condition.

Replace all values in A that are greater than 10 with the number 10.

```
A(A>10) = 10
```

```
A = 5×5
```

10	2	3	3	10
10	5	10	7	1
2	9	10	10	10
10	10	8	10	10
10	10	10	10	10

Next, replace all values in A that are not equal to 10 with a NaN value.

```
A(A~=10) = NaN
```

```
A = 5×5
```

10	NaN	NaN	NaN	10
----	-----	-----	-----	----

```
10 NaN 10 NaN NaN
NaN NaN 10 10 10
10 10 NaN 10 10
10 10 10 10 10
```

Lastly, replace all of the NaN values in A with zeros and apply the logical NOT operator, ~A.

```
A(isnan(A)) = 0;
C = ~A
```

C = 5x5 logical array

```
0 1 1 1 0
0 1 0 1 1
1 1 0 0 0
0 0 1 0 0
0 0 0 0 0
```

The resulting matrix has values of logical 1 (true) in place of the NaN values, and logical 0 (false) in place of the 10s. The logical NOT operation, ~A, converts the numeric array into a logical array such that A&C returns a matrix of logical 0 (false) values and A|C returns a matrix of logical 1 (true) values.

See Also

Logical Operators: Short Circuit | and | find | isnan | nan | not | or | xor

Determine if Arrays Are Logical

To determine whether an array is logical, you can test the entire array or each element individually. This is useful when you want to confirm the output data type of a function.

This page shows several ways to determine if an array is logical.

In this section...

“Identify Logical Matrix” on page 5-7

“Test an Entire Array” on page 5-8

“Test Each Array Element” on page 5-8

“Summary Table” on page 5-9

Identify Logical Matrix

Create a 3-by-6 matrix and locate all elements greater than 0.5.

```
A = gallery('uniformdata',[3,6],0) > 0.5
```

```
A =
```

```

     1     0     0     0     1     0
     0     1     0     1     1     1
     1     1     1     1     0     1

```

The result, A, is a 3-by-6 logical matrix.

Use the `whos` function to confirm the size, byte count, and class (or data type) of the matrix, A.

```
whos A
```

```

Name      Size      Bytes  Class      Attributes
   A       3x6           18  logical

```

The result confirms that A is a 3-by-6 logical matrix.

Test an Entire Array

Use the `islogical` function to test whether A is logical.

```
islogical(A)
```

```
ans =
```

```
1
```

The result is logical 1 (true).

Use the `class` function to display a string with the class name of A.

```
class(A)
```

```
ans =
```

```
logical
```

The result confirms that A is logical.

Test Each Array Element

Create a cell array, C, and use the 'islogical' option of the `cellfun` function to identify which cells contain logical values.

```
C = {1, 0, true, false, pi, A};  
cellfun('islogical',C)
```

```
ans =
```

```
0     0     1     1     0     1
```

The result is a logical array of the same size as C.

To test each element in a numeric matrix, use the `arrayfun` function.

```
arrayfun(@islogical,A)
```

```
ans =
```

```
1     1     1     1     1     1  
1     1     1     1     1     1  
1     1     1     1     1     1
```

The result is a matrix of logical values of the same size as `A`. `arrayfun(@islogical,A)` always returns a matrix of all logical 1 (true) or logical 0 (false) values.

Summary Table

Use these MATLAB functions to determine if an array is logical.

Function Syntax	Output Size	Description
<code>whos(A)</code>	N/A	Displays the name, size, storage bytes, class, and attributes of variable <code>A</code> .
<code>islogical(A)</code>	scalar	Returns logical 1 (true) if <code>A</code> is a logical array; otherwise, it returns logical 0 (false). The result is the same as using <code>isa(A,'logical')</code> .
<code>isa(A,'logical')</code>	scalar	Returns logical 1 (true) if <code>A</code> is a logical array; otherwise, it returns logical 0 (false). The result is the same as using <code>islogical(A)</code> .
<code>class(A)</code>	single string	Returns a string with the name of the class of variable <code>A</code> .
<code>cellfun('islogical',A)</code>	Array of the same size as <code>A</code>	For cell arrays only. Returns logical 1 (true) for each cell that contains a logical array; otherwise, it returns logical 0 (false).
<code>arrayfun(@islogical,A)</code>	Array of the same size as <code>A</code>	Returns an array of logical 1 (true) values if <code>A</code> is logical; otherwise, it returns an array of logical 0 (false) values.

See Also

`arrayfun` | `cellfun` | `class` | `isa` | `islogical` | `whos`

Reduce Logical Arrays to Single Value

This example shows how to use the `any` and `all` functions to reduce an entire array to a single logical value.

The `any` and `all` functions are natural extensions of the logical `|` (OR) and `&` (AND) operators, respectively. However, rather than comparing just two elements, the `any` and `all` functions compare all of the elements in a particular dimension of an array. It is as if all of those elements are connected by `&` or `|` operators and the `any` or `all` functions evaluate the resulting long logical expressions. Therefore, unlike the core logical operators, the `any` and `all` functions reduce the size of the array dimension that they operate on so that it has size 1. This enables the reduction of many logical values into a single logical condition.

First, create a matrix `A` that contains random integers between 1 and 25. Reset the random number generator to the default state for reproducibility.

```
rng default
A = randi(25,5)
```

```
A = 5x5
```

```
    21     3     4     4    17
    23     7    25    11     1
     4    14    24    23    22
    23    24    13    20    24
    16    25    21    24    17
```

Next, use the `mod` function along with the logical NOT operator, `~`, to determine which elements in `A` are even.

```
A = ~mod(A,2)
```

```
A = 5x5 logical array
```

```
    0     0     1     1     0
    0     0     0     0     0
    1     1     1     0     1
    0     1     0     1     1
    1     0     0     1     0
```

The resulting matrices have values of logical 1 (`true`) where an element is even, and logical 0 (`false`) where an element is odd.

Since the `any` and `all` functions reduce the dimension that they operate on to size 1, it normally takes two applications of one of the functions to reduce a 2-D matrix into a single logical condition, such as `any(any(A))`. However, if you use the notation `A(:)` to regard all of the elements of `A` as a single column vector, you can use `any(A(:))` to get the same logical information without nesting the function calls.

Determine if any elements in `A` are even.

```
any(A(:))
```

```
ans = logical  
     1
```

You can perform logical and relational comparisons within the function call to `any` or `all`. This makes it easy to quickly test an array for a variety of properties.

Determine if all elements in `A` are odd.

```
all(~A(:))
```

```
ans = logical  
     0
```

Determine whether any main or super diagonal elements in `A` are even. Since the vectors returned by `diag(A)` and `diag(A,1)` are not the same size, you first need to reduce each diagonal to a single scalar logical condition before comparing them. You can use the short-circuit OR operator `||` to perform the comparison, since if any elements in the first diagonal are even then the entire expression evaluates to true regardless of what appears on the right-hand side of the operator.

```
any(diag(A) || any(diag(A,1)))
```

```
ans = logical  
     1
```

See Also

Logical Operators: Short Circuit | all | and | any | or | xor

Truth Table for Logical Operations

The following reference table shows the results of applying the binary logical operators to a series of logical 1 (true) and logical 0 (false) scalar pairs. To calculate NAND, NOR or XNOR logical operations, simply apply the logical NOT operator to the result of a logical AND, OR, or XOR operation, respectively.

Inputs A and B		and A & B	or A B	xor xor(A,B)	not ~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

See Also

Logical Operators: Short Circuit | and | not | or | xor

Characters and Strings

- “Represent Text with Character and String Arrays” on page 6-2
- “Create Character Arrays” on page 6-6
- “Create String Arrays” on page 6-12
- “Cell Arrays of Character Vectors” on page 6-21
- “Analyze Text Data with String Arrays” on page 6-24
- “Test for Empty Strings and Missing Values” on page 6-31
- “Formatting Text” on page 6-36
- “Compare Text” on page 6-47
- “Search and Replace Text” on page 6-54
- “Convert from Numeric Values to Character Array” on page 6-61
- “Convert from Character Arrays to Numeric Values” on page 6-63
- “Frequently Asked Questions About String Arrays” on page 6-66
- “Update Your Code to Accept Strings” on page 6-72
- “Function Summary” on page 6-82

Represent Text with Character and String Arrays

There are two ways to represent text in MATLAB®. You can store text in character arrays. A typical use is to store short pieces of text as *character vectors*. And starting in R2016b, you can also store multiple pieces of text in string arrays. String arrays provide a set of functions for working with text as data.

Represent Text with Character Vectors

Create a character vector by enclosing a sequence of characters in single quotation marks. MATLAB® displays character vectors using single quotation marks.

```
chr = 'Hello, world'
```

```
chr =  
'Hello, world'
```

Character vectors store characters as 1-by-n vectors. You can index directly into character vectors to get characters, or to change them.

```
chr(1:5)
```

```
ans =  
'Hello'
```

```
chr(1:5) = 'HELLO';  
chr
```

```
chr =  
'HELLO, world'
```

You can work with character vectors just as you would with arrays of any other type. For example, you can concatenate character vectors.

```
street = '123 Maple St.';  
city = 'Lakeview, MA 01234';  
fullAddress = [street ' ', city]
```

```
fullAddress =  
'123 Maple St., Lakeview, MA 01234'
```

Typical uses for character vectors include specifying file names, plot labels, or input arguments for functions. For more information on character arrays, see “Create Character Arrays” on page 6-6.

Represent Text with String Arrays

You also can store text in string arrays. Each element of a string array stores a 1-by-n character vector.

Starting in R2017a, you can create strings using double quotes. MATLAB® displays strings with double quotes.

```
str = "Welcome, friend"

str =
"Welcome, friend"
```

As an alternative, you can convert a character vector to a string using the `string` function.

```
chr = 'Hello, world';
str = string(chr)

str =
"Hello, world"
```

`str` is a 1-by-1 string, or string scalar. To find the number of characters in a string, use the `strlength` function.

```
whos str
```

Name	Size	Bytes	Class	Attributes
str	1x1	174	string	

```
strlength(str)
```

```
ans = 12
```

You can store multiple pieces of text in a string array. Each element of the array can contain a string of a different size.

```
str = ["Mercury", "Gemini", "Apollo"; ...
      "Skylab", "Skylab B", "ISS"]

str = 2x3 string array
    "Mercury"    "Gemini"    "Apollo"
    "Skylab"    "Skylab B" "ISS"
```

`str` is a 2-by-3 string array. You can find the lengths of the strings with the `strlength` function.

```
whos str
```

Name	Size	Bytes	Class	Attributes
str	2x3	484	string	

```
L = strlength(str)
```

```
L = 2x3
```

7	6	6
6	8	3

As an alternative, you also can convert a cell array of character vectors to a string array using the `string` function.

```
C = {'Mercury', 'Venus', 'Earth'};  
str = string(C)
```

```
str = 1x3 string array  
    "Mercury"    "Venus"    "Earth"
```

Use string arrays to store and work with multiple pieces of text. You can find and replace substrings, sort and reshape string arrays, and work with text as data. For more information on string arrays, see “Create String Arrays” on page 6-12.

See Also

`cellstr` | `char` | `string` | `strlength`

Related Examples

- “Create Character Arrays” on page 6-6
- “Create String Arrays” on page 6-12
- “Analyze Text Data with String Arrays” on page 6-24
- “Cell Arrays of Character Vectors” on page 6-21

- “Frequently Asked Questions About String Arrays” on page 6-66

Create Character Arrays

Create Character Vector

Create a *character vector* by enclosing a sequence of characters in single quotation marks.

```
chr = 'Hello, world'

chr =

    'Hello, world'
```

Character vectors are 1-by-n arrays of type `char`. In computer programming, *string* is a frequently-used term for a 1-by-n array of characters. However, starting in R2016b MATLAB also provides a `string` data type, so 1-by-n character arrays are referred to as character vectors in MATLAB documentation.

```
whos chr
```

Name	Size	Bytes	Class	Attributes
chr	1x12	24	char	

If the text contains a single quotation mark, include two quotation marks when assigning the character vector.

```
newChr = 'You''re right'

newChr =

    'You're right'
```

Functions such as `uint16` convert characters to their numeric codes.

```
chrNumeric = uint16(chr)

chrNumeric =

    1x12 uint16 row vector

    72    101    108    108    111    44    32    119    111    114    108    100
```

The `char` function converts the integer vector back to characters.

```
chrAlpha = char([72 101 108 108 111 44 32 119 111 114 108 100])
chrAlpha =
    'Hello, world'
```

Create Rectangular Character Array

Character arrays are m -by- n arrays of characters, where m is not always 1. You can join two or more character vectors together to create a character array. This is called concatenation and is explained for numeric arrays in “Creating, Concatenating, and Expanding Matrices”. As with numeric arrays, you can combine character arrays vertically or horizontally to create a new character array.

However, it is recommended that you store character vectors in a cell array on page 6-21, instead of using m -by- n character arrays. Cell arrays are flexible containers that allow you to easily store character vectors of varying length.

Combine Character Vectors Vertically

To combine character vectors into a two-dimensional character array, use square brackets or the `char` function.

- Apply the MATLAB concatenation operator, `[]`. Separate each row with a semicolon `(;)`. Each row must contain the same number of characters. For example, combine three character vectors of equal length:

```
devTitle = ['Thomas R. Lee'; ...
            'Sr. Developer'; ...
            'SFTware Corp. ']
```

```
devTitle =
    3x13 char array

    'Thomas R. Lee'
    'Sr. Developer'
    'SFTware Corp.'
```

If the character vectors have different lengths, pad with space characters as needed. For example:

```
mgrTitle = ['Harold A. Jorgensen      '; ...  
           'Assistant Project Manager'; ...  
           'SFTware Corp.'           '']
```

```
mgrTitle =
```

```
3×25 char array
```

```
'Harold A. Jorgensen      '  
'Assistant Project Manager'  
'SFTware Corp.'         '
```

- Call the `char` function. If the character vectors have different lengths, `char` pads the shorter vectors with trailing blanks so that each row has the same number of characters.

```
mgrTitle = char('Harold A. Jorgensen', ...  
              'Assistant Project Manager', 'SFTware Corp.')
```

```
mgrTitle =
```

```
3×25 char array
```

```
'Harold A. Jorgensen      '  
'Assistant Project Manager'  
'SFTware Corp.'         '
```

Combining Character Vectors Horizontally

To combine character vectors into a single row vector, use square brackets or the `strcat` function.

- Apply the MATLAB concatenation operator, `[]`. Separate the input character vectors with a comma or a space. This method preserves any trailing spaces in the input arrays.

```
name = 'Thomas R. Lee';  
title = 'Sr. Developer';  
company = 'SFTware Corp.';
```

```
fullName = [name ' ', ' title ' ', ' company]
```

MATLAB returns

```
fullName =
    'Thomas R. Lee, Sr. Developer, SFTware Corp.'
```

- Call the concatenation function, `strcat`. This method removes trailing spaces in the inputs. For example, combine character vectors to create a hypothetical email address.

```
name = 'myname ';
domain = 'mydomain ';
ext = 'com ';

address = strcat(name, '@', domain, '.', ext)
```

MATLAB returns

```
address =
    'myname@mydomain.com'
```

Identify Characters

Use any of the following functions to identify a character array, or certain characters in a character array.

Function	Description
<code>ischar</code>	Determine whether the input is a character array
<code>isletter</code>	Find all alphabetic letters in the input character array
<code>isspace</code>	Find all space characters in the input character array
<code>isstrprop</code>	Find all characters of a specific category

Find the spaces in a character vector.

```
chr = 'Find the space characters in this character vector';
%      |   |   |           |   |   |           |
%      5   9   15         26 29   34           44

find(isspace(chr))

ans =
     5     9    15    26    29    34    44
```

Work with Space Characters

The `blanks` function creates a character vector of space characters. Create a vector of 15 space characters. Character vectors always are displayed between single quotation marks.

```
chr = blanks(15)
```

```
chr =  
'                '
```

Insert a few nonspace characters in the middle of the blank character vector.

```
chr(6:10) = 'AAAAA'
```

```
chr =  
'      AAAAA      '
```

You can justify the positioning of these characters to the left or right using the `strjust` function:

```
chrLeft = strjust(chr, 'left')
```

```
chrLeft =  
'AAAAA           '
```

```
chrRight = strjust(chr, 'right')
```

```
chrRight =  
'           AAAAA'
```

Remove all trailing space characters with `deblank`:

```
chrDeblank = deblank(chr)
```

```
chrDeblank =  
'      AAAAA'
```

Remove all leading and trailing spaces with `strtrim`:

```
chrTrim = strtrim(chr)
```

```
chrTrim =  
    'AAAAA'
```

Expand 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.

See Also

[char](#) | [deblank](#) | [strcat](#) | [string](#) | [strjust](#) | [strtrim](#)

Related Examples

- “Represent Text with Character and String Arrays” on page 6-2
- “Search and Replace Text” on page 6-54
- “Compare Text” on page 6-47
- “Formatting Text” on page 6-36

Create String Arrays

String arrays were introduced in R2016b. String arrays store pieces of text and provide a set of functions for working with text as data. You can index into, reshape, and concatenate strings arrays just as you can with arrays of any other type. You also can access the characters in a string and append text to strings using the plus operator. To rearrange strings within a string array, use functions such as `split`, `join`, and `sort`.

Create String Arrays from Variables

MATLAB® provides string arrays to store pieces of text. Each element of a string array contains a 1-by-N character vector.

Starting in R2017a, you can create a string using double quotes.

```
str = "Hello, world"

str =
"Hello, world"
```

As an alternative, you can convert a character vector to a string using the `string` function. `chr` is a 1-by-17 character vector. `str` is a 1-by-1 string that has the same text as the character vector.

```
chr = 'Greetings, friend'

chr =
'Greetings, friend'

str = string(chr)

str =
"Greetings, friend"
```

Create a string array using the `[]` operator. `str` is a 2-by-3 string array that contains six strings.

```
str = ["Mercury","Gemini","Apollo";
      "Skylab","Skylab B","ISS"]

str = 2x3 string array
    "Mercury"    "Gemini"    "Apollo"
    "Skylab"    "Skylab B"  "ISS"
```

Find the length of each string in `str` with the `strlength` function. Use `strlength`, not `length`, to determine the number of characters in strings.

```
L = strlength(str)
```

```
L = 2×3
```

```
    7    6    6
    6    8    3
```

As an alternative, you can convert a cell array of character vectors to a string array using the `string` function. MATLAB® displays strings in string arrays with double quotes, and displays characters vectors in cell arrays with single quotes.

```
C = {'Mercury', 'Venus', 'Earth'}
```

```
C = 1x3 cell array
    {'Mercury'}    {'Venus'}    {'Earth'}
```

```
str = string(C)
```

```
str = 1x3 string array
    "Mercury"    "Venus"    "Earth"
```

In addition to character vectors, you can convert numeric, datetime, duration, and categorical values to strings using the `string` function.

Convert a numeric array to a string array.

```
X = [5 10 20 3.1416];
string(X)
```

```
ans = 1x4 string array
    "5"    "10"    "20"    "3.1416"
```

Convert a datetime value to a string.

```
d = datetime('now');
string(d)
```

```
ans =
"27-Aug-2018 14:46:36"
```

Also, you can read text from files into string arrays using the `readtable`, `textscan`, and `fscanf` functions.

Create Empty and Missing Strings

String arrays can contain both empty and missing values. An empty string contains zero characters. When you display an empty string, the result is a pair of double quotes with nothing between them (""). The missing string is the string equivalent to `NaN` for numeric arrays. It indicates where a string array has missing values. When you display a missing string, the result is `<missing>`, with no quotation marks.

Create an empty string array using the `strings` function. When you call `strings` with no arguments, it returns an empty string. Note that the size of `str` is 1-by-1, not 0-by-0. However, `str` contains zero characters.

```
str = strings
```

```
str =  
""
```

Create an empty character vector using single quotes. Note that the size of `chr` is 0-by-0.

```
chr = ''
```

```
chr =
```

```
0x0 empty char array
```

Create a string array where every element is an empty string. You can preallocate a string array with the `strings` function.

```
str = strings(2,3)
```

```
str = 2x3 string array  
    ""    ""    ""  
    ""    ""    ""
```

To create a missing string, convert a missing value using the `string` function. The missing string displays as `<missing>`.

```
str = string(missing)
```

```
str =  
<missing>
```

You can create a string array with both empty and missing strings. Use the `ismissing` function to determine which elements are strings with missing values. Note that the empty string is not a missing string.

```
str(1) = "";
str(2) = "Gemini";
str(3) = string(missing)

str = 1x3 string array
      ""      "Gemini"      <missing>

ismissing(str)

ans = 1x3 logical array

      0      0      1
```

Compare a missing string to another string. The result is always 0 (false), even when you compare a missing string to another missing string.

```
str = string(missing);
str == "Gemini"

ans = logical
      0

str == string(missing)

ans = logical
      0
```

Access Elements of String Array

String arrays support array operations such as indexing and reshaping. Use array indexing to access the first row of `str` and all of the columns.

```
str = ["Mercury", "Gemini", "Apollo";
      "Skylab", "Skylab B", "ISS"];
str(1,:)
ans = "Mercury" "Gemini" "Apollo"
```

```
ans = 1x3 string array
    "Mercury"    "Gemini"    "Apollo"
```

Access the second element in the second row of `str`.

```
str(2,2)

ans =
"Skylab B"
```

Assign a new string outside the bounds of `str`. MATLAB® expands the array and fills unallocated elements with missing values.

```
str(3,4) = "Mir"

str = 3x4 string array
    "Mercury"    "Gemini"    "Apollo"    <missing>
    "Skylab"    "Skylab B"    "ISS"        <missing>
    <missing>    <missing>    <missing>    "Mir"
```

Access Characters Within Strings

You can index into a string array using curly braces, `{}`, to access characters directly. Use curly braces when you need to access and modify characters within a string element. Indexing with curly braces provides compatibility for code that could work with either string arrays or cell arrays of character vectors. But whenever possible, use string functions to work with the characters in strings.

Access the second element in the second row with curly braces. `chr` is a character vector, not a string.

```
str = ["Mercury", "Gemini", "Apollo";
       "Skylab", "Skylab B", "ISS"];
chr = str{2,2}

chr =
'Skylab B'
```

Access the character vector and return the first three characters.

```
str{2,2}(1:3)

ans =
'Sky'
```

Find the space characters in a string and replace them with dashes. Use the `isspace` function to inspect individual characters within the string. `isspace` returns a logical vector that contains a true value wherever there is a space character. Finally, display the modified string element, `str(2,2)`.

```
tf = isspace(str{2,2})
tf = 1x8 logical array
    0    0    0    0    0    0    1    0
```

```
str{2,2}(tf) = "-";
str(2,2)
```

```
ans =
"Skylab-B"
```

Note that in this case, you can also replace spaces using the `replace` function, without resorting to curly brace indexing.

```
replace(str(2,2), " ", "-")
ans =
"Skylab-B"
```

Concatenate Strings into String Array

Concatenate strings into a string array just as you would concatenate arrays of any other kind.

Concatenate two string arrays using square brackets, `[]`.

```
str1 = ["Mercury", "Gemini", "Apollo"];
str2 = ["Skylab", "Skylab B", "ISS"];
str = [str1 str2]

str = 1x6 string array
    "Mercury"    "Gemini"    "Apollo"    "Skylab"    "Skylab B"    "ISS"
```

Transpose `str1` and `str2`. Concatenate them and then vertically concatenate column headings onto the string array. When you concatenate character vectors into a string array, the character vectors are automatically converted to strings.

```
str1 = str1';  
str2 = str2';  
str = [str1 str2];  
str = [{"Mission:","Station:"} ; str]  
  
str = 4x2 string array  
"Mission:"    "Station:"  
"Mercury"     "Skylab"  
"Gemini"      "Skylab B"  
"Apollo"      "ISS"
```

Append Text to Strings

To append text to strings, use the plus operator, +. The plus operator appends text to strings but does not change the size of a string array.

Append a last name to an array of names. If you append a character vector to strings, then the character vector is automatically converted to a string.

```
names = ["Mary";"John";"Elizabeth";"Paul";"Ann"];  
names = names + ' Smith'  
  
names = 5x1 string array  
"Mary Smith"  
"John Smith"  
"Elizabeth Smith"  
"Paul Smith"  
"Ann Smith"
```

Append different last names. You can append text to a string array from a string array or from a cell array of character vectors. When you add nonscalar arrays, they must be the same size.

```
names = ["Mary";"John";"Elizabeth";"Paul";"Ann"];  
lastnames = ["Jones";"Adams";"Young";"Burns";"Spencer"];  
names = names + " " + lastnames  
  
names = 5x1 string array  
"Mary Jones"  
"John Adams"  
"Elizabeth Young"  
"Paul Burns"  
"Ann Spencer"
```

Append a missing string. When you append a missing string with the plus operator, the output is a missing string.

```
str1 = "Jones";
str2 = string(missing);
str1 + str2
```

```
ans =
<missing>
```

Split, Join, and Sort String Array

MATLAB® provides a rich set of functions to work with string arrays. For example, you can use the `split`, `join`, and `sort` functions to rearrange the string array names so that the names are in alphabetical order by last name.

Split names on the space characters. Splitting changes names from a 5-by-1 string array to a 5-by-2 array.

```
names = ["Mary Jones"; "John Adams"; "Elizabeth Young"; "Paul Burns"; "Ann Spencer"];
names = split(names)
```

```
names = 5x2 string array
    "Mary"    "Jones"
    "John"    "Adams"
    "Elizabeth" "Young"
    "Paul"    "Burns"
    "Ann"     "Spencer"
```

Switch the columns of names so that the last names are in the first column. Add a comma after each last name.

```
names = [names(:,2) names(:,1)];
names(:,1) = names(:,1) + ','
```

```
names = 5x2 string array
    "Jones,"    "Mary"
    "Adams,"    "John"
    "Young,"    "Elizabeth"
    "Burns,"    "Paul"
    "Spencer,"  "Ann"
```

Join the last and first names. The `join` function places a space character between the strings it joins. After the join, `names` is a 5-by-1 string array.

```
names = join(names)
```

```
names = 5x1 string array
    "Jones, Mary"
    "Adams, John"
    "Young, Elizabeth"
    "Burns, Paul"
    "Spencer, Ann"
```

Sort the elements of `names` so that they are in alphabetical order.

```
names = sort(names)
```

```
names = 5x1 string array
    "Adams, John"
    "Burns, Paul"
    "Jones, Mary"
    "Spencer, Ann"
    "Young, Elizabeth"
```

See Also

`ismissing` | `isspace` | `join` | `plus` | `sort` | `split` | `string` | `strings` | `strlength`

Related Examples

- “Analyze Text Data with String Arrays” on page 6-24
- “Search and Replace Text” on page 6-54
- “Compare Text” on page 6-47
- “Test for Empty Strings and Missing Values” on page 6-31
- “Frequently Asked Questions About String Arrays” on page 6-66

Cell Arrays of Character Vectors

Convert to Cell Array of Character Vectors

When you create character arrays from character vectors, all of the vectors must have the same length. This often means that you have to pad blanks at the end of character vectors 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. A *cell array of character vectors* is a cell array where every cell contains a character vector. Cell arrays of character vectors provide a flexible way to store character vectors of varying lengths.

While the phrase "cell array of strings" frequently has been used to describe such arrays, the phrase is not accurate because such a cell array holds character vectors, not strings. Starting in R2016b, MATLAB provides *string arrays* as another means of storing text. If you create variables that have the `string` data type, store them in string arrays, not cell arrays. For more information, see "Represent Text with Character and String Arrays" on page 6-2.

Convert a character array to a cell array of character vectors. `data` is padded with spaces so that each row has an equal number of characters. Use `cellstr` to convert the character array.

```
data = ['Allison Jones'; 'Development  '; 'Phoenix      '];
celldata = cellstr(data)
```

```
celldata =
```

```
3×1 cell array

    {'Allison Jones'}
    {'Development' }
    {'Phoenix'      }
```

`data` is a 3-by-13 character array, while `celldata` is a 3-by-1 cell array of character vectors. `cellstr` also strips the blank spaces at the ends of the rows of `data`.

The `iscellstr` function determines if the input argument is a cell array of character vectors. It returns a logical 1 (true) in the case of `celldata`:

```
iscellstr(celldata)
```

```
ans =
```

```
logical
```

```
1
```

Use `char` to convert back to a padded character array.

```
chr = char(celldata)
```

```
chr =
```

```
3×13 char array
```

```
'Allison Jones'  
'Development '  
'Phoenix      '
```

For more information on cell arrays, see “Access Data in Cell Array” on page 12-5.

Functions for Cell Arrays of Character Vectors

This table describes the MATLAB functions for working with cell arrays of character vectors.

Function	Description
<code>cellstr</code>	Convert a character array to a cell array of character vectors.
<code>char</code>	Convert a cell array of character vectors to a character array.
<code>string</code>	Starting in R2016b, convert a cell array of character vectors to a string array.
<code>deblank</code>	Remove trailing blanks from a character array.
<code>iscellstr</code>	Return <code>true</code> for a cell array of character arrays.
<code>sort</code>	Sort elements in ascending or descending order.
<code>strcat</code>	Concatenate character arrays or cell arrays of character arrays.
<code>strcmp</code>	Compare character arrays or cell arrays of character arrays.

You can also use the following `set` functions with cell arrays of character vectors.

Function	Description
<code>intersect</code>	Set the intersection of two vectors.

Function	Description
<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.

Analyze Text Data with String Arrays

This example shows how to store text from a file as a string array, sort the words by their frequency, plot the result, and collect basic statistics for the words found in the file.

Import Text File to String Array

Read text from Shakespeare's Sonnets with the `fileread` function. `fileread` returns the text as a 1-by-100266 character vector.

```
sonnets = fileread('sonnets.txt');  
sonnets(1:35)
```

```
ans =  
    'THE SONNETS'  
  
    'by William Shakespeare'
```

Convert the text to a string using the `string` function. Then, split it on newline characters using the `splitlines` function. `sonnets` becomes a 2625-by-1 string array, where each string contains one line from the poems. Display the first five lines of `sonnets`.

```
sonnets = string(sonnets);  
sonnets = splitlines(sonnets);  
sonnets(1:5)
```

```
ans = 5x1 string array  
    "THE SONNETS"  
    ""  
    "by William Shakespeare"  
    ""  
    ""
```

Clean String Array

To calculate the frequency of the words in `sonnets`, first clean it by removing empty strings and punctuation marks. Then reshape it into a string array that contains individual words as elements.

Remove the strings with zero characters ("") from the string array. Compare each element of `sonnets` to "", the empty string. Starting in R2017a, you can create strings,

including an empty string, using double quotes. TF is a logical vector that contains a true value wherever `sonnets` contains a string with zero characters. Index into `sonnets` with TF and delete all strings with zero characters.

```
TF = (sonnets == "");
sonnets(TF) = [];
sonnets(1:10)

ans = 10x1 string array
"THE SONNETS"
"by William Shakespeare"
" I"
" From fairest creatures we desire increase,"
" That thereby beauty's rose might never die,"
" But as the riper should by time decease,"
" His tender heir might bear his memory:"
" But thou, contracted to thine own bright eyes,"
" Feed'st thy light's flame with self-substantial fuel,"
" Making a famine where abundance lies,"
```

Replace some punctuation marks with space characters. For example, replace periods, commas, and semi-colons. Keep apostrophes because they can be part of some words in the Sonnets, such as *light's*.

```
p = [".", "?", "!", ",", ";", ":"];
sonnets = replace(sonnets,p, " ");
sonnets(1:10)

ans = 10x1 string array
"THE SONNETS"
"by William Shakespeare"
" I"
" From fairest creatures we desire increase "
" That thereby beauty's rose might never die "
" But as the riper should by time decease "
" His tender heir might bear his memory "
" But thou contracted to thine own bright eyes "
" Feed'st thy light's flame with self-substantial fuel "
" Making a famine where abundance lies "
```

Strip leading and trailing space characters from each element of `sonnets`.

```
sonnets = strip(sonnets);
sonnets(1:10)

ans = 10x1 string array
    "THE SONNETS"
    "by William Shakespeare"
    "I"
    "From fairest creatures we desire increase"
    "That thereby beauty's rose might never die"
    "But as the ripper should by time decease"
    "His tender heir might bear his memory"
    "But thou contracted to thine own bright eyes"
    "Feed'st thy light's flame with self-substantial fuel"
    "Making a famine where abundance lies"
```

Split `sonnets` into a string array whose elements are individual words. You can use the `split` function to split elements of a string array on whitespace characters, or on delimiters that you specify. However, `split` requires that every element of a string array must be divisible into an equal number of new strings. The elements of `sonnets` have different numbers of spaces, and therefore are not divisible into equal numbers of strings. To use the `split` function on `sonnets`, write a for-loop that calls `split` on one element at a time.

Create the empty string array `sonnetWords` using the `strings` function. Write a for-loop that splits each element of `sonnets` using the `split` function. Concatenate the output from `split` onto `sonnetWords`. Each element of `sonnetWords` is an individual word from `sonnets`.

```
sonnetWords = strings(0);
for i = 1:length(sonnets)
    sonnetWords = [sonnetWords ; split(sonnets(i))];
end
sonnetWords(1:10)

ans = 10x1 string array
    "THE"
    "SONNETS"
    "by"
    "William"
    "Shakespeare"
    "I"
    "From"
    "fairest"
```

```
"creatures"  
"we"
```

Sort Words Based on Frequency

Find the unique words in `sonnetWords`. Count them and sort them based on their frequency.

To count words that differ only by case as the same word, convert `sonnetWords` to lowercase. For example, `The` and `the` count as the same word. Find the unique words using the `unique` function. Then, count the number of times each unique word occurs using the `histcounts` function.

```
sonnetWords = lower(sonnetWords);  
[words,~,idx] = unique(sonnetWords);  
numOccurrences = histcounts(idx,numel(words));
```

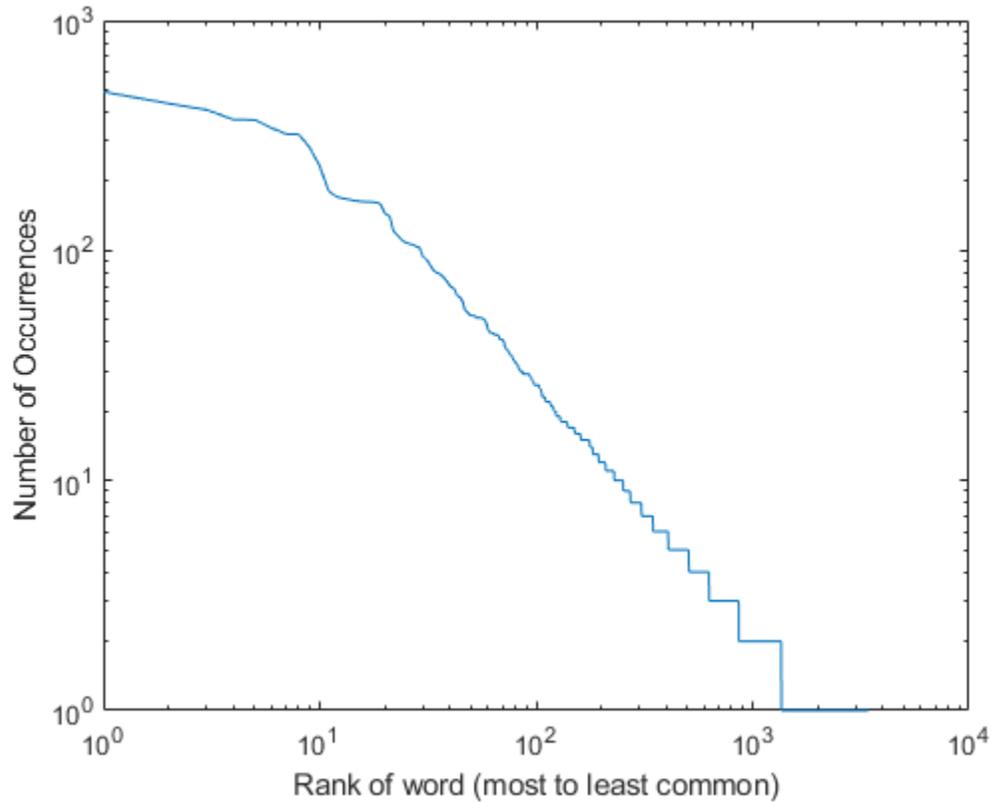
Sort the words in `sonnetWords` by number of occurrences, from most to least common.

```
[rankOfOccurrences,rankIndex] = sort(numOccurrences,'descend');  
wordsByFrequency = words(rankIndex);
```

Plot Word Frequency

Plot the occurrences of words in the Sonnets from the most to least common words. Zipf's Law states that the distribution of occurrences of words in a large body text follows a power-law distribution.

```
loglog(rankOfOccurrences);  
xlabel('Rank of word (most to least common)');  
ylabel('Number of Occurrences');
```



Display the ten most common words in the Sonnets.

```
wordsByFrequency(1:10)
```

```
ans = 10x1 string array
```

```
"and"  
"the"  
"to"  
"my"  
"of"  
"i"  
"in"  
"that"  
"thy"
```

"thou"

Collect Basic Statistics in Table

Calculate the total number of occurrences of each word in `sonnetWords`. Calculate the number of occurrences as a percentage of the total number of words, and calculate the cumulative percentage from most to least common. Write the words and the basic statistics for them to a table.

```
numOccurrences = numOccurrences(rankIndex);
numOccurrences = numOccurrences';
numWords = length(sonnetWords);
T = table;
T.Words = wordsByFrequency;
T.NumOccurrences = numOccurrences;
T.PercentOfText = numOccurrences / numWords * 100.0;
T.CumulativePercentOfText = cumsum(numOccurrences) / numWords * 100.0;
```

Display the statistics for the ten most common words.

```
T(1:10, :)
```

```
ans=10x4 table
```

Words	NumOccurrences	PercentOfText	CumulativePercentOfText
"and"	490	2.7666	2.7666
"the"	436	2.4617	5.2284
"to"	409	2.3093	7.5377
"my"	371	2.0947	9.6324
"of"	370	2.0891	11.722
"i"	341	1.9254	13.647
"in"	321	1.8124	15.459
"that"	320	1.8068	17.266
"thy"	280	1.5809	18.847
"thou"	233	1.3156	20.163

The most common word in the Sonnets, *and*, occurs 490 times. Together, the ten most common words account for 20.163% of the text.

See Also

histcounts | join | lower | replace | sort | split | splitlines | string | strip | table | unique

Related Examples

- “Create String Arrays” on page 6-12
- “Search and Replace Text” on page 6-54
- “Compare Text” on page 6-47
- “Test for Empty Strings and Missing Values” on page 6-31

Test for Empty Strings and Missing Values

String arrays can contain both empty strings and missing values. Empty strings contain zero characters and display as double quotes with nothing between them (""). You can determine if a string is an empty string using the == operator. The empty string is a substring of every other string. Therefore, functions such as `contains` always find the empty string within other strings. String arrays also can contain missing values. Missing values in string arrays display as `<missing>`. To find missing values in a string array, use the `ismissing` function instead of the == operator.

Test for Empty Strings

You can test a string array for empty strings using the == operator.

Starting in R2017a, you can create an empty string using double quotes with nothing between them (""). Note that the size of `str` is 1-by-1, not 0-by-0. However, `str` contains zero characters.

```
str = ""

str =
""
```

Create an empty character vector using single quotes. Note that the size of `chr` is 0-by-0. The character array `chr` actually is an empty array, and not just an array with zero characters.

```
chr = ''

chr =

    0x0 empty char array
```

Create an array of empty strings using the `strings` function. Each element of the array is a string with no characters.

```
str2 = strings(1,3)

str2 = 1x3 string array
      ""    ""    ""
```

Test if `str` is an empty string by comparing it to an empty string.

```
if (str == "")
    disp 'str has zero characters'
end
```

```
str has zero characters
```

Do not use the `isempty` function to test for empty strings. A string with zero characters still has a size of 1-by-1. However, you can test if a string array has at least one dimension with a size of zero using the `isempty` function.

Create an empty string array using the `strings` function. To be an empty array, at least one dimension must have a size of zero.

```
str = strings(0,3)
```

```
str =
```

```
    0x3 empty string array
```

Test `str` using the `isempty` function.

```
isempty(str)
```

```
ans = logical
      1
```

Test a string array for empty strings. The `==` operator returns a logical array that is the same size as the string array.

```
str = ["Mercury", "", "Apollo"]
```

```
str = 1x3 string array
    "Mercury"    ""    "Apollo"
```

```
str == ''
```

```
ans = 1x3 logical array
```

```
    0    1    0
```

Find Empty Strings Within Other Strings

Strings always contain the empty string as a substring. In fact, the empty string is always at both the start and the end of every string. Also, the empty string is always found between any two consecutive characters in a string.

Create a string. Then test if it contains the empty string.

```
str = "Hello, world";
TF = contains(str, "")

TF = logical
    1
```

Test if `str` starts with the empty string.

```
TF = startsWith(str, "")

TF = logical
    1
```

Count the number of characters in `str`. Then count the number of empty strings in `str`. The `count` function counts empty strings at the beginning and end of `str`, and between each pair of characters. Therefore if `str` has `N` characters, it also has `N+1` empty strings.

```
str

str =
"Hello, world"

strlength(str)

ans = 12

count(str, "")

ans = 13
```

Replace a substring with the empty string. When you call `replace` with an empty string, it removes the substring and replaces it with a string that has zero characters.

```
replace(str, "world", "")
```

```
ans =  
"Hello, "
```

Insert a substring after empty strings using the `insertAfter` function. Because there are empty strings between each pair of characters, `insertAfter` inserts substrings between each pair.

```
insertAfter(str, "", "-")
```

```
ans =  
"-H-e-l-l-o-, -w-o-r-l-d-"
```

In general, string functions that replace, erase, extract, or insert substrings allow you to specify empty strings as the starts and ends of the substrings to modify. When you do so, these functions operate on the start and end of the string, and between every pair of characters.

Test for Missing Values

You can test a string array for missing values using the `ismissing` function. The missing string is the string equivalent to `NaN` for numeric arrays. It indicates where a string array has missing values. The missing string displays as `<missing>`.

To create a missing string, convert a missing value using the `string` function.

```
str = string(missing)
```

```
str =  
<missing>
```

You can create a string array with both empty and missing strings. Use the `ismissing` function to determine which elements are strings with missing values. Note that the empty string is not a missing string.

```
str(1) = "";  
str(2) = "Gemini";  
str(3) = string(missing)
```

```
str = 1x3 string array  
    ""    "Gemini"    <missing>
```

```
ismissing(str)
```

```
ans = 1x3 logical array
```

```
0 0 1
```

Compare `str` to a missing string. The comparison is always `0` (false), even when you compare a missing string to another missing string.

```
str == string(missing)
ans = 1x3 logical array
0 0 0
```

To find missing strings, use the `ismissing` function. Do not use the `==` operator.

See Also

`contains` | `endsWith` | `eq` | `erase` | `eraseBetween` | `extractAfter` | `extractBefore` | `extractBetween` | `insertAfter` | `insertBefore` | `ismissing` | `replace` | `replaceBetween` | `startsWith` | `string` | `strings` | `strlength`

Related Examples

- “Create String Arrays” on page 6-12
- “Analyze Text Data with String Arrays” on page 6-24
- “Search and Replace Text” on page 6-54
- “Compare Text” on page 6-47

Formatting Text

To convert data to text and control its format, you can use *formatting operators* with common conversion functions, such as `num2str` and `sprintf`. These operators control notation, alignment, significant digits, and so on. They are similar to those used by the `printf` function in the C programming language. Typical uses for formatted text include text for display and output files.

For example, `%f` converts floating-point values to text using fixed-point notation. Adjust the format by adding information to the operator, such as `%.2f` to represent two digits after the decimal mark, or `%12f` to represent 12 characters in the output, padding with spaces as needed.

```
A = pi*ones(1,3);  
txt = sprintf('%f | %.2f | %12f', A)
```

```
txt =  
'3.141593 | 3.14 |      3.141593'
```

You can combine operators with ordinary text and special characters in a *format specifier*. For instance, `\n` inserts a newline character.

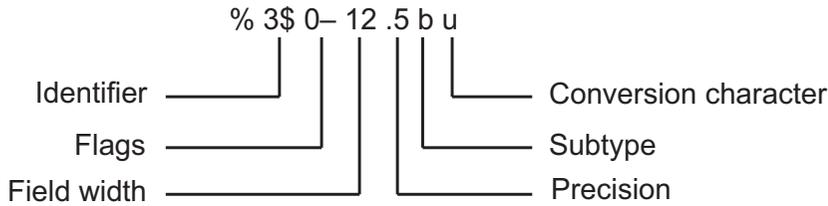
```
txt = sprintf('Displaying pi: \n %f \n %.2f \n %12f', A)
```

```
txt =  
'Displaying pi:  
 3.141593  
 3.14  
      3.141593'
```

Functions that support formatting operators are `compose`, `num2str`, `sprintf`, `fprintf`, and the error handling functions `assert`, `error`, `warning`, and `MException`.

Fields of the Formatting Operator

A formatting operator can have six fields, as shown in the figure. From right to left, the fields are the conversion character, subtype, precision, field width, flags, and numeric identifier. (Space characters are not allowed in the operator. They are shown here only to improve readability of the figure.) The conversion character is the only required field, along with the leading `%` character.



Conversion Character

The conversion character specifies the notation of the output. It consists of a single character and appears last in the format specifier.

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.
g	The more compact of %e or %f. (Insignificant zeroes do not print.)
G	Same as %g, but using an uppercase E.
o	Octal notation (unsigned).
s	Character vector or string array.
u	Decimal notation (unsigned).
x	Hexadecimal notation (unsigned, using lowercase letters a-f).
X	Hexadecimal notation (unsigned, using uppercase letters A-F).

For example, format the number 46 using different conversion characters to display the number in decimal, fixed-point, exponential, and hexadecimal formats.

```
A = 46*ones(1,4);
txt = sprintf('%d %f %e %X', A)

txt =
'46 46.000000 4.600000e+01 2E'
```

Subtype

The subtype field is a single alphabetic character that immediately precedes the conversion character. Without the subtype field, the conversion characters %o, %x, %X, and %u treat input data as integers. To treat input data as floating-point values instead and convert them to octal, decimal, or hexadecimal representations, use one of following subtype specifiers.

b	The input data are double-precision floating-point values rather than unsigned integers. For example, to print a double-precision value in hexadecimal, use a format like %bx.
t	The input data are single-precision floating-point values rather than unsigned integers.

Precision

The precision field in a formatting operator is a nonnegative integer that immediately follows a period. For example, in the operator %7.3f, the precision is 3. For the %g operator, the precision indicates the number of significant digits to display. For the %f, %e, and %E operators, the precision indicates how many digits to display to the right of the decimal point.

Display numbers to different precisions using the precision field.

```
txt = sprintf('%g   %.2g   %f   %.2f', pi*50*ones(1,4))  
  
txt =  
'157.08   1.6e+02   157.079633   157.08'
```

While you can specify the precision in a formatting operator for input text (for example, in the %s operator), there is usually no reason to do so. If you specify the precision as p, and p is less than the number of characters in the input text, then the output contains only the first p characters.

Field Width

The field width in a formatting operator is a nonnegative integer that specifies the number of digits or characters in the output when formatting input values. For example, in the operator %7.3f, the field width is 7.

Specify different field widths. To show the width for each output, use the | character. By default, the output text is padded with space characters when the field width is greater than the number of characters.

```
txt = sprintf(' |%e|%15e|%f|%15f|', pi*50*ones(1,4))
txt =
'|1.570796e+02| 1.570796e+02|157.079633| 157.079633|'
```

When used on text input, the field width can determine whether to pad the output text with spaces. If the field width is less than or equal to the number of characters in the input text, then it has no effect.

```
txt = sprintf('%30s', 'Pad left with spaces')
txt =
'          Pad left with spaces'
```

Flags

Optional flags control additional formatting of the output text. The table describes the characters you can use as flags.

Character	Description	Example
Minus sign (-)	Left-justify the converted argument in its field.	%-5.2d
Plus sign (+)	For numeric values, always print a leading sign character (+ or -). For text values, right-justify the converted argument in its field.	%+5.2d %+5s
Space	Insert a space before the value.	% 5.2f
Zero (0)	Pad with zeroes rather than spaces.	%05.2f

Character	Description	Example
Pound sign (#)	Modify selected numeric conversions: <ul style="list-style-type: none">• For %o, %x, or %X, print 0, 0x, or 0X prefix.• For %f, %e, or %E, print decimal point even when precision is 0.• For %g or %G, do not remove trailing zeroes or decimal point.	##5.0f

Right- and left-justify the output. The default behavior is to right-justify the output text.

```
txt = sprintf('right-justify: %12.2f\nleft-justify: %-12.2f', ...
             12.3, 12.3)
```

```
txt =
'right-justify:          12.30
left-justify: 12.30      '
```

Display a + sign for positive numbers. The default behavior is to omit the leading + sign for positive numbers.

```
txt = sprintf('no sign: %12.2f\nsign: %+12.2f', ...
             12.3, 12.3)
```

```
txt =
'no sign:          12.30
sign:          +12.30'
```

Pad to the left with spaces and zeroes. The default behavior is to pad with spaces.

```
txt = sprintf('Pad with spaces: %12.2f\nPad with zeroes: %012.2f', ...
             5.2, 5.2)
```

```
txt =
'Pad with spaces:          5.20
Pad with zeroes: 000000005.20'
```

Note You can specify more than one flag in a formatting operator.

Value Identifiers

By default, functions such as `sprintf` insert values from input arguments into the output text in sequential order. To process the input arguments in a nonsequential order, specify the order using numeric identifiers in the format specifier. Specify nonsequential arguments with an integer immediately following the `%` sign, followed by a `$` sign.

Ordered Sequentially	Ordered By Identifier
<code>sprintf('%s %s %s', ... '1st', '2nd', '3rd')</code>	<code>sprintf('%3\$s %2\$s %1\$s', ... '1st', '2nd', '3rd')</code>
<code>ans =</code> <code> '1st 2nd 3rd'</code>	<code>ans =</code> <code> '3rd 2nd 1st'</code>

Special Characters

Special characters can be part of the output text. But because they cannot be entered as ordinary text, they require specific character sequences to represent them. To insert special characters into output text, use any of the character sequences in the table.

Special Character	Representation in Format Specifier
Single quotation mark	<code>' '</code>
Percent character	<code>%%</code>
Backslash	<code>\\</code>
Alarm	<code>\a</code>
Backspace	<code>\b</code>
Form feed	<code>\f</code>
New line	<code>\n</code>
Carriage return	<code>\r</code>
Horizontal tab	<code>\t</code>
Vertical tab	<code>\v</code>

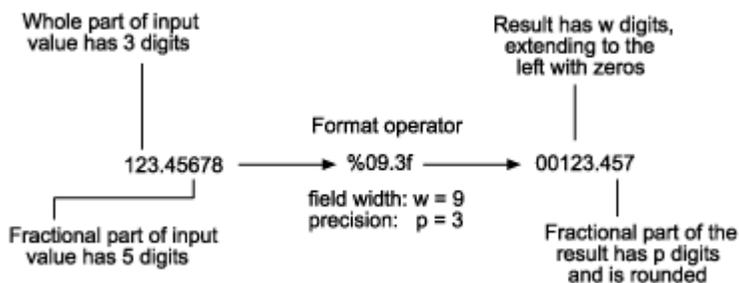
Special Character	Representation in Format Specifier
Character whose Unicode numeric value can be represented by the hexadecimal number, N	\xN Example: <code>printf('\x5A')</code> returns 'Z'
Character whose Unicode numeric value can be represented by the octal number, N	\N Example: <code>printf('\132')</code> returns 'Z'

Setting Field Width and Precision

The formatting operator follows a set of rules for formatting output text to the specified field width and precision. You also can specify values for the field width and precision outside the format specifier, and use numbered identifiers with the field width and precision.

Rules for Formatting Precision and Field Width

The figure illustrates how the field width and precision settings affect the output of the formatting functions. In this figure, the zero following the % sign in the formatting operator means to add leading zeroes to the output text rather than space characters.



- If the precision is not specified, then it defaults to six.
- If the precision p is less than the number of digits in the fractional part of the input, then only p digits are shown after the decimal point. The fractional value is rounded in the output.

- If the precision p is greater than the number of digits f in the fractional part of the input, then p digits are shown after the decimal point. The fractional part is extended to the right with $p - f$ zeroes in the output.
- If the field width is not specified, then it defaults to $p+1+n$, where n is the number of digits in the whole part of the input value.
- If the field width w is greater than $p+1+n$, then the whole part of the output value is padded to the left with $w - (p+1+n)$ additional characters. The additional characters are space characters unless the formatting operator includes the θ flag. In that case, the additional characters are zeroes.

Specify Field Width and Precision Outside Format Specifier

You can specify the field width and precision using values from a sequential argument list. Use an asterisk (*) in place of the field width or precision fields of the formatting operator.

For example, format and display three numbers. In each case, use an asterisk to specify that the field width or precision come from input arguments that follow the format specifier.

```
txt = sprintf('%*f  %.*f  %*.*f', ...
             15, 123.45678, ...
             3, 16.42837, ...
             6, 4, pi)

txt =
'   123.456780   16.428   3.1416'
```

The table describes the effects of each formatting operator in the example.

Formatting Operator	Description
<code>%*f</code>	Specify width as the following input argument, 15.
<code>%.*f</code>	Specify precision as the following input argument, 3.
<code>%*.*f</code>	Specify width and precision as the following input arguments, 6, and 4.

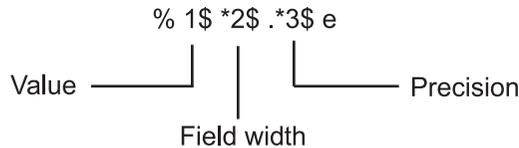
You can mix the two styles. For example, get the field width from the following input argument and the precision from the format specifier.

```
txt = sprintf('%*.2f', 5, 123.45678)

txt =
'123.46'
```

Specify Numbered Identifiers in Width and Precision Fields

You also can specify field width and precision as values from a nonsequential argument list, using an alternate syntax shown in the figure. Within the formatting operator, specify the field width and precision with asterisks that follow numbered identifiers and \$ signs. Specify the values of the field width and precision with input arguments that follow the format specifier.



For example, format and display three numbers. In each case, use a numbered identifier to specify that the field width or precision come from input arguments that follow the format specifier.

```
txt = sprintf('%1$*4$f    %2$.*5$f    %3$*6$.*7$f', ...
             123.45678, 16.42837, pi, 15, 3, 6, 4)

txt =
'123.456780    16.428    3.1416'
```

The table describes the effect of each formatting operator in the example.

Formatting Operator	Description
%1\$*4\$f	1\$ specifies the first input argument, 123.45678, as the value
	*4\$ specifies the fourth input argument, 15, as the field width
%2\$.*5\$f	2\$ specifies the second input argument, 16.42837, as the value
	.*5\$ specifies the fifth input argument, 3, as the precision

Formatting Operator	Description
%3\$*6\$.*7\$f	3\$ specifies the third input argument, pi, as the value
	*6\$ specifies the sixth input argument, 6, as the field width
	. *7\$ specifies the seventh input argument, 4, as the precision

Restrictions on Using Identifiers

If any of the formatting operators include an identifier field, then all the operators in the format specifier must include identifier fields. If you use both sequential and nonsequential ordering in the same function call, then the output is truncated at the first switch between sequential and nonsequential identifiers.

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 function call provides more input arguments than there are formatting operators in the format specifier, then the operators are reused. However, only function calls that use sequential ordering reuse formatting operators. You cannot reuse formatting operators when you use numbered identifiers.

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>

If you use numbered identifiers when the input data is a vector or array, then the output does not contain formatted data.

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 = 1×0 empty char array</pre>

See Also

[compose](#) | [fprintf](#) | [num2str](#) | [sprintf](#)

Related Examples

- “Convert from Character Arrays to Numeric Values” on page 6-63
- “Convert from Numeric Values to Character Array” on page 6-61

Compare Text

Compare text in character arrays and string arrays in different ways. String arrays were introduced in R2016b. You can compare string arrays and character vectors with relational operators and with the `strcmp` function. You can sort string arrays using the `sort` function, just as you would sort arrays of any other type. MATLAB® also provides functions to inspect characters in pieces of text. For example, you can determine which characters in a character vector or string array are letters or space characters.

Compare String Arrays for Equality

You can compare string arrays for equality with the relational operators `==` and `~=`. When you compare string arrays, the output is a logical array that has 1 where the relation is true, and 0 where it is not true.

Create two string scalars. Starting in R2017a, you can create strings using double quotes.

```
str1 = "Hello";  
str2 = "World";  
str1, str2
```

```
str1 =  
"Hello"
```

```
str2 =  
"World"
```

Compare `str1` and `str2` for equality.

```
str1 == str2
```

```
ans = logical
```

```
0
```

Compare a string array with multiple elements to a string scalar.

```
str1 = ["Mercury", "Gemini", "Apollo"; ...  
        "Skylab", "Skylab B", "International Space Station"];  
str2 = "Apollo";  
str1 == str2
```

```
ans = 2x3 logical array
```

```
0 0 1
0 0 0
```

Compare a string array to a character vector. As long as one of the variables is a string array, you can make the comparison.

```
chr = 'Gemini';
TF = (str1 == chr)
```

```
TF = 2x3 logical array
```

```
0 1 0
0 0 0
```

Index into `str1` with `TF` to extract the string elements that matched Gemini. You can use logical arrays to index into an array.

```
str1(TF)
```

```
ans =
"Gemini"
```

Compare for inequality using the `~=` operator. Index into `str1` to extract the elements that do not match 'Gemini'.

```
TF = (str1 ~= chr)
```

```
TF = 2x3 logical array
```

```
1 0 1
1 1 1
```

```
str1(TF)
```

```
ans = 5x1 string array
"Mercury"
"Skylab"
"Skylab B"
"Apollo"
"International Space Station"
```

Compare two nonscalar string arrays. When you compare two nonscalar arrays, they must be the same size.

```
str2 = ["Mercury", "Mars", "Apollo";...
        "Jupiter", "Saturn", "Neptune"];
TF = (str1 == str2)
```

TF = 2x3 logical array

```
 1  0  1
 0  0  0
```

Index into `str1` to extract the matches.

```
str1(TF)
```

```
ans = 2x1 string array
    "Mercury"
    "Apollo"
```

Compare String Arrays with Other Relational Operators

You can also compare strings with the relational operators `>`, `>=`, `<`, and `<=`. Strings that start with uppercase letters come before strings that start with lowercase letters. For example, the string "ABC" is less than "abc". Digits and some punctuation marks also come before letters.

```
"ABC" < "abc"
```

```
ans = logical
```

```
 1
```

Compare a string array that contains names to another name with the `>` operator. The names Sanchez, de Ponte, and Nash come after Matthews, because S, d, and N all are greater than M.

```
str = ["Sanchez", "Jones", "de Ponte", "Crosby", "Nash"];
TF = (str > "Matthews")
```

TF = 1x5 logical array

```
 1  0  1  0  1
```

```
str(TF)
```

```
ans = 1x3 string array
    "Sanchez"    "de Ponte"    "Nash"
```

Sort String Arrays

You can sort string arrays. MATLAB® stores characters as Unicode® using the UTF-16 character encoding scheme. Character and string arrays are sorted according to the UTF-16 code point order. For the characters that are also the ASCII characters, this order means that uppercase letters come before lowercase letters. Digits and some punctuation also come before letters.

Sort the string array `str`.

```
sort(str)
ans = 1x5 string array
    "Crosby"    "Jones"    "Nash"    "Sanchez"    "de Ponte"
```

Sort a 2-by-3 string array. The `sort` function sorts the elements in each column separately.

```
sort(str2)
ans = 2x3 string array
    "Jupiter"    "Mars"    "Apollo"
    "Mercury"    "Saturn"    "Neptune"
```

To sort the elements in each row, sort `str2` along the second dimension.

```
sort(str2,2)
ans = 2x3 string array
    "Apollo"    "Mars"    "Mercury"
    "Jupiter"    "Neptune"    "Saturn"
```

Compare Character Vectors

You can compare character vectors and cell arrays of character vectors to each other. Use the `strcmp` function to compare two character vectors, or `strncmp` to compare the first `N` characters. You also can use `strcmpi` and `strncmpi` for case-insensitive comparisons.

Compare two character vectors with the `strcmp` function. `chr1` and `chr2` are not equal.

```
chr1 = 'hello';
chr2 = 'help';
TF = strcmp(chr1,chr2)
```

```
TF = logical
```

```
0
```

Note that the MATLAB `strcmp` differs from the C version of `strcmp`. The C version of `strcmp` returns 0 when two character arrays are the same, not when they are different.

Compare the first two characters with the `strncmp` function. TF is 1 because both character vectors start with the characters `he`.

```
TF = strncmp(chr1,chr2,2)
```

```
TF = logical
```

```
1
```

Compare two cell arrays of character vectors. `strcmp` returns a logical array that is the same size as the cell arrays.

```
C1 = {'pizza'; 'chips'; 'candy'};
C2 = {'pizza'; 'chocolate'; 'pretzels'};
strcmp(C1,C2)
```

```
ans = 3x1 logical array
```

```
1
```

```
0
```

```
0
```

Inspect Characters in String and Character Arrays

You can inspect the characters in string arrays or character arrays with the `isstrprop`, `isletter`, and `isspace` functions.

- The `isstrprop` inspects characters in either string arrays or character arrays.
- The `isletter` and `isspace` functions inspect characters in character arrays only.

Determine which characters in a character vector are space characters. `isspace` returns a logical vector that is the same size as `chr`.

```
chr = 'Four score and seven years ago';  
TF = isspace(chr)
```

```
TF = 1x30 logical array
```

```
0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1
```

The `isstrprop` function can query characters for many different traits. `isstrprop` can determine whether characters in a string or character vector are letters, alphanumeric characters, decimal or hexadecimal digits, or punctuation characters.

Determine which characters in a string are punctuation marks. `isstrprop` returns a logical vector whose length is equal to the number of characters in `str`.

```
str = "A horse! A horse! My kingdom for a horse!"
```

```
str =  
"A horse! A horse! My kingdom for a horse!"
```

```
isstrprop(str,"punct")
```

```
ans = 1x41 logical array
```

```
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0
```

Determine which characters in the character vector `chr` are letters.

```
isstrprop(chr,"alpha")
```

```
ans = 1x30 logical array
```

```
1 1 1 1 0 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 0
```

See Also

`eq` | `ge` | `gt` | `isletter` | `isspace` | `isstrprop` | `le` | `lt` | `ne` | `sort` | `strcmp`

Related Examples

- “Create Character Arrays” on page 6-6

- “Create String Arrays” on page 6-12
- “Analyze Text Data with String Arrays” on page 6-24
- “Search and Replace Text” on page 6-54
- “Test for Empty Strings and Missing Values” on page 6-31

Search and Replace Text

You can search for text in character arrays and string arrays, and replace substrings with new text. String arrays, and new functions to search for and replace text, were introduced in R2016b. Search for substrings with functions such as the `contains` function. Similarly, replace text in strings with the `replace` function, or extract text with functions such as `extractBetween`. You can use any of these functions with either character vectors or string arrays. For compatibility, you can also use functions such as `strfind` and `strrep` with both character vectors and string arrays.

Search for Text

Identify text in string arrays, character vectors, or cell arrays of character vectors with the `contains`, `startsWith`, and `endsWith` function.

Create a string. Starting in R2017a, you can create strings using double quotes.

```
str = "Rosemary Jones"
```

```
str =  
"Rosemary Jones"
```

Determine whether `str` contains the substring `mary`. The `contains` function returns a logical 1 if it finds the substring any place within the string.

```
TF = contains(str, "mary")
```

```
TF = logical  
    1
```

You can also use the `strfind` function to find matching text. `strfind` returns the index of the start of each match. In this case, `strfind` returns 5 because the `m` in `mary` is the fifth character of `str`.

```
idx = strfind(str, "mary")
```

```
idx = 5
```

Find multiple matches with `strfind`. When there are multiple matches, `strfind` returns the indices as an array.

```
idx = strfind(str, "s")
```

```
idx = 1×2
      3   14
```

Create a string array that contains many names. Determine which names contain the substring Ann. The `contains` function returns a logical array that has a 1 wherever `str` has an element that contains Ann. To create a new string array that includes only the matches, index into `str` with TF.

```
str = ["Rosemary Ann Jones", "Peter Michael Smith", "Ann Marie Young"]
str = 1×3 string array
      "Rosemary Ann Jones"   "Peter Michael Smith"   "Ann Marie Young"
```

```
TF = contains(str, "Ann")
```

```
TF = 1×3 logical array
```

```
      1   0   1
```

```
matches = str(TF)
```

```
matches = 1×2 string array
      "Rosemary Ann Jones"   "Ann Marie Young"
```

Find the strings that begin with Ann.

```
TF = startsWith(str, "Ann");
matches = str(TF)
```

```
matches =
"Ann Marie Young"
```

Similarly, the `endsWith` function find strings that end with a specified piece of text.

You can also use the `contains`, `startsWith`, and `endsWith` functions to determine whether character vectors contains text.

```
chr = 'John Paul Jones'
```

```
chr =
'John Paul Jones'
```

```
TF = contains(chr, 'Paul')
```

```
TF = logical  
    1
```

```
TF = endsWith(chr, 'Paul')
```

```
TF = logical  
    0
```

Use the `contains` function to find text in rows of a string array. `census1905` contains a few rows of simulated census data for the year 1905. Each row contains a name, year of birth, and number of times that name was given in that year.

```
census1905 = ["Ann Mary", "1905", "230";  
             "John", "1905", "5400";  
             "Mary", "1905", "4600";  
             "Maryjane", "1905", "304";  
             "Paul", "1905", "1206"];
```

Find the rows where the name is equal to Mary.

```
TF = (census1905(:,1) == "Mary");  
census1905(TF,:)
```

```
ans = 1x3 string array  
    "Mary"    "1905"    "4600"
```

Find the rows where the name is a variation of Mary with the `contains` function.

```
TF = contains(census1905(:,1), "Mary");  
census1905(TF,:)
```

```
ans = 3x3 string array  
    "Ann Mary"    "1905"    "230"  
    "Mary"        "1905"    "4600"  
    "Maryjane"    "1905"    "304"
```

Replace Text

You can replace text in string arrays, character vectors, or cell arrays of character vectors with the `replace` function.

Create a string. Replace the substring `mary` with `anne`.

```
str = "Rosemary Jones"

str =
"Rosemary Jones"

newStr = replace(str,"mary","anne")

newStr =
"Roseanne Jones"
```

You can also replace text using the `strrep` function. However, the `replace` function is recommended.

```
newStr = strrep(str,"Jones","Day")

newStr =
"Rosemary Day"
```

Create a string array that contains many names.

```
str = ["Rosemary Ann Jones","Peter Michael Smith","Ann Marie Young"]

str = 1x3 string array
    "Rosemary Ann Jones"    "Peter Michael Smith"    "Ann Marie Young"
```

Specify multiple names to replace.

```
oldText = ["Ann","Michael"];
newText = ["Beth","John"];
newStr = replace(str,oldText,newText)

newStr = 1x3 string array
    "Rosemary Beth Jones"    "Peter John Smith"    "Beth Marie Young"
```

Replace text in a character vector. You can use `replace` and `replaceBetween` with character vectors, as well as with strings.

```
chr = 'Mercury, Gemini, Apollo'

chr =
'Mercury, Gemini, Apollo'

replace(chr,'Gemini','Mars')
```

```
ans =  
'Mercury, Mars, Apollo'
```

Replace text in a string array of file names. Append the file names to the address of a website. The file names contain spaces, but spaces cannot be part of web addresses. Replace the space character, " ", with %20, which is the standard for web addresses.

```
str = ["Financial Report.docx";  
      "Quarterly 2015 Details.docx";  
      "Slides.pptx"]
```

```
str = 3x1 string array  
      "Financial Report.docx"  
      "Quarterly 2015 Details.docx"  
      "Slides.pptx"
```

```
newStr = replace(str, " ", "%20")
```

```
newStr = 3x1 string array  
      "Financial%20Report.docx"  
      "Quarterly%202015%20Details.docx"  
      "Slides.pptx"
```

Append the file names to the address of a website.

```
filenames = "http://example.com/Documents/" + newStr
```

```
filenames = 3x1 string array  
      "http://example.com/Documents/Financial%20Report.docx"  
      "http://example.com/Documents/Quarterly%202015%20Details.docx"  
      "http://example.com/Documents/Slides.pptx"
```

Extract Text

Extract a substring from string arrays or character vectors with the `extractAfter`, `extractBefore`, and `extractBetween` functions. Use these functions to extract different substrings that precede, follow, or occur between specified pieces of text.

Create a string array that contains file names. Extract the portions of the names after C:\Temp\ with the `extractAfter` function.

```
str = ["C:\Temp\MyReport.docx";
      "C:\Temp\Data\Sample1.csv";
      "C:\Temp\Slides.pptx"]

str = 3x1 string array
      "C:\Temp\MyReport.docx"
      "C:\Temp\Data\Sample1.csv"
      "C:\Temp\Slides.pptx"

filenames = extractAfter(str, "C:\Temp\")

filenames = 3x1 string array
      "MyReport.docx"
      "Data\Sample1.csv"
      "Slides.pptx"
```

Extract customer names from a string array that encodes the names within XML tags.

```
str = ["<CustomerName>Elizabeth Day</CustomerName>";
      "<CustomerName>George Adams</CustomerName>";
      "<CustomerName>Sarah Young</CustomerName>"]

str = 3x1 string array
      "<CustomerName>Elizabeth Day</CustomerName>"
      "<CustomerName>George Adams</CustomerName>"
      "<CustomerName>Sarah Young</CustomerName>"

names = extractBetween(str, "<CustomerName>", "</CustomerName>")

names = 3x1 string array
      "Elizabeth Day"
      "George Adams"
      "Sarah Young"
```

See Also

contains | endsWith | erase | eraseBetween | extractAfter | extractBefore |
 extractBetween | insertAfter | insertBefore | replace | replaceBetween |
 startsWith | strfind | strrep

Related Examples

- “Create Character Arrays” on page 6-6
- “Create String Arrays” on page 6-12
- “Analyze Text Data with String Arrays” on page 6-24
- “Compare Text” on page 6-47
- “Test for Empty Strings and Missing Values” on page 6-31
- “Regular Expressions” on page 2-41

Convert from Numeric Values to Character Array

In this section...

“Function Summary” on page 6-61

“Convert Numbers to Character Codes” on page 6-62

“Represent Numbers as Text” on page 6-62

“Convert to Specific Radix” on page 6-62

Function Summary

The functions listed in this table provide a number of ways to convert numeric data to character arrays.

Function	Description	Example
char	Convert a positive integer to an equivalent character. (Truncates any fractional parts.)	[72 105] → 'Hi '
string	Convert an array of double-precision values to a string array.	[72 105] → "72" "105" [3.1416 2.178] → "3.1416" "2.178"
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 %ld/)
mat2str	Convert a numeric type to a character type of the specified precision, returning a character vector 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'

Function	Description	Example
dec2base	Convert a positive integer to a character type of any base from 2 through 36.	[72 105] → '110151' (base set to 8)

Convert Numbers to Character Codes

The `char` function converts integers to Unicode character codes and returns a character array composed of the equivalent characters:

```
x = [77 65 84 76 65 66];  
char(x)  
ans =  
  
    'MATLAB'
```

Represent Numbers as Text

The `int2str`, `num2str`, and `mat2str` functions represent numeric values as text 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)  
chr1 = num2str(min(x));  
chr2 = num2str(max(x));  
out = ['Value of f from ' chr1 ' to ' chr2];  
xlabel(out);
```

Convert to Specific Radix

Another class of conversion functions changes numeric values into character arrays representing a decimal value in another base, such as binary or hexadecimal representation. This includes `dec2hex`, `dec2bin`, and `dec2base`.

Convert from Character Arrays to Numeric Values

In this section...

“Function Summary” on page 6-63

“Convert from Character Code” on page 6-64

“Convert Text that Represents Numeric Values” on page 6-64

“Convert from Specific Radix” on page 6-65

Function Summary

The functions listed in this table provide a number of ways to convert character arrays 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 string arrays and cell arrays of character vectors.	"72" "105" → [72 105] {'72' '105'} → [72 105]
hex2num	Convert a numeric type to a character type of specified precision, returning a character array 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 character type of binary number to a decimal number.	'1010' → 10
base2dec	Convert a character type of any base number from 2 through 36 to a decimal number.	'12' → 10 (if base == 8)

Convert from Character Code

Character arrays and string 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 characters 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'
```

Convert Text that Represents Numeric Values

Use `str2num` to convert a character array to the numeric value it represents:

```
chr = '37.294e-1';

val = str2num(chr)
val =

    3.7294
```

The `str2double` function converts a string array or a cell array of character vectors to the double-precision values they represent:

```
c = {'37.294e-1'; '-58.375'; '13.796'};
str = string({'3.14159', '2.718'});

d = str2double(c)

d =

    3.7294
   -58.3750
    13.7960

x = str2double(str)
```

```
x =
```

```
    3.1416    2.7180
```

```
whos
```

Name	Size	Bytes	Class	Attributes
c	3x1	380	cell	
d	3x1	24	double	
str	1x2	196	string	
x	1x2	16	double	

Convert from 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.

Frequently Asked Questions About String Arrays

MATLAB introduced the `string` data type in R2016b. Starting in R2018b, you can use string arrays to work with text throughout MathWorks products. String arrays store pieces of text and provide a set of functions for working with text as data. You can index into, reshape, and concatenate strings arrays just as you can with arrays of any other type. For more information, see “Create String Arrays” on page 6-12.

In most respects, strings arrays behave like character vectors and cell arrays of character vectors. However, there are a few key differences between string arrays and character arrays that can lead to results you might not expect. For each of these differences, there is a recommended way to use strings that leads to the expected result.

Why Does Using Command Form With Strings Return An Error?

When you use functions such as the `cd`, `dir`, `copyfile`, or `load` functions in command form, avoid using double quotes. In command form, arguments enclosed in double quotes can result in errors. To specify arguments as strings, use functional form.

With command syntax, you separate inputs with spaces rather than commas, and you do not enclose input arguments in parentheses. For example, you can use the `cd` function with command syntax to change folders.

```
cd C:\Temp
```

The text `C:\Temp` is a character vector. In command form, all arguments are always character vectors. If you have an argument, such as a folder name, that contains spaces, then specify it as one input argument by enclosing it in single quotes.

```
cd 'C:\Program Files'
```

But if you specify the argument using double quotes, then `cd` throws an error.

```
cd "C:\Program Files"
```

```
Error using cd
Too many input arguments.
```

The error message can vary depending on the function that you use and the arguments that you specify. For example, if you use the `load` function with command syntax and specify the argument using double quotes, then `load` throws a different error.

```
load "myVariables.mat"
```

```
Error using load
Unable to read file 'myVariables.mat': Invalid argument.
```

In command form, double quotes are treated as part of the literal text rather than as the string construction operator. If you wrote the equivalent of `cd "C:\Program Files"` in functional form, then it would look like a call to `cd` with two arguments.

```
cd('C:\Program', 'Files')
```

When specifying arguments as strings, use function syntax. All functions that support command syntax also support function syntax. For example, you can use `cd` with function syntax and input arguments that are double quoted strings.

```
cd("C:\Program Files")
```

Why Do Strings in Cell Arrays Return an Error?

When you have multiple strings, store them in a string array, *not* a cell array. Create a string array using square brackets, not curly braces. String arrays are more efficient than cell arrays for storing and manipulating text.

```
str = ["Venus", "Earth", "Mars"]

str = 1x3 string array
    "Venus"    "Earth"    "Mars"
```

Avoid using cell arrays of strings. When you use cell arrays, you give up the performance advantages that come from using string arrays. And in fact, most functions do not accept cell arrays of strings as input arguments, options, or values of name-value pairs. For example, if you specify a cell array of strings as an input argument, then the `contains` function throws an error.

```
C = {"Venus", "Earth", "Mars"}

C = 1x3 cell array
    {"Venus"} {"Earth"} {"Mars"}

TF = contains(C, "Earth")
```

```
Error using contains
First argument must be a string array, character vector, or cell array of character vectors.
```

Instead, specify the argument as a string array.

```
str = ["Venus", "Earth", "Mars"];  
TF = contains(str, "Earth");
```

Before R2016b, the term "cell array of strings" meant a cell array whose elements all contain character vectors. But it is more precise to refer to such cell arrays as "cell arrays of character vectors," to distinguish them from string arrays.

Cell arrays can contain variables having any data types, including strings. It is still possible to create a cell array whose elements all contain strings. And if you already have specified cell arrays of character vectors in your code, then replacing single quotes with double quotes might seem like a simple update. However, it is not recommended that you create or use cell arrays of strings.

Why Does `length()` of String Return 1?

It is common to use the `length` function to determine the number of characters in a character vector. But to determine the number of characters in a string, use the `strlength` function, not `length`.

Create a character vector using single quotes. To determine its length, use the `length` function. Because `C` is a vector, its length is equal to the number of characters. `C` is a 1-by-11 vector.

```
C = 'Hello world';  
L = length(C)
```

```
L = 11
```

Create a string with the same characters, using double quotes. Though it stores 11 characters, `str` is a 1-by-1 string array, or *string scalar*. If you call `length` on a string scalar, then the output argument is 1, no matter how many characters it stores.

```
str = "Hello World";  
L = length(str)
```

```
L = 1
```

To determine the number of characters in a string, use the `strlength` function, introduced in R2016b. For compatibility, `strlength` operates on character vectors as well. In both cases `strlength` returns the number of characters.

```
L = strlength(C)
```

```
L = 11
```

```
L = strlen(str)
```

```
L = 11
```

You also can use `strlen` on string arrays containing multiple strings and on cell arrays of character vectors.

The `length` function returns the size of the longest dimension of an array. For a string array, `length` returns the number of *strings* along the longest dimension of the array. It does not return the number of characters *within* strings.

Why Does `isempty("")` Return 0?

A string can have no characters at all. Such a string is an *empty string*. You can specify an empty string using an empty pair of double quotes.

```
L = strlen("")
```

```
L = 0
```

However, an empty string is *not* an empty array. An empty string is a string scalar that happens to have no characters.

```
sz = size("")
```

```
sz = 1x2  
    1    1
```

If you call `isempty` on an empty string, then it returns 0 (false) because the string is not an empty array.

```
tf = isempty("")
```

```
tf = logical  
    0
```

However, if you call `isempty` on an empty character array, then it returns 1 (true). A character array specified as a empty pair of single quotes, `''`, is a 0-by-0 character array.

```
tf = isempty('')
```

```
tf = logical  
    1
```

To test whether a piece of text has no characters, the best practice is to use the `strlength` function. You can use the same call whether the input is a string scalar or a character vector.

```
str = "";  
if strlength(str) == 0  
    disp('String has no text')  
end
```

```
String has no text
```

```
chr = '';  
if strlength(chr) == 0  
    disp('Character vector has no text')  
end
```

```
Character vector has no text
```

Why Does Appending Strings Using Square Brackets Return Multiple Strings?

You can append text to a character vector using square brackets. But if you add text to a string array using square brackets, then the new text is concatenated as new elements of the string array. To append text to strings, use the `plus` operator or the `strcat` function.

For example, if you concatenate two strings, then the result is a 1-by-2 string array.

```
str = ["Hello" "World"]  
  
str = 1x2 string array  
    "Hello"    "World"
```

However, if you concatenate two character vectors, then the result is a longer character vector.

```
str = ['Hello' 'World']  
  
chr = 'HelloWorld'
```

To append text to a string (or to the elements of a string array), use the `plus` operator instead of square brackets.

```
str = "Hello" + "World"  
  
str = "HelloWorld"
```

As an alternative, you can use the `strcat` function. `strcat` appends text whether the input arguments are strings or character vectors.

```
str = strcat("Hello","World")  
str = "HelloWorld"
```

Whether you use square brackets, `plus`, or `strcat`, you can specify an arbitrary number of arguments. Append a space character between `Hello` and `World`.

```
str = "Hello" + " " + "World"  
str = "Hello World"
```

See Also

`cd` | `contains` | `copyfile` | `dir` | `isempty` | `length` | `load` | `plus` | `size` | `sprintf` | `strcat` | `string` | `strlength`

Related Examples

- “Create String Arrays” on page 6-12
- “Test for Empty Strings and Missing Values” on page 6-31
- “Compare Text” on page 6-47
- “Update Your Code to Accept Strings” on page 6-72

Update Your Code to Accept Strings

In R2016b, MATLAB introduced string arrays as a data type for text. In a future release, all MathWorks products will be *compatible* with string arrays. Compatible means that if you can specify text as a character vector or a cell array of character vectors, then you also can specify it as a string array. Now you can adopt string arrays as a text data type in your own code.

If you write code for other MATLAB users, then it is to your advantage to update your API to accept string arrays, while maintaining backward compatibility with other text data types. String adoption makes your code consistent with MathWorks products.

If your code has few dependencies, or if you are developing new code, then consider using string arrays as your primary text data type for better performance. In that case, best practice is to write or update your API to accept input arguments that are character vectors, cell arrays of character vectors, or string arrays.

For the definitions of string array and other terms, see “Terminology for Character and String Arrays” on page 6-80.

What Are String Arrays?

In MATLAB, you can store text data in two ways. One way is to use a character array, which is a sequence of characters, just as a numeric array is a sequence of numbers. Or, starting in R2016b, the other way is to store a sequence of characters in a *string*. You can store multiple strings in a *string array*. For more information, see “Characters and Strings”.

Recommended Approaches for String Adoption in Old APIs

When your code has many dependencies, and you must maintain backward compatibility, follow these approaches for updating functions and classes to present a compatible API.

Functions

- Accept string arrays as input arguments.
 - If an input argument can be either a character vector or a cell array of character vectors, then update your code so that the argument also can be a string array. For example, consider a function that has an input argument you can specify as a

character vector (using single quotes). Best practice is to update the function so that the argument can be specified as either a character vector or a string scalar (using double quotes).

- Accept strings as both names and values in name-value pair arguments.
 - In name-value pair arguments, allow names to be specified as either character vectors or strings—that is, with either single or double quotes around the name. If a value can be a character vector or cell array of character vectors, then update your code so that it also can be a string array.
- Do not accept cell arrays of string arrays for text input arguments.
 - A cell array of string arrays has a string array in each cell. For example, `{"hello", "world"}` is a cell array of string arrays. While you can create such a cell array, it is not recommended for storing text. The elements of a string array have the same data type and are stored efficiently. If you store strings in a cell array, then you lose the advantages of using a string array.

However, if your code accepts heterogeneous cell arrays as inputs, then consider accepting cell arrays that contain strings. You can convert any strings in such a cell array to character vectors.

- In general, do not change the output type.
 - If your function returns a character vector or cell array of character vectors, then do not change the output type, even if the function accepts string arrays as inputs. For example, the `fileread` function accepts an input file name specified as either a character vector or a string, but the function returns the file contents as a character vector. By keeping the output type the same, you can maintain backward compatibility.
- Return the same data type when the function modifies input text.
 - If your function modifies input text and returns the modified text as the output argument, then the input and output arguments should have the same data type. For example, the `lower` function accepts text as the input argument, converts it to all lowercase letters, and returns it. If the input argument is a character vector, then `lower` returns a character vector. If the input is a string array, then `lower` returns a string array.
- Consider adding a `'TextType'` argument to import functions.
 - If your function imports data from files, and at least some of that data can be text, then consider adding an input argument that specifies whether to return text as a

character array or a string array. For example, the `readtable` function provides the `'TextType'` name-value pair argument. This argument specifies whether `readtable` returns a table with text in cell arrays of character vectors or string arrays.

Classes

- Treat methods as functions.
 - For string adoption, treat methods as though they are functions. Accept string arrays as input arguments, and in general, do not change the data type of the output arguments, as described in the previous section.
- Do not change the data types of properties.
 - If a property is a character vector or a cell array of character vectors, then do not change its type. When you access such a property, the value that is returned is still a character vector or a cell array of character vectors.

As an alternative, you can add a new property that is a string, and make it dependent on the old property to maintain compatibility.

- Set properties using string arrays.
 - If you can set a property using a character vector or cell array of character vectors, then update your class to set that property using a string array too. However, do not change the data type of the property. Instead, convert the input string array to the data type of the property, and then set the property.
- Add a `string` method.
 - If your class already has a `char` and/or a `cellstr` method, then add a `string` method. If you can represent an object of your class as a character vector or cell array of character vectors, then represent it as a string array too.

How to Adopt String Arrays in Old APIs

You can adopt strings in old APIs by accepting string arrays as input arguments, and then converting them to character vectors or cell arrays of character vectors. If you perform such a conversion at the start of a function, then you do not need to update the rest of it.

The `convertStringsToChars` function provides a way to process all input arguments, converting only those arguments that are string arrays. To enable your existing code to

accept string arrays as inputs, add a call to `convertStringsToChars` at the beginnings of your functions and methods.

For example, if you have defined a function `myFunc` that accepts three input arguments, process all three inputs using `convertStringsToChars`. Leave the rest of your code unaltered.

```
function y = myFunc(a,b,c)
    [a,b,c] = convertStringsToChars(a,b,c);
    <line 1 of original code>
    <line 2 of original code>
    ...
```

In this example, the arguments `[a, b, c]` overwrite the input arguments in place. If any input argument is not a string array, then it is unaltered.

If `myFunc` accepts a variable number of input arguments, then process all the arguments specified by `varargin`.

```
function y = myFunc(varargin)
    [varargin{:}] = convertStringsToChars(varargin{:});
    ...
```

Performance Considerations

The `convertStringsToChars` function is more efficient when converting one input argument. If your function is performance sensitive, then you can convert input arguments one at a time, while still leaving the rest of your code unaltered.

```
function y = myFunc(a,b,c)
    a = convertStringsToChars(a);
    b = convertStringsToChars(b);
    c = convertStringsToChars(c);
    ...
```

Recommended Approaches for String Adoption in New Code

When your code has few dependencies, or you are developing entirely new code, consider using string arrays as the primary text data type. String arrays provide good performance and efficient memory usage when working with large amounts of text. Unlike cell arrays of character vectors, string arrays have a homogenous data type. String arrays make it easier to write maintainable code. To use string arrays while maintaining backward compatibility to other text data types, follow these approaches.

Functions

- Accept any text data types as input arguments.
 - If an input argument can be a string array, then also allow it to be a character vector or cell array of character vectors.
- Accept character arrays as both names and values in name-value pair arguments.
 - In name-value pair arguments, allow names to be specified as either character vectors or strings—that is, with either single or double quotes around the name. If a value can be a string array, then also allow it to be a character vector or cell array of character vectors.
- Do not accept cell arrays of string arrays for text input arguments.
 - A cell array of string arrays has a string array in each cell. While you can create such a cell array, it is not recommended for storing text. If your code uses strings as the primary text data type, store multiple pieces of text in a string array, not a cell array of string arrays.

However, if your code accepts heterogeneous cell arrays as inputs, then consider accepting cell arrays that contain strings.

- In general, return strings.
 - If your function returns output arguments that are text, then return them as string arrays.
- Return the same data type when the function modifies input text.
 - If your function modifies input text and returns the modified text as the output argument, then the input and output arguments should have the same data type.

Classes

- Treat methods as functions.
 - Accept character vectors and cell arrays of character vectors as input arguments, as described in the previous section. In general, return strings as outputs.
- Specify properties as string arrays.
 - If a property contains text, then set the property using a string array. When you access the property, return the value as a string array.

How to Maintain Compatibility in New Code

When you write new code, or modify code to use string arrays as the primary text data type, maintain backward compatibility with other text data types. You can accept character vectors or cell arrays of character vectors as input arguments, and then immediately convert them to string arrays. If you perform such a conversion at the start of a function, then the rest of your code can use string arrays only.

The `convertCharsToStrings` function provides a way to process all input arguments, converting only those arguments that are character vectors or cell arrays of character vectors. To enable your new code to accept these text data types as inputs, add a call to `convertCharsToStrings` at the beginnings of your functions and methods.

For example, if you have defined a function `myFunc` that accepts three input arguments, process all three inputs using `convertCharsToStrings`.

```
function y = myFunc(a,b,c)
    [a,b,c] = convertCharsToStrings(a,b,c);
    <line 1 of original code>
    <line 2 of original code>
    ...
```

In this example, the arguments `[a, b, c]` overwrite the input arguments in place. If any input argument is not a character vector or cell array of character vectors, then it is unaltered.

If `myFunc` accepts a variable number of input arguments, then process all the arguments specified by `varargin`.

```
function y = myFunc(varargin)
    [varargin{:}] = convertCharsToStrings(varargin{:});
    ...
```

Performance Considerations

The `convertCharsToStrings` function is more efficient when converting one input argument. If your function is performance sensitive, then you can convert input arguments one at a time, while still leaving the rest of your code unaltered.

```
function y = myFunc(a,b,c)
    a = convertCharsToStrings(a);
    b = convertCharsToStrings(b);
    c = convertCharsToStrings(c);
    ...
```

How to Manually Convert Input Arguments

If it is at all possible, **avoid** manual conversion of input arguments that contain text, and instead use the `convertStringsToChars` or `convertCharsToStrings` functions. Checking the data types of input arguments and converting them yourself is a tedious approach, prone to errors.

If you must convert input arguments, then use the functions in this table.

Conversion	Function
String scalar to character vector	<code>char</code>
String array to cell array of character vectors	<code>cellstr</code>
Character vector to string scalar	<code>string</code>
Cell array of character vectors to string array	<code>string</code>

How to Check Argument Data Types

To check the data type of an input argument that could contain text, consider using the patterns shown in this table.

Required Input Argument Type	Old Check	New Check
Character vector or string scalar	<code>ischar(X)</code>	<code>ischar(X) isStringScalar(X)</code>
		<code>validateattributes(X, {'char','string'}, {'scalartext'})</code>
Character vector or string scalar	<code>validateattributes(X, {'char'}, {'row'})</code>	<code>validateattributes(X, {'char','string'}, {'scalartext'})</code>
Nonempty character vector or string scalar	<code>ischar(X) && ~isempty(X)</code>	<code>(ischar(X) isStringScalar(X)) && strlength(X) ~= 0</code>

Required Input Argument Type	Old Check	New Check
		<code>(ischar(X) isStringScalar(X)) && X ~= ""</code>
Cell array of character vectors or string array	<code>iscellstr(X)</code>	<code>iscellstr(X) isstring(X)</code>
Any text data type	<code>ischar(X) iscellstr(X)</code>	<code>ischar(X) iscellstr(X) isstring(X)</code>

Check for Empty Strings

An *empty string* is a string with no characters. MATLAB displays an empty string as a pair of double quotes with nothing between them (""). However, an empty string is still a 1-by-1 string array. It is **not** an empty array.

The recommended way to check whether a string is empty is to use the `strlength` function.

```
str = "";
tf = (strlength(str) ~= 0)
```

Note Do **not** use the `isempty` function to check for an empty string. An empty string has no characters but is still a 1-by-1 string array.

The `strlength` function returns the length of each string in a string array. If the string must be a string scalar, and also not empty, then check for both conditions.

```
tf = (isStringScalar(str) && strlength(str) ~= 0)
```

If `str` could be either a character vector or string scalar, then you still can use `strlength` to determine its length. `strlength` returns 0 if the input argument is an empty character vector ('').

```
tf = ((ischar(str) || isStringScalar(str)) && strlength(str) ~= 0)
```

Check for Empty String Arrays

An *empty string array* is, in fact, an empty array—that is, an array that has at least one dimension whose length is 0.

The recommended way to create an empty string array is to use the `strings` function, specifying 0 as at least one of the input arguments. The `isempty` function returns 1 when the input is an empty string array.

```
str = strings(0);  
tf = isempty(str)
```

The `strlength` function returns a numeric array that is the same size as the input string array. If the input is an empty string array, then `strlength` returns an empty array.

```
str = strings(0);  
L = strlength(str)
```

Check for Missing Strings

String arrays also can contain *missing strings*. The missing string is the string equivalent to NaN for numeric arrays. It indicates where a string array has missing values. The missing string displays as `<missing>`, with no quotation marks.

You can create missing strings using the `missing` function. The recommended way to check for missing strings is to use the `ismissing` function.

```
str = string(missing);  
tf = ismissing(str)
```

Note Do **not** check for missing strings by comparing a string to the missing string.

The missing string is not equal to itself, just as NaN is not equal to itself.

```
str = string(missing);  
f = (str == missing)
```

Terminology for Character and String Arrays

MathWorks documentation uses these terms to describe character and string arrays. For consistency, use these terms in your own documentation, error messages, and warnings.

- Character vector — 1-by-n array of characters, of data type `char`.
- Character array — m-by-n array of characters, of data type `char`.
- Cell array of character vectors — Cell array in which each cell contains a character vector.
- String *or* string scalar — 1-by-1 string array. A string scalar can contain a 1-by-n sequence of characters, but is itself one object. Use the terms "string scalar" and "character vector" alongside each other when to be precise about size and data type. Otherwise, you can use the term "string" in descriptions.
- String vector — 1-by-n or n-by-1 string array. If only one size is possible, then use it in your description. For example, use "1-by-n string array" to describe an array of that size.
- String array — m-by-n string array.
- Empty string — String scalar that has no characters.
- Empty string array — String array with at least one dimension whose size is 0.
- Missing string — String scalar that is the missing value (displays as `<missing>`).

See Also

`cellstr` | `char` | `convertCharsToStrings` | `convertContainedStringsToChars` | `convertStringsToChars` | `isStringScalar` | `iscellstr` | `ischar` | `isstring` | `string` | `strings` | `strlength` | `validateattributes`

More About

- "Create String Arrays" on page 6-12
- "Test for Empty Strings and Missing Values" on page 6-31
- "Compare Text" on page 6-47
- "Search and Replace Text" on page 6-54
- "Frequently Asked Questions About String Arrays" on page 6-66

Function Summary

MATLAB provides these functions for working with character arrays:

- Functions to Create Character Arrays
- Functions to Modify Character Arrays
- Functions to Read and Operate on Character Arrays
- Functions to Search or Compare Character Arrays
- Functions to Determine Class or Content
- Functions to Convert Between Numeric and Text Data Types
- Functions to Work with Cell Arrays of Character Vectors as Sets

Functions to Create Character Arrays

Function	Description
'chr'	Create the character vector specified between quotes.
blanks	Create a character vector of blanks.
sprintf	Write formatted data as text.
strcat	Concatenate character arrays.
char	Concatenate character arrays 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 character array.
strrep	Replace text within a character array.
strtrim	Remove leading and trailing white space.
upper	Make all letters uppercase.

Functions to Read and Operate on Character Arrays

Function	Description
<code>eval</code>	Execute a MATLAB expression.
<code>sscanf</code>	Read a character array under format control.

Functions to Search or Compare Character Arrays

Function	Description
<code>regex</code>	Match regular expression on page 2-41.
<code>strcmp</code>	Compare character arrays.
<code>strcmpi</code>	Compare character arrays, ignoring case.
<code>strfind</code>	Find a term within a character vector.
<code>strncmp</code>	Compare the first N characters of character arrays.
<code>strncmpi</code>	Compare the first N characters, ignoring case.
<code>strtok</code>	Find a token in a character vector.
<code>textscan</code>	Read data from a character array.

Functions to Determine Class or Content

Function	Description
<code>iscellstr</code>	Return <code>true</code> for a cell array of character vectors.
<code>ischar</code>	Return <code>true</code> for a character array.
<code>isletter</code>	Return <code>true</code> for letters of the alphabet.
<code>isstrprop</code>	Determine if a string is of the specified category.
<code>isspace</code>	Return <code>true</code> for white-space characters.

Functions to Convert Between Numeric and Text Data Types

Function	Description
<code>char</code>	Convert to a character array.
<code>cellstr</code>	Convert a character array to a cell array of character vectors.
<code>double</code>	Convert a character array to numeric codes.
<code>int2str</code>	Represent an integer as text.
<code>mat2str</code>	Convert a matrix to a character array you can run <code>eval</code> on.
<code>num2str</code>	Represent a number as text.
<code>str2num</code>	Convert a character vector to the number it represents.
<code>str2double</code>	Convert a character vector to the double-precision value it represents.

Functions to Work with Cell Arrays of Character Vectors as Sets

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.

Dates and Time

- “Represent Dates and Times in MATLAB” on page 7-2
- “Specify Time Zones” on page 7-6
- “Convert Date and Time to Julian Date or POSIX Time” on page 7-8
- “Set Date and Time Display Format” on page 7-12
- “Generate Sequence of Dates and Time” on page 7-17
- “Share Code and Data Across Locales” on page 7-25
- “Extract or Assign Date and Time Components of Datetime Array” on page 7-28
- “Combine Date and Time from Separate Variables” on page 7-32
- “Date and Time Arithmetic” on page 7-34
- “Compare Dates and Time” on page 7-41
- “Plot Dates and Durations” on page 7-45
- “Core Functions Supporting Date and Time Arrays” on page 7-52
- “Convert Between Datetime Arrays, Numbers, and Text” on page 7-53
- “Carryover in Date Vectors and Strings” on page 7-59
- “Converting Date Vector Returns Unexpected Output” on page 7-60

Represent Dates and Times in MATLAB

The primary way to store date and time information is in `datetime` arrays, which support arithmetic, sorting, comparisons, plotting, and formatted display. The results of arithmetic differences are returned in `duration` arrays or, when you use calendar-based functions, in `calendarDuration` arrays.

For example, create a MATLAB `datetime` array that represents two dates: June 28, 2014 at 6 a.m. and June 28, 2014 at 7 a.m. Specify numeric values for the year, month, day, hour, minute, and second components for the `datetime`.

```
t = datetime(2014,6,28,6:7,0,0)
t =
    28-Jun-2014 06:00:00    28-Jun-2014 07:00:00
```

Change the value of a date or time component by assigning new values to the properties of the `datetime` array. For example, change the day number of each `datetime` by assigning new values to the `Day` property.

```
t.Day = 27:28
t =
    27-Jun-2014 06:00:00    28-Jun-2014 07:00:00
```

Change the display format of the array by changing its `Format` property. The following format does not display any time components. However, the values in the `datetime` array do not change.

```
t.Format = 'MMM dd, yyyy'
t =
    Jun 27, 2014    Jun 28, 2014
```

If you subtract one `datetime` array from another, the result is a `duration` array in units of fixed length.

```
t2 = datetime(2014,6,29,6,30,45)
t2 =
    29-Jun-2014 06:30:45
d = t2 - t
```

```
d =
    48:30:45    23:30:45
```

By default, a `duration` array displays in the format, hours:minutes:seconds. Change the display format of the duration by changing its `Format` property. You can display the duration value with a single unit, such as hours.

```
d.Format = 'h'
d =
    48.512 hrs    23.512 hrs
```

You can create a duration in a single unit using the `seconds`, `minutes`, `hours`, `days`, or `years` functions. For example, create a duration of 2 days, where each day is exactly 24 hours.

```
d = days(2)
d =
    2 days
```

You can create a calendar duration in a single unit of variable length. For example, one month can be 28, 29, 30, or 31 days long. Specify a calendar duration of 2 months.

```
L = calmonths(2)
L =
    2mo
```

Use the `caldays`, `calweeks`, `calquarters`, and `calyears` functions to specify calendar durations in other units.

Add a number of calendar months and calendar days. The number of days remains separate from the number of months because the number of days in a month is not fixed, and cannot be determined until you add the calendar duration to a specific datetime.

```
L = calmonths(2) + caldays(35)
L =
    2mo 35d
```

Add calendar durations to a datetime to compute a new date.

```
t2 = t + calmonths(2) + caldays(35)
```

```
t2 =  
    Oct 01, 2014    Oct 02, 2014
```

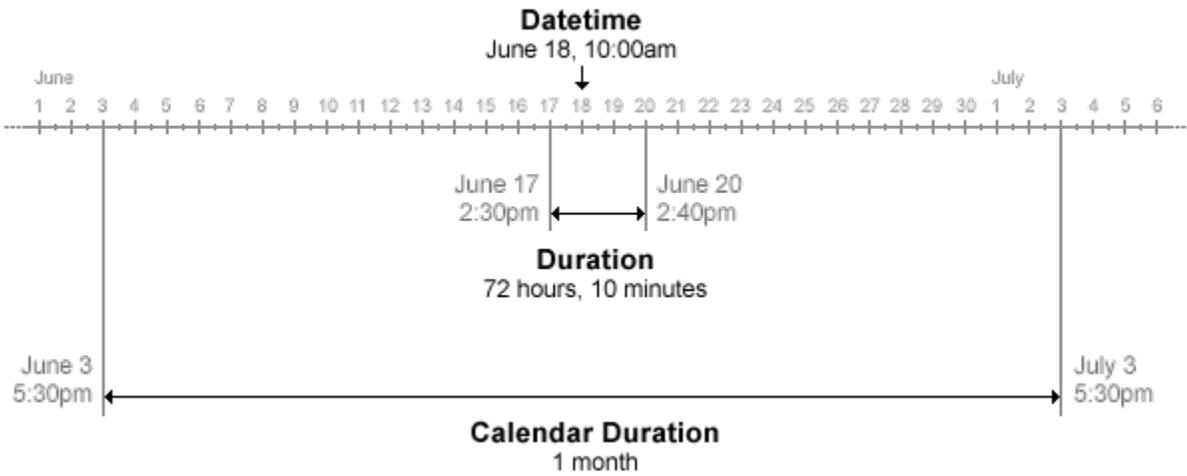
t2 is also a `datetime` array.

```
whos t2
```

Name	Size	Bytes	Class	Attributes
t2	1x2	161	datetime	

In summary, there are several ways to represent dates and times, and MATLAB has a data type for each approach:

- Represent a point in time, using the `datetime` data type.
Example: Wednesday, June 18, 2014 10:00:00
- Represent a length of time, or a duration in units of fixed length, using the `duration` data type. When using the `duration` data type, 1 day is always equal to 24 hours, and 1 year is always equal to 365.2425 days.
Example: 72 hours and 10 minutes
- Represent a length of time, or a duration in units of variable length, using the `calendarDuration` data type.
Example: 1 month, which can be 28, 29, 30, or 31 days long.
The `calendarDuration` data type also accounts for daylight saving time changes and leap years, so that 1 day might be more or less than 24 hours, and 1 year can have 365 or 366 days.



See Also

`calendarDuration` | `datetime` | `duration`

Specify Time Zones

In MATLAB, a time zone includes the time offset from Coordinated Universal Time (UTC), the daylight saving time offset, and a set of historical changes to those values. The time zone setting is stored in the `TimeZone` property of each `datetime` array. When you create a `datetime`, it is unzoned by default. That is, the `TimeZone` property of the `datetime` is empty (`''`). If you do not work with `datetime` values from multiple time zones and do not need to account for daylight saving time, you might not need to specify this property.

You can specify a time zone when you create a `datetime`, using the `'TimeZone'` name-value pair argument. The time zone value `'local'` specifies the system time zone. To display the time zone offset for each `datetime`, include a time zone offset specifier such as `'Z'` in the value for the `'Format'` argument.

```
t = datetime(2014,3,8:9,6,0,0, 'TimeZone', 'local', ...
            'Format', 'd-MMM-y HH:mm:ss Z')

t =

    8-Mar-2014 06:00:00 -0500    9-Mar-2014 06:00:00 -0400
```

A different time zone offset is displayed depending on whether the `datetime` occurs during daylight saving time.

You can modify the time zone of an existing `datetime`. For example, change the `TimeZone` property of `t` using dot notation. You can specify the time zone value as the name of a time zone region in the IANA Time Zone Database. A time zone region accounts for the current and historical rules for standard and daylight offsets from UTC that are observed in that geographic region.

```
t.TimeZone = 'Asia/Shanghai'

t =

    8-Mar-2014 19:00:00 +0800    9-Mar-2014 18:00:00 +0800
```

You also can specify the time zone value as a character vector of the form `+HH:mm` or `-HH:mm`, which represents a time zone with a fixed offset from UTC that does not observe daylight saving time.

```
t.TimeZone = '+08:00'
```

```
t =  
    8-Mar-2014 19:00:00 +0800    9-Mar-2014 18:00:00 +0800
```

Operations on datetime arrays with time zones automatically account for time zone differences. For example, create a datetime in a different time zone.

```
u = datetime(2014,3,9,6,0,0,'TimeZone','Europe/London',...  
    'Format','d-MMM-y HH:mm:ss Z')
```

```
u =  
    9-Mar-2014 06:00:00 +0000
```

View the time difference between the two datetime arrays.

```
dt = t - u  
dt =  
    -19:00:00    04:00:00
```

When you perform operations involving datetime arrays, the arrays either must all have a time zone associated with them, or they must all have no time zone.

See Also

[datetime](#) | [timezones](#)

Related Examples

- “Represent Dates and Times in MATLAB” on page 7-2
- “Convert Date and Time to Julian Date or POSIX Time” on page 7-8

Convert Date and Time to Julian Date or POSIX Time

You can convert `datetime` arrays to represent points in time in specialized numeric formats. In general, these formats represent a point in time as the number of seconds or days that have elapsed since a specified starting point. For example, the Julian date is the number of days and fractional days that have elapsed since the beginning of the Julian period. The POSIX[®] time is the number of seconds that have elapsed since 00:00:00 1-Jan-1970 UTC (Universal Coordinated Time). MATLAB[®] provides the `juliandate` and `posixtime` functions to convert `datetime` arrays to Julian dates and POSIX times.

While `datetime` arrays are not required to have a time zone, converting "unzoned" `datetime` values to Julian dates or POSIX times can lead to unexpected results. To ensure the expected result, specify the time zone before conversion.

Specify Time Zone Before Conversion

You can specify a time zone for a `datetime` array, but you are not required to do so. In fact, by default the `datetime` function creates an "unzoned" `datetime` array.

Create a `datetime` value for the current date and time.

```
d = datetime('now')  
  
d = datetime  
    27-Aug-2018 17:59:32
```

`d` is constructed from the local time on your machine and has no time zone associated with it. In many contexts, you might assume that you can treat the times in an unzoned `datetime` array as local times. However, the `juliandate` and `posixtime` functions treat the times in unzoned `datetime` arrays as UTC times, not local times. To avoid any ambiguity, it is recommended that you avoid using `juliandate` and `posixtime` on unzoned `datetime` arrays. For example, avoid using `posixtime(datetime('now'))` in your code.

If your `datetime` array has values that do not represent UTC times, specify the time zone using the `TimeZone` name-value pair argument so that `juliandate` and `posixtime` interpret the `datetime` values correctly.

```
d = datetime('now','TimeZone','America/New_York')
```

```
d = datetime
    27-Aug-2018 17:59:32
```

As an alternative, you can specify the `TimeZone` property after you create the array.

```
d.TimeZone = 'America/Los_Angeles'
```

```
d = datetime
    27-Aug-2018 14:59:32
```

To see a complete list of time zones, use the `timezones` function.

Convert Zoned and Unzoned Datetime Values to Julian Dates

A Julian date is the number of days (including fractional days) since noon on November 24, 4714 BCE, in the proleptic Gregorian calendar, or January 1, 4713 BCE, in the proleptic Julian calendar. To convert `datetime` arrays to Julian dates, use the `juliandate` function.

Create a `datetime` array and specify its time zone.

```
DZ = datetime('2016-07-29 10:05:24') + calmonths(1:3);
DZ.TimeZone = 'America/New_York'
```

```
DZ = 1x3 datetime array
    29-Aug-2016 10:05:24    29-Sep-2016 10:05:24    29-Oct-2016 10:05:24
```

Convert `D` to the equivalent Julian dates.

```
format longG
JDZ = juliandate(DZ)
```

```
JDZ = 1x3
```

```
2457630.08708333    2457661.08708333    2457691.08708333
```

Create an unzoned copy of `DZ`. Convert `D` to the equivalent Julian dates. As `D` has no time zone, `juliandate` treats the times as UTC times.

```
D = DZ;
D.TimeZone = '';
JD = juliandate(D)
```

```
JD = 1x3
      2457629.92041667      2457660.92041667      2457690.92041667
```

Compare JDZ and JD. The differences are equal to the time zone offset between UTC and the `America/New_York` time zone in fractional days.

```
JDZ - JD
ans = 1x3
      0.166666666511446      0.166666666511446      0.166666666511446
```

Convert Zoned and Unzoned Datetime Values to POSIX Times

The POSIX time is the number of seconds (including fractional seconds) elapsed since 00:00:00 1-Jan-1970 UTC (Universal Coordinated Time), ignoring leap seconds. To convert `datetime` arrays to POSIX times, use the `posixtime` function.

Create a `datetime` array and specify its time zone.

```
DZ = datetime('2016-07-29 10:05:24') + calmonths(1:3);
DZ.TimeZone = 'America/New_York'
```

```
DZ = 1x3 datetime array
      29-Aug-2016 10:05:24      29-Sep-2016 10:05:24      29-Oct-2016 10:05:24
```

Convert `D` to the equivalent POSIX times.

```
PTZ = posixtime(DZ)
PTZ = 1x3
      1472479524      1475157924      1477749924
```

Create an unzoned copy of `DZ`. Convert `D` to the equivalent POSIX times. As `D` has no time zone, `posixtime` treats the times as UTC times.

```
D = DZ;
D.TimeZone = '';
PT = posixtime(D)
```

```
PT = 1×3
```

```
1472465124
```

```
1475143524
```

```
1477735524
```

Compare PTZ and PT. The differences are equal to the time zone offset between UTC and the America/New_York time zone in seconds.

```
PTZ - PT
```

```
ans = 1×3
```

```
14400
```

```
14400
```

```
14400
```

See Also

`datetime` | `juliandate` | `posixtime` | `timezones`

Related Examples

- “Represent Dates and Times in MATLAB” on page 7-2
- “Specify Time Zones” on page 7-6

Set Date and Time Display Format

In this section...

“Formats for Individual Date and Duration Arrays” on page 7-12

“datetime Display Format” on page 7-12

“duration Display Format” on page 7-13

“calendarDuration Display Format” on page 7-14

“Default datetime Format” on page 7-15

Formats for Individual Date and Duration Arrays

`datetime`, `duration`, and `calendarDuration` arrays have a `Format` property that controls the display of values in each array. When you create a `datetime` array, it uses the MATLAB global default `datetime` display format unless you explicitly provide a format. Use dot notation to access the `Format` property to view or change its value. For example, to set the display format for the `datetime` array, `t`, to the default format, type:

```
t.Format = 'default'
```

Changing the `Format` property does not change the values in the array, only their display. For example, the following can be representations of the same `datetime` value (the latter two do not display any time components):

```
Thursday, August 23, 2012 12:35:00
August 23, 2012
23-Aug-2012
```

The `Format` property of the `datetime`, `duration`, and `calendarDuration` data types accepts different formats as inputs.

datetime Display Format

You can set the `Format` property to one of these character vectors.

Value of Format	Description
'default'	Use the default display format.

Value of Format	Description
'defaultdate'	Use the default date display format that does not show time components.

To change the default formats, see “Default datetime Format” on page 7-15.

Alternatively, you can use the letters A-Z and a-z to specify a custom date format. You can include nonletter characters such as a hyphen, space, or colon to separate the fields. This table shows several common display formats and examples of the formatted output for the date, Saturday, April 19, 2014 at 9:41:06 PM in New York City.

Value of Format	Example
'yyyy-MM-dd'	2014-04-19
'dd/MM/yyyy'	19/04/2014
'dd.MM.yyyy'	19.04.2014
'yyyy 年 MM 月 dd 日'	2014 年 04 月 19 日
'MMMM d, yyyy'	April 19, 2014
'eeee, MMMM d, yyyy h:mm a'	Saturday, April 19, 2014 9:41 PM
'MMMM d, yyyy HH:mm:ss Z'	April 19, 2014 21:41:06 -0400
'yyyy-MM-dd' 'T' 'HH:mmXXX'	2014-04-19T21:41-04:00

For a complete list of valid symbolic identifiers, see the `Format` property for datetime arrays.

Note The letter identifiers that `datetime` accepts are different from those used by the `datestr`, `datenum`, and `datevec` functions.

duration Display Format

To display a duration as a single number that includes a fractional part (for example, 1.234 hours), specify one of these character vectors:

Value of Format	Description
'y'	Number of exact fixed-length years. A fixed-length year is equal to 365.2425 days.
'd'	Number of exact fixed-length days. A fixed-length day is equal to 24 hours.
'h'	Number of hours
'm'	Number of minutes
's'	Number of seconds

To specify the number of fractional digits displayed, use the `format` function.

To display a duration in the form of a digital timer, specify one of the following character vectors.

- `'dd:hh:mm:ss'`
- `'hh:mm:ss'`
- `'mm:ss'`
- `'hh:mm'`

You also can display up to nine fractional second digits by appending up to nine `S` characters. For example, `'hh:mm:ss.SSS'` displays the milliseconds of a duration value to 3 digits.

Changing the `Format` property does not change the values in the array, only their display.

calendarDuration Display Format

Specify the `Format` property of a `calendarDuration` array as a character vector that can include the characters `y`, `q`, `m`, `w`, `d`, and `t`, in this order. The format must include `m`, `d`, and `t`.

This table describes the date and time components that the characters represent.

Character	Unit	Required?
y	Years	no

Character	Unit	Required?
q	Quarters (multiples of 3 months)	no
m	Months	yes
w	Weeks	no
d	Days	yes
t	Time (hours, minutes, and seconds)	yes

To specify the number of digits displayed for fractional seconds, use the `format` function.

If the value of a date or time component is zero, it is not displayed.

Changing the `Format` property does not change the values in the array, only their display.

Default datetime Format

You can set default formats to control the display of `datetime` arrays created without an explicit display format. These formats also apply when you set the `Format` property of a `datetime` array to `'default'` or `'defaultdate'`. When you change the default setting, `datetime` arrays set to use the default formats are displayed automatically using the new setting.

Changes to the default formats persist across MATLAB sessions.

To specify a default format, type

```
datetime.setDefaultFormats('default',fmt)
```

where *fmt* is a character vector composed of the letters A-Z and a-z described for the `Format` property of `datetime` arrays, above. For example,

```
datetime.setDefaultFormats('default','yyyy-MM-dd hh:mm:ss')
```

sets the default `datetime` format to include a 4-digit year, 2-digit month number, 2-digit day number, and hour, minute, and second values.

In addition, you can specify a default format for datetimes created without time components. For example,

```
datetime.setDefaultFormats('defaultdate','yyyy-MM-dd')
```

sets the default date format to include a 4-digit year, 2-digit month number, and 2-digit day number.

To reset the both the default format and the default date-only formats to the factory defaults, type

```
datetime.setDefaultFormats('reset')
```

The factory default formats depend on your system locale.

You also can set the default formats in the **Preferences** dialog box. For more information, see “Set Command Window Preferences”.

See Also

`calendarDuration` | `datetime` | `duration` | `format`

Generate Sequence of Dates and Time

In this section...

“Sequence of Datetime or Duration Values Between Endpoints with Step Size” on page 7-17

“Add Duration or Calendar Duration to Create Sequence of Dates” on page 7-19

“Specify Length and Endpoints of Date or Duration Sequence” on page 7-21

“Sequence of Datetime Values Using Calendar Rules” on page 7-22

Sequence of Datetime or Duration Values Between Endpoints with Step Size

This example shows how to use the colon (:) operator to generate sequences of `datetime` or `duration` values in the same way that you create regularly spaced numeric vectors.

Use Default Step Size

Create a sequence of datetime values starting from November 1, 2013 and ending on November 5, 2013. The default step size is one calendar day.

```
t1 = datetime(2013,11,1,8,0,0);
t2 = datetime(2013,11,5,8,0,0);
t = t1:t2
```

```
t = 1x5 datetime array
Columns 1 through 3
```

```
    01-Nov-2013 08:00:00    02-Nov-2013 08:00:00    03-Nov-2013 08:00:00
```

```
Columns 4 through 5
```

```
    04-Nov-2013 08:00:00    05-Nov-2013 08:00:00
```

Specify Step Size

Specify a step size of 2 calendar days using the `caldays` function.

```
t = t1:caldays(2):t2
```

```
t = 1x3 datetime array
    01-Nov-2013 08:00:00    03-Nov-2013 08:00:00    05-Nov-2013 08:00:00
```

Specify a step size in units other than days. Create a sequence of datetime values spaced 18 hours apart.

```
t = t1:hours(18):t2
```

```
t = 1x6 datetime array
Columns 1 through 3
```

```
    01-Nov-2013 08:00:00    02-Nov-2013 02:00:00    02-Nov-2013 20:00:00
```

```
Columns 4 through 6
```

```
    03-Nov-2013 14:00:00    04-Nov-2013 08:00:00    05-Nov-2013 02:00:00
```

Use the `years`, `days`, `minutes`, and `seconds` functions to create datetime and duration sequences using other fixed-length date and time units. Create a sequence of duration values between 0 and 3 minutes, incremented by 30 seconds.

```
d = 0:seconds(30):minutes(3)
```

```
d = 1x7 duration array
```

```
    0 sec    30 sec    60 sec    90 sec    120 sec    150 sec    180 sec
```

Compare Fixed-Length Duration and Calendar Duration Step Sizes

Assign a time zone to `t1` and `t2`. In the `America/New_York` time zone, `t1` now occurs just before a daylight saving time change.

```
t1.TimeZone = 'America/New_York';
t2.TimeZone = 'America/New_York';
```

If you create the sequence using a step size of one calendar day, then the difference between successive `datetime` values is not always 24 hours.

```
t = t1:t2;
dt = diff(t)
```

```
dt = 1x4 duration array
```

```
    24:00:00    25:00:00    24:00:00    24:00:00
```

Create a sequence of datetime values spaced one fixed-length day apart,

```
t = t1:days(1):t2
```

```
t = 1x5 datetime array
Columns 1 through 3
```

```
    01-Nov-2013 08:00:00    02-Nov-2013 08:00:00    03-Nov-2013 07:00:00
```

```
Columns 4 through 5
```

```
    04-Nov-2013 07:00:00    05-Nov-2013 07:00:00
```

Verify that the difference between successive `datetime` values is 24 hours.

```
dt = diff(t)
```

```
dt = 1x4 duration array
```

```
    24:00:00    24:00:00    24:00:00    24:00:00
```

Integer Step Size

If you specify a step size in terms of an integer, it is interpreted as a number of 24-hour days.

```
t = t1:1:t2
```

```
t = 1x5 datetime array
Columns 1 through 3
```

```
    01-Nov-2013 08:00:00    02-Nov-2013 08:00:00    03-Nov-2013 07:00:00
```

```
Columns 4 through 5
```

```
    04-Nov-2013 07:00:00    05-Nov-2013 07:00:00
```

Add Duration or Calendar Duration to Create Sequence of Dates

This example shows how to add a duration or calendar duration to a datetime to create a sequence of datetime values.

Create a datetime scalar representing November 1, 2013 at 8:00 AM.

```
t1 = datetime(2013,11,1,8,0,0);
```

Add a sequence of fixed-length hours to the datetime.

```
t = t1 + hours(0:2)
```

```
t = 1x3 datetime array  
    01-Nov-2013 08:00:00    01-Nov-2013 09:00:00    01-Nov-2013 10:00:00
```

Add a sequence of calendar months to the datetime.

```
t = t1 + calmonths(1:5)
```

```
t = 1x5 datetime array  
Columns 1 through 3  
    01-Dec-2013 08:00:00    01-Jan-2014 08:00:00    01-Feb-2014 08:00:00
```

```
Columns 4 through 5
```

```
    01-Mar-2014 08:00:00    01-Apr-2014 08:00:00
```

Each datetime in `t` occurs on the first day of each month.

Verify that the dates in `t` are spaced 1 month apart.

```
dt = caldiff(t)
```

```
dt = 1x4 calendarDuration array  
    1mo    1mo    1mo    1mo
```

Determine the number of days between each date.

```
dt = caldiff(t, 'days')
```

```
dt = 1x4 calendarDuration array  
    31d    31d    28d    31d
```

Add a number of calendar months to the date, January 31, 2014, to create a sequence of dates that fall on the last day of each month.

```
t = datetime(2014,1,31) + calmonths(0:11)

t = 1x12 datetime array
Columns 1 through 5

    31-Jan-2014    28-Feb-2014    31-Mar-2014    30-Apr-2014    31-May-2014

Columns 6 through 10

    30-Jun-2014    31-Jul-2014    31-Aug-2014    30-Sep-2014    31-Oct-2014

Columns 11 through 12

    30-Nov-2014    31-Dec-2014
```

Specify Length and Endpoints of Date or Duration Sequence

This example shows how to use the `linspace` function to create equally spaced datetime or duration values between two specified endpoints.

Create a sequence of five equally spaced dates between April 14, 2014 and August 4, 2014. First, define the endpoints.

```
A = datetime(2014,04,14);
B = datetime(2014,08,04);
```

The third input to `linspace` specifies the number of linearly spaced points to generate between the endpoints.

```
C = linspace(A,B,5)

C = 1x5 datetime array
    14-Apr-2014    12-May-2014    09-Jun-2014    07-Jul-2014    04-Aug-2014
```

Create a sequence of six equally spaced durations between 1 and 5.5 hours.

```
A = duration(1,0,0);
B = duration(5,30,0);
C = linspace(A,B,6)
```

```
C = 1x6 duration array
    01:00:00    01:54:00    02:48:00    03:42:00    04:36:00    05:30:00
```

Sequence of Datetime Values Using Calendar Rules

This example shows how to use the `dateshift` function to generate sequences of dates and time where each instance obeys a rule relating to a calendar unit or a unit of time. For instance, each datetime must occur at the beginning a month, on a particular day of the week, or at the end of a minute. The resulting datetime values in the sequence are not necessarily equally spaced.

Dates on Specific Day of Week

Generate a sequence of dates consisting of the next three occurrences of Monday. First, define today's date.

```
t1 = datetime('today', 'Format', 'dd-MMM-yyyy eee')
```

```
t1 = datetime
    27-Aug-2018 Mon
```

The first input to `dateshift` is always the `datetime` array from which you want to generate a sequence. Specify `'dayofweek'` as the second input to indicate that the datetime values in the output sequence must fall on a specific day of the week. You can specify the day of the week either by number or by name. For example, you can specify Monday either as 2 or `'Monday'`.

```
t = dateshift(t1, 'dayofweek', 2, 1:3)
```

```
t = 1x3 datetime array
    27-Aug-2018 Mon    03-Sep-2018 Mon    10-Sep-2018 Mon
```

Dates at Start of Month

Generate a sequence of start-of-month dates beginning with April 1, 2014. Specify `'start'` as the second input to `dateshift` to indicate that all datetime values in the output sequence should fall at the start of a particular unit of time. The third input argument defines the unit of time, in this case, month. The last input to `dateshift` can be an array of integer values that specifies how `t1` should be shifted. In this case, 0

corresponds to the start of the current month, and 4 corresponds to the start of the fourth month from `t1`.

```
t1 = datetime(2014,04,01);
t = dateshift(t1,'start','month',0:4)

t = 1x5 datetime array
    01-Apr-2014    01-May-2014    01-Jun-2014    01-Jul-2014    01-Aug-2014
```

Dates at End of Month

Generate a sequence of end-of-month dates beginning with April 1, 2014.

```
t1 = datetime(2014,04,01);
t = dateshift(t1,'end','month',0:2)

t = 1x3 datetime array
    30-Apr-2014    31-May-2014    30-Jun-2014
```

Determine the number of days between each date.

```
dt = caldiff(t,'days')

dt = 1x2 calendarDuration array
    31d    30d
```

The dates are not equally spaced.

Other Units of Dates and Time

You can specify other units of time such as week, day, and hour.

```
t1 = datetime('now')

t1 = datetime
    27-Aug-2018 18:46:44

t = dateshift(t1,'start','hour',0:4)

t = 1x5 datetime array
Columns 1 through 3
```

```
27-Aug-2018 18:00:00 27-Aug-2018 19:00:00 27-Aug-2018 20:00:00
```

```
Columns 4 through 5
```

```
27-Aug-2018 21:00:00 27-Aug-2018 22:00:00
```

Previous Occurrences of Dates and Time

Generate a sequence of datetime values beginning with the previous hour. Negative integers in the last input to `dateshift` correspond to datetime values earlier than `t1`.

```
t = dateshift(t1, 'start', 'hour', -1:1)
```

```
t = 1x3 datetime array
```

```
27-Aug-2018 17:00:00 27-Aug-2018 18:00:00 27-Aug-2018 19:00:00
```

See Also

`dateshift` | `linspace`

Share Code and Data Across Locales

In this section...

“Write Locale-Independent Date and Time Code” on page 7-25

“Write Dates in Other Languages” on page 7-26

“Read Dates in Other Languages” on page 7-27

Write Locale-Independent Date and Time Code

Follow these best practices when sharing code that handles dates and time with MATLAB® users in other locales. These practices ensure that the same code produces the same output display and that output files containing dates and time are read correctly on systems in different countries or with different language settings.

Create language-independent datetime values. That is, create datetime values that use month numbers rather than month names, such as 01 instead of January. Avoid using day of week names.

For example, do this:

```
t = datetime('today','Format','yyyy-MM-dd')
```

```
t = datetime
    2018-08-27
```

instead of this:

```
t = datetime('today','Format','eeee, dd-MMM-yyyy')
```

```
t = datetime
    Monday, 27-Aug-2018
```

Display the hour using 24-hour clock notation rather than 12-hour clock notation. Use the 'HH' identifiers when specifying the display format for datetime values.

For example, do this:

```
t = datetime('now','Format','HH:mm')
```

```
t = datetime
    15:59
```

instead of this:

```
t = datetime('now','Format','hh:mm a')
```

```
t = datetime
    03:59 PM
```

When specifying the display format for time zone information, use the Z or X identifiers instead of the lowercase z to avoid the creation of time zone names that might not be recognized in other languages or regions.

Assign a time zone to `t`.

```
t.TimeZone = 'America/New_York';
```

Specify a language-independent display format that includes a time zone.

```
t.Format = 'dd-MM-yyyy Z'
```

```
t = datetime
    27-08-2018 -0400
```

If you share files but not code, you do not need to write locale-independent code while you work in MATLAB. However, when you write to a file, ensure that any text representing dates and times is language-independent. Then, other MATLAB users can read the files easily without having to specify a locale in which to interpret date and time data.

Write Dates in Other Languages

Specify an appropriate format for text representing dates and times when you use the `char` or `cellstr` functions. For example, convert two `datetime` values to a cell array of character vectors using `cellstr`. Specify the format and the locale to represent the day, month, and year of each `datetime` value as text.

```
t = [datetime('today');datetime('tomorrow')]
```

```
t = 2x1 datetime array
    27-Aug-2018
```

28-Aug-2018

```
S = cellstr(t, 'dd. MMMM yyyy', 'de_DE')
```

```
S = 2x1 cell array  
    {'27. August 2018'}  
    {'28. August 2018'}
```

S is a cell array of character vectors representing dates in German. You can export S to a text file to use with systems in the `de_DE` locale.

Read Dates in Other Languages

You can read text files containing dates and time in a language other than the language that MATLAB uses, which depends on your system locale. Use the `textscan` or `readtable` functions with the `DateLocale` name-value pair argument to specify the locale in which the function interprets the dates in the file. In addition, you might need to specify the character encoding of a file that contains characters that are not recognized by your computer's default encoding.

- When reading text files using the `textscan` function, specify the file encoding when opening the file with `fopen`. The encoding is the fourth input argument to `fopen`.
- When reading text files using the `readtable` function, use the `FileEncoding` name-value pair argument to specify the character encoding associated with the file.

See Also

`cellstr` | `char` | `datetime` | `readtable` | `textscan`

Extract or Assign Date and Time Components of Datetime Array

This example shows two ways to extract date and time components from existing datetime arrays: accessing the array properties or calling a function. Then, the example shows how to modify the date and time components by modifying the array properties.

Access Properties to Retrieve Date and Time Component

Create a datetime array.

```
t = datetime('now') + calyears(0:2) + calmonths(0:2) + hours(20:20:60)
t = 1x3 datetime array
    28-Aug-2018 14:47:18    29-Sep-2019 10:47:18    30-Oct-2020 06:47:18
```

Get the year values of each datetime in the array. Use dot notation to access the Year property of `t`.

```
t_years = t.Year
t_years = 1x3
    2018    2019    2020
```

The output, `t_years`, is a numeric array.

Get the month values of each datetime in `t` by accessing the Month property.

```
t_months = t.Month
t_months = 1x3
    8     9    10
```

You can retrieve the day, hour, minute, and second components of each datetime in `t` by accessing the Hour, Minute, and Second properties, respectively.

Use Functions to Retrieve Date and Time Component

Use the `month` function to get the month number for each datetime in `t`. Using functions is an alternate way to retrieve specific date or time components of `t`.

```
m = month(t)
m = 1x3
      8      9     10
```

Use the `month` function rather than the `Month` property to get the full month names of each datetime in `t`.

```
m = month(t, 'name')
m = 1x3 cell array
      {'August'}      {'September'}      {'October'}
```

You can retrieve the year, quarter, week, day, hour, minute, and second components of each datetime in `t` using the `year`, `quarter`, `week`, `hour`, `minute`, and `second` functions, respectively.

Get the week of year numbers for each datetime in `t`.

```
w = week(t)
w = 1x3
      35     40     44
```

Get Multiple Date and Time Components

Use the `ymd` function to get the year, month, and day values of `t` as three separate numeric arrays.

```
[y,m,d] = ymd(t)
y = 1x3
      2018      2019      2020
```

```
m = 1×3
      8      9      10
```

```
d = 1×3
      28      29      30
```

Use the `hms` function to get the hour, minute, and second values of `t` as three separate numeric arrays.

```
[h,m,s] = hms(t)
```

```
h = 1×3
      14      10      6
```

```
m = 1×3
      47      47      47
```

```
s = 1×3
      18.1660    18.1660    18.1660
```

Modify Date and Time Components

Assign new values to components in an existing `datetime` array by modifying the properties of the array. Use dot notation to access a specific property.

Change the year number of all `datetime` values in `t` to 2014. Use dot notation to modify the `Year` property.

```
t.Year = 2014
```

```
t = 1×3 datetime array
```

```
28-Aug-2014 14:47:18    29-Sep-2014 10:47:18    30-Oct-2014 06:47:18
```

Change the months of the three `datetime` values in `t` to January, February, and March, respectively. You must specify the new value as a numeric array.

```
t.Month = [1,2,3]
```

```
t = 1x3 datetime array
```

```
28-Jan-2014 14:47:18 01-Mar-2014 10:47:18 30-Mar-2014 06:47:18
```

Set the time zone of `t` by assigning a value to the `TimeZone` property.

```
t.TimeZone = 'Europe/Berlin';
```

Change the display format of `t` to display only the date, and not the time information.

```
t.Format = 'dd-MMM-yyyy'
```

```
t = 1x3 datetime array
```

```
28-Jan-2014 01-Mar-2014 30-Mar-2014
```

If you assign values to a datetime component that are outside the conventional range, MATLAB® normalizes the components. The conventional range for day of month numbers is from 1 to 31. Assign day values that exceed this range.

```
t.Day = [-1 1 32]
```

```
t = 1x3 datetime array
```

```
30-Dec-2013 01-Mar-2014 01-Apr-2014
```

The month and year numbers adjust so that all values remain within the conventional range for each date component. In this case, January -1, 2014 converts to December 30, 2013.

See Also

`datetime` | `hms` | `week` | `ymd`

Combine Date and Time from Separate Variables

This example shows how to read date and time data from a text file. Then, it shows how to combine date and time information stored in separate variables into a single datetime variable.

Create a space-delimited text file named `schedule.txt` that contains the following (to create the file, use any text editor, and copy and paste):

```
Date Name Time
10.03.2015 Joe 14:31
10.03.2015 Bob 15:33
11.03.2015 Bob 11:29
12.03.2015 Kim 12:09
12.03.2015 Joe 13:05
```

Read the file using the `readtable` function. Use the `%D` conversion specifier to read the first and third columns of data as datetime values.

```
T = readtable('schedule.txt','Format','%{dd.MM.uuuu}D %s %{HH:mm}D','Delimiter',' ')
```

```
T =
      Date      Name      Time
-----
10.03.2015    'Joe'    14:31
10.03.2015    'Bob'    15:33
11.03.2015    'Bob'    11:29
12.03.2015    'Kim'    12:09
12.03.2015    'Joe'    13:05
```

`readtable` returns a table containing three variables.

Change the display format for the `T.Date` and `T.Time` variables to view both date and time information. Since the data in the first column of the file ("Date") have no time information, the time of the resulting datetime values in `T.Date` default to midnight. Since the data in the third column of the file ("Time") have no associated date, the date of the datetime values in `T.Time` defaults to the current date.

```
T.Date.Format = 'dd.MM.uuuu HH:mm';
T.Time.Format = 'dd.MM.uuuu HH:mm';
T
```

```
T =
      Date      Name      Time
```

10.03.2015 00:00	'Joe'	12.12.2014 14:31
10.03.2015 00:00	'Bob'	12.12.2014 15:33
11.03.2015 00:00	'Bob'	12.12.2014 11:29
12.03.2015 00:00	'Kim'	12.12.2014 12:09
12.03.2015 00:00	'Joe'	12.12.2014 13:05

Combine the date and time information from two different table variables by adding `T.Date` and the time values in `T.Time`. Extract the time information from `T.Time` using the `timeofday` function.

```
myDatetime = T.Date + timeofday(T.Time)
```

```
myDatetime =  
10.03.2015 14:31  
10.03.2015 15:33  
11.03.2015 11:29  
12.03.2015 12:09  
12.03.2015 13:05
```

See Also

`readtable` | `timeofday`

Date and Time Arithmetic

This example shows how to add and subtract date and time values to calculate future and past dates and elapsed durations in exact units or calendar units. You can add, subtract, multiply, and divide date and time arrays in the same way that you use these operators with other MATLAB® data types. However, there is some behavior that is specific to dates and time.

Add and Subtract Durations to Datetime Array

Create a datetime scalar. By default, datetime arrays are not associated with a time zone.

```
t1 = datetime('now')
```

```
t1 = datetime
    27-Aug-2018 17:58:26
```

Find future points in time by adding a sequence of hours.

```
t2 = t1 + hours(1:3)
```

```
t2 = 1x3 datetime array
    27-Aug-2018 18:58:26    27-Aug-2018 19:58:26    27-Aug-2018 20:58:26
```

Verify that the difference between each pair of datetime values in `t2` is 1 hour.

```
dt = diff(t2)
```

```
dt = 1x2 duration array
    01:00:00    01:00:00
```

`diff` returns durations in terms of exact numbers of hours, minutes, and seconds.

Subtract a sequence of minutes from a datetime to find past points in time.

```
t2 = t1 - minutes(20:10:40)
```

```
t2 = 1x3 datetime array
    27-Aug-2018 17:38:26    27-Aug-2018 17:28:26    27-Aug-2018 17:18:26
```

Add a numeric array to a `datetime` array. MATLAB® treats each value in the numeric array as a number of exact, 24-hour days.

```
t2 = t1 + [1:3]
```

```
t2 = 1x3 datetime array
    28-Aug-2018 17:58:26    29-Aug-2018 17:58:26    30-Aug-2018 17:58:26
```

Add to Datetime with Time Zone

If you work with datetime values in different time zones, or if you want to account for daylight saving time changes, work with datetime arrays that are associated with time zones. Create a `datetime` scalar representing March 8, 2014 in New York.

```
t1 = datetime(2014,3,8,0,0,0, 'TimeZone', 'America/New_York')
```

```
t1 = datetime
    08-Mar-2014 00:00:00
```

Find future points in time by adding a sequence of fixed-length (24-hour) days.

```
t2 = t1 + days(0:2)
```

```
t2 = 1x3 datetime array
    08-Mar-2014 00:00:00    09-Mar-2014 00:00:00    10-Mar-2014 01:00:00
```

Because a daylight saving time shift occurred on March 9, 2014, the third datetime in `t2` does not occur at midnight.

Verify that the difference between each pair of datetime values in `t2` is 24 hours.

```
dt = diff(t2)
```

```
dt = 1x2 duration array
    24:00:00    24:00:00
```

You can add fixed-length durations in other units such as years, hours, minutes, and seconds by adding the outputs of the `years`, `hours`, `minutes`, and `seconds` functions, respectively.

To account for daylight saving time changes, you should work with calendar durations instead of durations. Calendar durations account for daylight saving time shifts when they are added to or subtracted from datetime values.

Add a number of calendar days to `t1`.

```
t3 = t1 + caldays(0:2)
```

```
t3 = 1x3 datetime array
    08-Mar-2014 00:00:00    09-Mar-2014 00:00:00    10-Mar-2014 00:00:00
```

View that the difference between each pair of datetime values in `t3` is not always 24 hours due to the daylight saving time shift that occurred on March 9.

```
dt = diff(t3)
```

```
dt = 1x2 duration array
    24:00:00    23:00:00
```

Add Calendar Durations to Datetime Array

Add a number of calendar months to January 31, 2014.

```
t1 = datetime(2014,1,31)
```

```
t1 = datetime
    31-Jan-2014
```

```
t2 = t1 + calmonths(1:4)
```

```
t2 = 1x4 datetime array
    28-Feb-2014    31-Mar-2014    30-Apr-2014    31-May-2014
```

Each datetime in `t2` occurs on the last day of each month.

Calculate the difference between each pair of datetime values in `t2` in terms of a number of calendar days using the `caldiff` function.

```
dt = caldiff(t2, 'days')
```

```
dt = 1x3 calendarDuration array
    31d    30d    31d
```

The number of days between successive pairs of datetime values in `dt` is not always the same because different months consist of a different number of days.

Add a number of calendar years to January 31, 2014.

```
t2 = t1 + calyears(0:4)
```

```
t2 = 1x5 datetime array
    31-Jan-2014    31-Jan-2015    31-Jan-2016    31-Jan-2017    31-Jan-2018
```

Calculate the difference between each pair of datetime values in `t2` in terms of a number of calendar days using the `caldiff` function.

```
dt = caldiff(t2, 'days')
```

```
dt = 1x4 calendarDuration array
    365d    365d    366d    365d
```

The number of days between successive pairs of datetime values in `dt` is not always the same because 2016 is a leap year and has 366 days.

You can use the `calquarters`, `calweeks`, and `caldays` functions to create arrays of calendar quarters, calendar weeks, or calendar days that you add to or subtract from datetime arrays.

Adding calendar durations is not commutative. When you add more than one `calendarDuration` array to a datetime, MATLAB® adds them in the order in which they appear in the command.

Add 3 calendar months followed by 30 calendar days to January 31, 2014.

```
t2 = datetime(2014,1,31) + calmonths(3) + caldays(30)
```

```
t2 = datetime
    30-May-2014
```

First add 30 calendar days to the same date, and then add 3 calendar months. The result is not the same because when you add a calendar duration to a datetime, the number of days added depends on the original date.

```
t2 = datetime(2014,1,31) + caldays(30) + calmonths(3)
```

```
t2 = datetime  
02-Jun-2014
```

Calendar Duration Arithmetic

Create two calendar durations and then find their sum.

```
d1 = calyears(1) + calmonths(2) + caldays(20)
```

```
d1 = calendarDuration  
1y 2mo 20d
```

```
d2 = calmonths(11) + caldays(23)
```

```
d2 = calendarDuration  
11mo 23d
```

```
d = d1 + d2
```

```
d = calendarDuration  
2y 1mo 43d
```

When you sum two or more calendar durations, a number of months greater than 12 roll over to a number of years. However, a large number of days does not roll over to a number of months, because different months consist of different numbers of days.

Increase `d` by multiplying it by a factor of 2. Calendar duration values must be integers, so you can multiply them only by integer values.

```
2*d
```

```
ans = calendarDuration  
4y 2mo 86d
```

Calculate Elapsed Time in Exact Units

Subtract one `datetime` array from another to calculate elapsed time in terms of an exact number of hours, minutes, and seconds.

Find the exact length of time between a sequence of datetime values and the start of the previous day.

```
t2 = datetime('now') + caldays(1:3)
```

```
t2 = 1x3 datetime array
    28-Aug-2018 17:58:27    29-Aug-2018 17:58:27    30-Aug-2018 17:58:27
```

```
t1 = datetime('yesterday')
```

```
t1 = datetime
    26-Aug-2018
```

```
dt = t2 - t1
```

```
dt = 1x3 duration array
    65:58:27    89:58:27    113:58:27
```

```
whos dt
```

Name	Size	Bytes	Class	Attributes
dt	1x3	40	duration	

`dt` contains durations in the format, hours:minutes:seconds.

View the elapsed durations in units of days by changing the `Format` property of `dt`.

```
dt.Format = 'd'
```

```
dt = 1x3 duration array
    2.7489 days    3.7489 days    4.7489 days
```

Scale the duration values by multiplying `dt` by a factor of 1.2. Because durations have an exact length, you can multiply and divide them by fractional values.

```
dt2 = 1.2*dt
```

```
dt2 = 1x3 duration array
      3.2987 days   4.4987 days   5.6987 days
```

Calculate Elapsed Time in Calendar Units

Use the `between` function to find the number of calendar years, months, and days elapsed between two dates.

```
t1 = datetime('today')
```

```
t1 = datetime
      27-Aug-2018
```

```
t2 = t1 + calmonths(0:2) + caldays(4)
```

```
t2 = 1x3 datetime array
      31-Aug-2018   01-Oct-2018   31-Oct-2018
```

```
dt = between(t1,t2)
```

```
dt = 1x3 calendarDuration array
      4d   1mo 4d   2mo 4d
```

See Also

`between` | `caldiff` | `diff`

Compare Dates and Time

This example shows how to compare `datetime` and `duration` arrays. You can perform an element-by-element comparison of values in two `datetime` arrays or two `duration` arrays using relational operators, such as `>` and `<`.

Compare Datetime Arrays

Compare two `datetime` arrays. The arrays must be the same size or one can be a scalar.

```
A = datetime(2013,07,26) + calyears(0:2:6)
```

```
A = 1x4 datetime array
    26-Jul-2013    26-Jul-2015    26-Jul-2017    26-Jul-2019
```

```
B = datetime(2014,06,01)
```

```
B = datetime
    01-Jun-2014
```

```
A < B
```

```
ans = 1x4 logical array
     1     0     0     0
```

The `<` operator returns logical 1 (true) where a datetime in A occurs before a datetime in B.

Compare a `datetime` array to text representing a date.

```
A >= 'September 26, 2014'
```

```
ans = 1x4 logical array
     0     1     1     1
```

Comparisons of `datetime` arrays account for the time zone information of each array.

Compare September 1, 2014 at 4:00 p.m. in Los Angeles with 5:00 p.m. on the same day in New York.

```
A = datetime(2014,09,01,16,0,0,'TimeZone','America/Los_Angeles',...
            'Format','dd-MMM-yyyy HH:mm:ss Z')
```

```
A = datetime
    01-Sep-2014 16:00:00 -0700
```

```
B = datetime(2014,09,01,17,0,0,'TimeZone','America/New_York',...
            'Format','dd-MMM-yyyy HH:mm:ss Z')
```

```
B = datetime
    01-Sep-2014 17:00:00 -0400
```

```
A < B
```

```
ans = logical
    0
```

4:00 p.m. in Los Angeles occurs after 5:00 p.m. on the same day in New York.

Compare Durations

Compare two duration arrays.

```
A = duration([2,30,30;3,15,0])
```

```
A = 2x1 duration array
    02:30:30
    03:15:00
```

```
B = duration([2,40,0;2,50,0])
```

```
B = 2x1 duration array
    02:40:00
    02:50:00
```

```
A >= B
```

```
ans = 2x1 logical array
    0
    1
```

Compare a duration array to a numeric array. Elements in the numeric array are treated as a number of fixed-length (24-hour) days.

```
A < [1; 1/24]
```

```
ans = 2x1 logical array
```

```
 1  
 0
```

Determine if Dates and Time Are Contained Within an Interval

Use the `isbetween` function to determine whether values in a `datetime` array lie within a closed interval.

Define endpoints of an interval.

```
tlower = datetime(2014,08,01)
```

```
tlower = datetime  
 01-Aug-2014
```

```
tupper = datetime(2014,09,01)
```

```
tupper = datetime  
 01-Sep-2014
```

Create a `datetime` array and determine whether the values lie within the interval bounded by `t1` and `t2`.

```
A = datetime(2014,08,21) + calweeks(0:2)
```

```
A = 1x3 datetime array  
 21-Aug-2014  28-Aug-2014  04-Sep-2014
```

```
tf = isbetween(A,tlower,tupper)
```

```
tf = 1x3 logical array
```

1 1 0

See Also

`isbetween`

More About

- “Array Comparison with Relational Operators” on page 2-25

Plot Dates and Durations

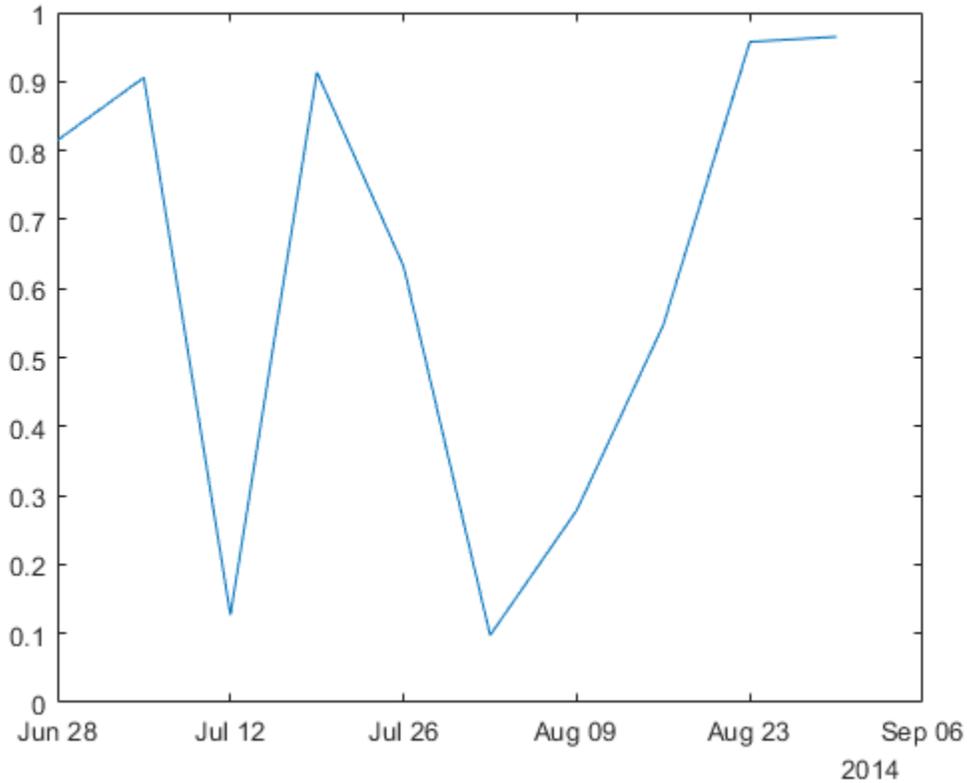
You can create plots of datetime and duration values with a variety of graphics functions. You also can customize the axes, such as changing the format of the tick labels or changing the axis limits.

Line Plot with Dates

Create a line plot with datetime values on the x-axis. Then, change the format of the tick labels and the x-axis limits.

Create `t` as a sequence of dates and create `y` as random data. Plot the vectors using the `plot` function.

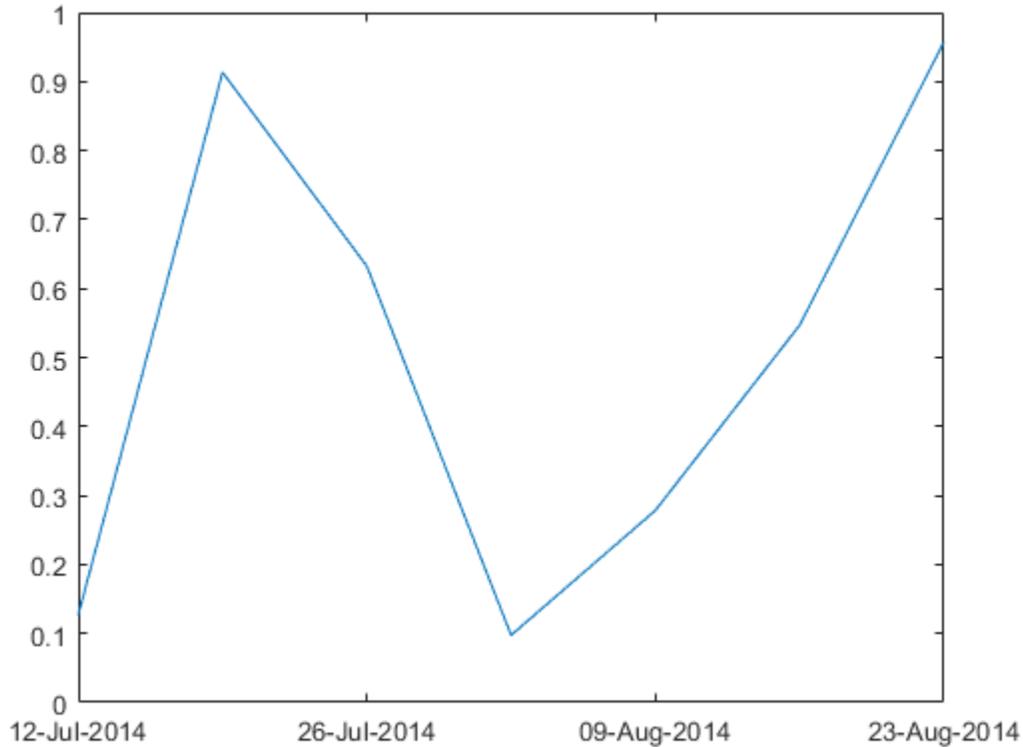
```
t = datetime(2014,6,28) + calweeks(0:9);  
y = rand(1,10);  
plot(t,y);
```



By default, `plot` chooses tick mark locations based on the range of data. When you zoom in and out of a plot, the tick labels automatically adjust to the new axis limits.

Change the x-axis limits. Also, change the format for the tick labels along the x-axis. For a list of formatting options, see the `xtickformat` function.

```
xlim(datetime(2014,[7 8],[12 23]))  
xtickformat('dd-MMM-yyyy')
```

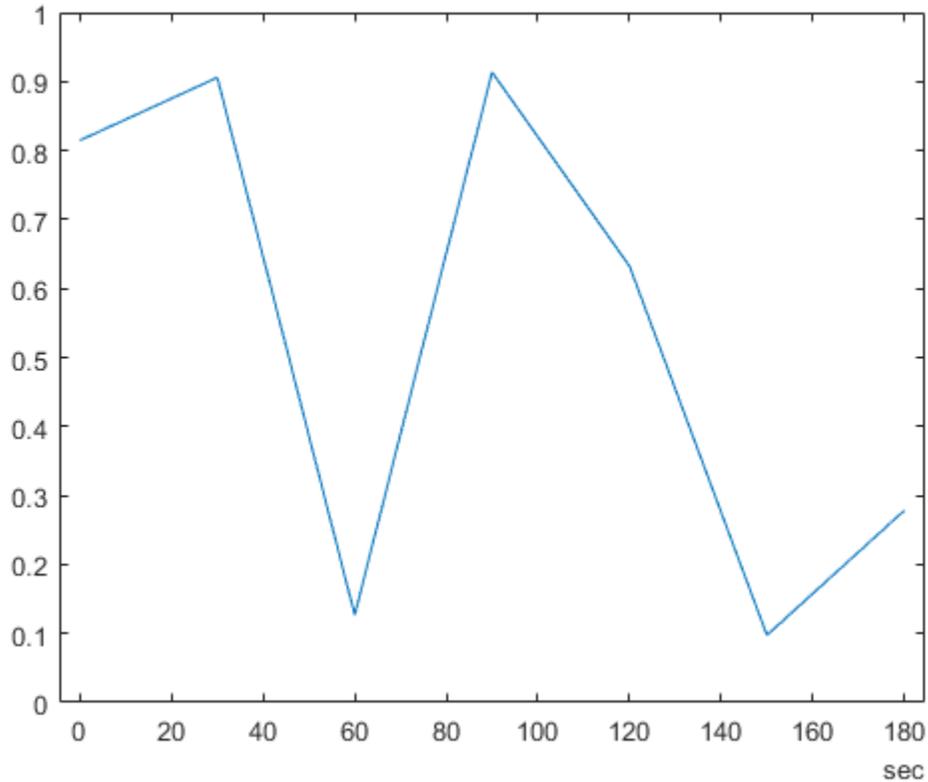


Line Plot with Durations

Create a line plot with duration values on the x-axis. Then, change the format of the tick labels and the x-axis limits.

Create `t` as seven linearly spaced duration values between 0 and 3 minutes. Create `y` as a vector of random data. Plot the data.

```
t = 0:seconds(30):minutes(3);  
y = rand(1,7);  
plot(t,y);
```



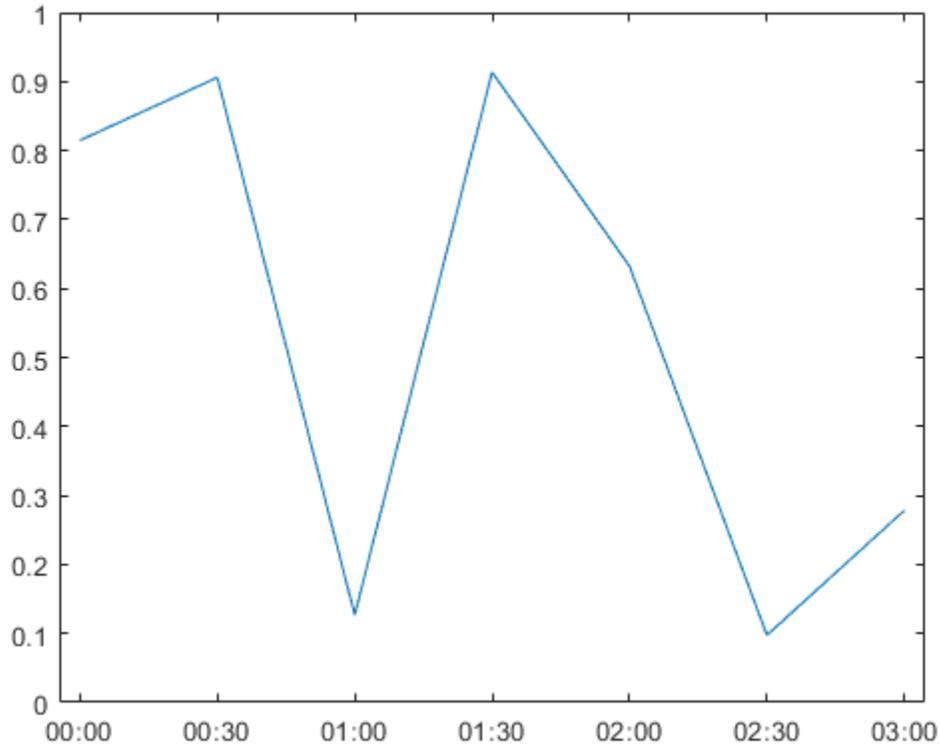
View the x-axis limits. Since the duration tick labels are in terms of a single unit (minutes), the limits are stored in terms of that unit.

```
x1 = xlim
```

```
x1 = 1x2 duration array  
    -4.5 sec    184.5 sec
```

Change the format for the duration tick labels to display in the form of a digital timer that includes more than one unit. For a list of formatting options, see the `xtickformat` function.

```
xtickformat('mm:ss')
```



View the x-axis limits again. Since the duration tick labels are now in terms of multiple units, the limits are stored in units of 24-hour days.

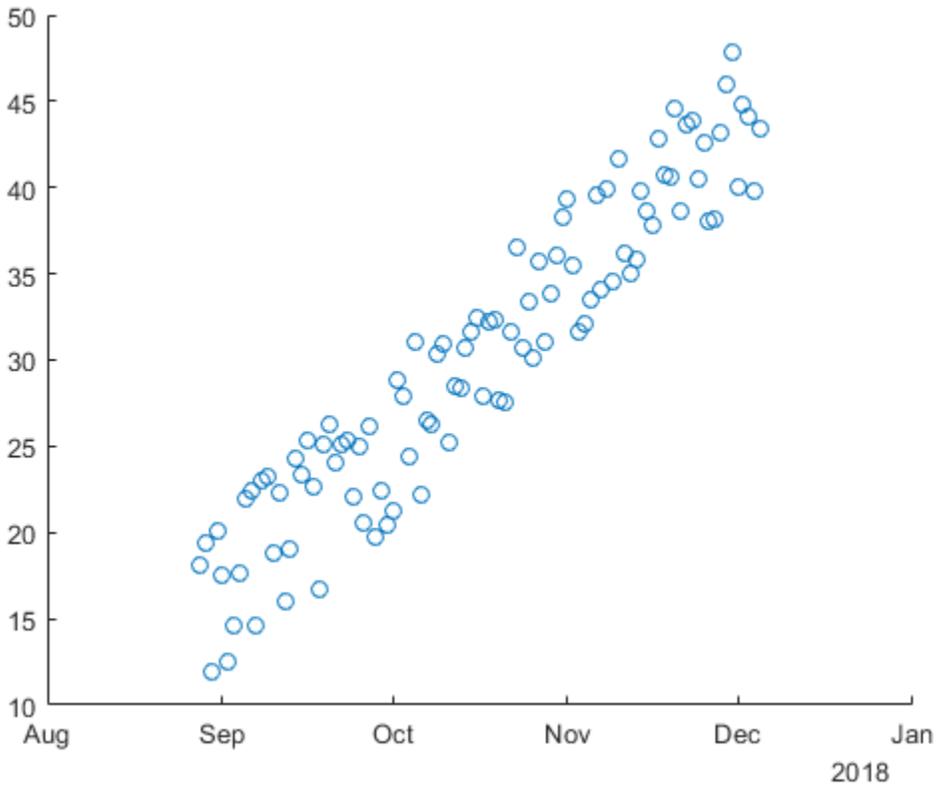
```
x1 = xlim
```

```
x1 = 1x2 duration array  
-00:04    03:04
```

Scatter Plot with Dates and Durations

Create a scatter plot with datetime or duration inputs using the `scatter` or `scatter3` functions. For example, create a scatter plot with dates along the x-axis.

```
t = datetime('today') + caldays(1:100);
y = linspace(10,40,100) + 10*rand(1,100);
scatter(t,y)
```



Plots that Support Dates and Durations

You can create other types of plots with datetime or duration values. These graphics functions support datetime and duration values.

bar	barh
plot	plot3

<code>semilogx</code> (x values must be numeric)	<code>semilogy</code> (y values must be numeric)
<code>stem</code>	<code>stairs</code>
<code>scatter</code>	<code>scatter3</code>
<code>area</code>	<code>mesh</code>
<code>surf</code>	<code>surface</code>
<code>fill</code>	<code>fill3</code>
<code>line</code>	<code>text</code>
<code>histogram</code>	

See Also

`datetime` | `plot` | `xtickformat`

Core Functions Supporting Date and Time Arrays

Many functions in MATLAB operate on date and time arrays in much the same way that they operate on other arrays.

This table lists notable MATLAB functions that operate on `datetime`, `duration`, and `calendarDuration` arrays in addition to other arrays.

size	isequal	intersect	plus	plot
length	isequaln	ismember	minus	plot3
ndims		setdiff	uminus	scatter
numel	eq	setxor	times	scatter3
isrow	ne	unique	rdivide	bar
iscolumn	lt	union	ldivide	barh
	le		mtimes	histogram
cat	ge	abs	mrdivide	
horzcat	gt	floor	mldivide	stem
vertcat		ceil	diff	stairs
	sort	round	sum	area
permute	sortrows	min	char	mesh
reshape	issorted	max	string	surf
transpose		mean	cellstr	surface
ctranspose		median		semilogx
		mode		semilogy
linspace				fill
				fill3
				line
				text

Convert Between Datetime Arrays, Numbers, and Text

In this section...

“Overview” on page 7-53

“Convert Between Datetime and Character Vectors” on page 7-54

“Convert Between Datetime and String Arrays” on page 7-55

“Convert Between Datetime and Date Vectors” on page 7-56

“Convert Serial Date Numbers to Datetime” on page 7-57

“Convert Datetime Arrays to Numeric Values” on page 7-57

Overview

`datetime` is the best data type for representing points in time. `datetime` values have flexible display formats and up to nanosecond precision, and can account for time zones, daylight saving time, and leap seconds. However, if you work with code authored in MATLAB R2014a or earlier, or if you share code with others who use such a version, you might need to work with dates and time stored in one of these three formats:

- Date String on page 7-54 — A character vector.

Example: Thursday, August 23, 2012 9:45:44.946 AM

- Date Vector on page 7-56 — A 1-by-6 numeric vector containing the year, month, day, hour, minute, and second.

Example: [2012 8 23 9 45 44.946]

- Serial Date Number on page 7-57 — A single number equal to the number of days since January 0, 0000 in the proleptic ISO calendar. Serial date numbers are useful as inputs to some MATLAB functions that do not accept the `datetime` or `duration` data types.

Example: 7.3510e+005

Date strings, vectors, and numbers can be stored as arrays of values. Store multiple date strings in a cell array of character vectors, multiple date vectors in an `m`-by-6 matrix, and multiple serial date numbers in a matrix.

You can convert any of these formats to a `datetime` array using the `datetime` function. If your existing MATLAB code expects a serial date number or date vector, use the

`datenum` or `datevec` functions, respectively, to convert a `datetime` array to the expected data format. To convert a `datetime` array to character vectors, use the `char` or `cellstr` functions.

Starting in R2016b, you also can convert a `datetime` array to a string array with the `string` function.

Convert Between Datetime and Character Vectors

A date string can be a character vector composed of fields related to a specific date and/or time. There are several ways to represent dates and times in text format. For example, all of the following are character vectors representing 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'
```

A date string includes characters that separate the fields, such as the hyphen, space, and colon used here:

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

Convert one or more date strings to a `datetime` array using the `datetime` function. For best performance, specify the format of the input date strings as an input to `datetime`.

Note The specifiers that `datetime` uses to describe date and time formats differ from the specifiers that the `datestr`, `datevec`, and `datenum` functions accept.

```
t = datetime(d,'InputFormat','dd-MMM-yyyy HH:mm:ss')
```

```
t =
```

```
datetime
```

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

Although the date string, `d`, and the `datetime` scalar, `t`, look similar, they are not equal. View the size and data type of each variable.

```
whos d t
```

Name	Size	Bytes	Class	Attributes
d	1x20	40	char	
t	1x1	17	datetime	

Convert a `datetime` array to a character vector using `char` or `cellstr`. For example, convert the current date and time to a timestamp to append to a file name.

```
t = datetime('now','Format','yyyy-MM-dd'T'HHmmss')
t =
    datetime
    2017-01-03T151105
S = char(t);
filename = ['myTest_',S]
filename =
    'myTest_2017-01-03T151105'
```

Convert Between Datetime and String Arrays

Starting in R2016b, you can use the `string` function to create a string array. If a string array contains date strings, then you can convert the string array to a `datetime` array with the `datetime` function. Similarly, you can convert a `datetime` array to a string array with the `string` function.

Convert a string array. MATLAB displays strings in double quotes. For best performance, specify the format of the input date strings as an input to `datetime`.

```
str = string({'24-Oct-2016 11:58:17';
            '19-Nov-2016 09:36:29';
            '12-Dec-2016 10:09:06'})
str =
    3x1 string array
    "24-Oct-2016 11:58:17"
    "19-Nov-2016 09:36:29"
    "12-Dec-2016 10:09:06"
```

```
t = datetime(str, 'InputFormat', 'dd-MMM-yyyy HH:mm:ss')  
t =  
  
3×1 datetime array  
  
24-Oct-2016 11:58:17  
19-Nov-2016 09:36:29  
12-Dec-2016 10:09:06
```

Convert a `datetime` value to a string.

```
t = datetime('25-Dec-2016 06:12:34');  
str = string(t)  
  
str =  
  
"25-Dec-2016 06:12:34"
```

Convert Between Datetime and Date Vectors

A date vector is a 1-by-6 vector of double-precision numbers. Elements of a date vector are integer-valued, except for the seconds element, which can be fractional. Time values are expressed in 24-hour notation. There is no AM or PM setting.

A date vector is arranged in the following order:

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

The following date vector represents 10:45:07 AM on October 24, 2012:

```
[2012 10 24 10 45 07]
```

Convert one or more date vectors to a `datetime` array using the `datetime` function:

```
t = datetime([2012 10 24 10 45 07])  
t =  
  
datetime  
  
24-Oct-2012 10:45:07
```

Instead of using `datevec` to extract components of `datetime` values, use functions such as `year`, `month`, and `day` instead:

```
y = year(t)
```

```
y =
```

```
    2012
```

Alternatively, access the corresponding property, such as `t.Year` for year values:

```
y = t.Year
```

```
y =
```

```
    2012
```

Convert Serial Date Numbers to Datetime

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.

Serial time can represent fractions of days beginning at midnight; for example, 6 p.m. equals 0.75 serial days. So the character vector `'31-Oct-2003, 6:00 PM'` in MATLAB is date number 731885.75.

Convert one or more serial date numbers to a `datetime` array using the `datetime` function. Specify the type of date number that is being converted:

```
t = datetime(731885.75, 'ConvertFrom', 'datenum')
```

```
t =
```

```
datetime
```

```
    31-Oct-2003 18:00:00
```

Convert Datetime Arrays to Numeric Values

Some MATLAB functions accept numeric data types but not datetime values as inputs. To apply these functions to your date and time data, convert datetime values to meaningful numeric values. Then, call the function. For example, the `log` function accepts `double` inputs, but not `datetime` inputs. Suppose that you have a `datetime` array of dates spanning the course of a research study or experiment.

```
t = datetime(2014,6,18) + calmonths(1:4)
```

```
t =  
  
1×4 datetime array  
  
18-Jul-2014 18-Aug-2014 18-Sep-2014 18-Oct-2014
```

Subtract the origin value. For example, the origin value might be the starting day of an experiment.

```
dt = t - datetime(2014,7,1)  
  
dt =  
  
1×4 duration array  
  
408:00:00 1152:00:00 1896:00:00 2616:00:00
```

`dt` is a `duration` array. Convert `dt` to a `double` array of values in units of years, days, hours, minutes, or seconds using the `years`, `days`, `hours`, `minutes`, or `seconds` function, respectively.

```
x = hours(dt)  
  
x =  
  
408 1152 1896 2616
```

Pass the `double` array as the input to the `log` function.

```
y = log(x)  
  
y =  
  
6.0113 7.0493 7.5475 7.8694
```

See Also

`cellstr` | `char` | `datenum` | `datetime` | `datevec`

More About

- “Represent Dates and Times in MATLAB” on page 7-2

Carryover in Date Vectors and Strings

If an element falls outside the conventional range, MATLAB adjusts both that date vector element and the previous element. For example, if the minutes element is 70, MATLAB adjusts the hours element by 1 and sets the minutes element to 10. If the minutes element is -15, then MATLAB decreases the hours element by 1 and sets the minutes element to 45. Month values are an exception. MATLAB sets month values less than 1 to 1.

In the following example, the month element has a value of 22. MATLAB increments the year value to 2010 and sets the month to October.

```
datestr([2009 22 03 00 00 00])
```

```
ans =  
03-Oct-2010
```

The carrying forward of values also applies to time and day values in text representing dates and times. For example, October 3, 2010 and September 33, 2010 are interpreted to be the same date, and correspond to the same serial date number.

```
datenum('03-Oct-2010')
```

```
ans =  
734414
```

```
datenum('33-Sep-2010')
```

```
ans =  
734414
```

The following 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
```

Converting Date Vector Returns Unexpected Output

Because a date vector is a 1-by-6 vector of numbers, `datestr` might interpret your input date vectors as vectors of serial date numbers, or vice versa, and return unexpected output.

Consider a date vector that includes the year 3000. This year is outside the range of years that `datestr` interprets as elements of date vectors. Therefore, the input is interpreted as a 1-by-6 vector of serial 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
```

Here `datestr` interprets 3000 as a serial date number, and converts it to the date string '18-Mar-0008'. Also, `datestr` converts the next five elements to date strings.

When converting such a date vector to a character vector, first convert it to a serial date number using `datenum`. Then, convert the date number to a character vector using `datestr`:

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

```
ds =
```

```
05-Nov-3000 10:32:56
```

When converting dates to character vectors, `datestr` interprets input as either date vectors or serial date numbers using a heuristic rule. Consider an m -by-6 matrix. `datestr` interprets the matrix as m date vectors when:

- The first five columns contain integers.
- The absolute value of the sum of each row is in the range 1500–2500.

If either condition is false, for any row, then `datestr` interprets the `m-by-6` matrix as `m-by-6` serial date numbers.

Usually, dates with years in the range 1700-2300 are interpreted as date vectors. However, `datestr` might interpret rows with month, day, hour, minute, or second values outside their normal ranges as serial date numbers. For example, `datestr` correctly interprets the following date vector for the year 2014:

```
datestr([2014 06 21 10 51 00])
ans =
21-Jun-2014 10:51:00
```

But given a day value outside the typical range (1-31), `datestr` returns a date for each element of the vector:

```
datestr([2014 06 2110 10 51 00])
ans =
06-Jul-0005
06-Jan-0000
10-Oct-0005
10-Jan-0000
20-Feb-0000
00-Jan-0000
```

When you have a matrix of date vectors that `datestr` might interpret incorrectly as serial date numbers, first convert the matrix to serial date numbers using `datenum`. Then, use `datestr` to convert the date numbers.

When you have a matrix of serial date numbers that `datestr` might interpret as date vectors, first convert the matrix to a column vector. Then, use `datestr` to convert the column vector.

Categorical Arrays

- “Create Categorical Arrays” on page 8-2
- “Convert Text in Table Variables to Categorical” on page 8-7
- “Plot Categorical Data” on page 8-12
- “Compare Categorical Array Elements” on page 8-20
- “Combine Categorical Arrays” on page 8-23
- “Combine Categorical Arrays Using Multiplication” on page 8-27
- “Access Data Using Categorical Arrays” on page 8-30
- “Work with Protected Categorical Arrays” on page 8-38
- “Advantages of Using Categorical Arrays” on page 8-43
- “Ordinal Categorical Arrays” on page 8-46
- “Core Functions Supporting Categorical Arrays” on page 8-50

Create Categorical Arrays

This example shows how to create a categorical array. `categorical` is a data type for storing data with values from a finite set of discrete categories. These categories can have a natural order, but it is not required. A categorical array provides efficient storage and convenient manipulation of data, while also maintaining meaningful names for the values. Categorical arrays are often used in a table to define groups of rows.

By default, categorical arrays contain categories that have no mathematical ordering. For example, the discrete set of pet categories {'dog' 'cat' 'bird'} has no meaningful mathematical ordering, so MATLAB® uses the alphabetical ordering {'bird' 'cat' 'dog'}. *Ordinal* categorical arrays contain categories that have a meaningful mathematical ordering. For example, the discrete set of size categories {'small', 'medium', 'large'} has the mathematical ordering `small < medium < large`.

When you create categorical arrays from cell arrays of character vectors or string arrays, leading and trailing spaces are removed. For example, if you specify the text {'cat' 'dog '} as categories, then when you convert them to categories they become {'cat' 'dog'}.

Create Categorical Array from Cell Array of Character Vectors

You can use the `categorical` function to create a categorical array from a numeric array, logical array, string array, cell array of character vectors, or an existing categorical array.

Create a 1-by-11 cell array of character vectors containing state names from New England.

```
state = {'MA', 'ME', 'CT', 'VT', 'ME', 'NH', 'VT', 'MA', 'NH', 'CT', 'RI'};
```

Convert the cell array, `state`, to a categorical array that has no mathematical order.

```
state = categorical(state)
```

```
state = 1x11 categorical array  
Columns 1 through 9
```

```
MA     ME     CT     VT     ME     NH     VT     MA     NH
```

```
Columns 10 through 11
```

```
CT     RI
```

```
class(state)
```

```
ans =  
'categorical'
```

List the discrete categories in the variable `state`.

```
categories(state)
```

```
ans = 6x1 cell array  
    {'CT'}  
    {'MA'}  
    {'ME'}  
    {'NH'}  
    {'RI'}  
    {'VT'}
```

The categories are listed in alphabetical order.

Create Ordinal Categorical Array from Cell Array of Character Vectors

Create a 1-by-8 cell array of character vectors containing the sizes of eight objects.

```
AllSizes = {'medium', 'large', 'small', 'small', 'medium', ...  
            'large', 'medium', 'small'};
```

The cell array, `AllSizes`, has three distinct values: 'large', 'medium', and 'small'. With the cell array of character vectors, there is no convenient way to indicate that `small < medium < large`.

Convert the cell array, `AllSizes`, to an ordinal categorical array. Use `valueset` to specify the values `small`, `medium`, and `large`, which define the categories. For an ordinal categorical array, the first category specified is the smallest and the last category is the largest.

```
valueset = {'small', 'medium', 'large'};  
sizeOrd = categorical(AllSizes, valueset, 'Ordinal', true)
```

```
sizeOrd = 1x8 categorical array  
Columns 1 through 6
```

```
    medium    large    small    small    medium    large
```

```
Columns 7 through 8  
    medium    small
```

```
class(sizeOrd)  
ans =  
'categorical'
```

The order of the values in the categorical array, `sizeOrd`, remains unchanged.

List the discrete categories in the categorical variable, `sizeOrd`.

```
categories(sizeOrd)  
ans = 3x1 cell array  
    {'small' }  
    {'medium' }  
    {'large' }
```

The categories are listed in the specified order to match the mathematical ordering `small < medium < large`.

Create Ordinal Categorical Array by Binning Numeric Data

Create a vector of 100 random numbers between zero and 50.

```
x = rand(100,1)*50;
```

Use the `discretize` function to create a categorical array by binning the values of `x`. Put all values between zero and 15 in the first bin, all the values between 15 and 35 in the second bin, and all the values between 35 and 50 in the third bin. Each bin includes the left endpoint, but does not include the right endpoint.

```
catnames = {'small', 'medium', 'large'};  
binnedData = discretize(x,[0 15 35 50], 'categorical', catnames);
```

`binnedData` is a 100-by-1 ordinal categorical array with three categories, such that `small < medium < large`.

Use the `summary` function to print the number of elements in each category.

```
summary(binnedData)
```

```

small      30
medium    35
large     35

```

Create Categorical Array from String Array

Starting in R2016b, you can create string arrays with the `string` function and convert them to categorical array.

Create a string array that contains names of planets.

```

str = string({'Earth', 'Jupiter', 'Neptune', 'Jupiter', 'Mars', 'Earth'})
str = 1x6 string array
    "Earth"    "Jupiter"    "Neptune"    "Jupiter"    "Mars"    "Earth"

```

Convert `str` to a categorical array.

```

planets = categorical(str)
planets = 1x6 categorical array
    Earth    Jupiter    Neptune    Jupiter    Mars    Earth

```

Add missing elements to `str` and convert it to a categorical array. Where `str` has missing values, `planets` has undefined values.

```

str(8) = 'Mars'
str = 1x8 string array
Columns 1 through 6
    "Earth"    "Jupiter"    "Neptune"    "Jupiter"    "Mars"    "Earth"
Columns 7 through 8
    <missing>    "Mars"
planets = categorical(str)
planets = 1x8 categorical array
Columns 1 through 6
    Earth    Jupiter    Neptune    Jupiter    Mars    Earth

```

Columns 7 through 8

<undefined> Mars

See Also

[categorical](#) | [categories](#) | [discretize](#) | [summary](#)

Related Examples

- “Convert Text in Table Variables to Categorical” on page 8-7
- “Access Data Using Categorical Arrays” on page 8-30
- “Compare Categorical Array Elements” on page 8-20

More About

- “Advantages of Using Categorical Arrays” on page 8-43
- “Ordinal Categorical Arrays” on page 8-46

Convert Text in Table Variables to Categorical

This example shows how to convert a variable in a table from a cell array of character vectors to a categorical array.

Load Sample Data and Create a Table

Load sample data gathered from 100 patients.

```
load patients
```

```
whos
```

Name	Size	Bytes	Class	Attributes
Age	100x1	800	double	
Diastolic	100x1	800	double	
Gender	100x1	12212	cell	
Height	100x1	800	double	
LastName	100x1	12416	cell	
Location	100x1	15008	cell	
SelfAssessedHealthStatus	100x1	12340	cell	
Smoker	100x1	100	logical	
Systolic	100x1	800	double	
Weight	100x1	800	double	

Store the patient data from Age, Gender, Height, Weight, SelfAssessedHealthStatus, and Location in a table. Use the unique identifiers in the variable LastName as row names.

```
T = table(Age,Gender,Height,Weight,...
         SelfAssessedHealthStatus,Location,...
         'RowNames',LastName);
```

Convert Table Variables from Cell Arrays of Character Vectors to Categorical Arrays

The cell arrays of character vectors, Gender and Location, contain discrete sets of unique values.

Convert Gender and Location to categorical arrays.

```
T.Gender = categorical(T.Gender);
T.Location = categorical(T.Location);
```

The variable, `SelfAssessedHealthStatus`, contains four unique values: Excellent, Fair, Good, and Poor.

Convert `SelfAssessedHealthStatus` to an ordinal categorical array, such that the categories have the mathematical ordering `Poor < Fair < Good < Excellent`.

```
T.SelfAssessedHealthStatus = categorical(T.SelfAssessedHealthStatus,...
    {'Poor', 'Fair', 'Good', 'Excellent'}, 'Ordinal', true);
```

Print a Summary

View the data type, description, units, and other descriptive statistics for each variable by using `summary` to summarize the table.

```
format compact
```

```
summary(T)
```

```
Variables:
```

```
Age: 100x1 double
```

```
Values:
```

```
Min      25
Median   39
Max      50
```

```
Gender: 100x1 categorical
```

```
Values:
```

```
Female   53
Male     47
```

```
Height: 100x1 double
```

```
Values:
```

```
Min      60
Median   67
Max      72
```

```
Weight: 100x1 double
```

```
Values:
```

```
Min      111
Median   142.5
Max      202
```

```
SelfAssessedHealthStatus: 100x1 ordinal categorical
```

```
Values:
```

```
Poor           11
Fair           15
Good           40
Excellent     34
```

```
Location: 100x1 categorical
```

```

Values:
  County General Hospital      39
  St. Mary s Medical Center   24
  VA Hospital                  37

```

The table variables `Gender`, `SelfAssessedHealthStatus`, and `Location` are categorical arrays. The summary contains the counts of the number of elements in each category. For example, the summary indicates that 53 of the 100 patients are female and 47 are male.

Select Data Based on Categories

Create a subtable, `T1`, containing the age, height, and weight of all female patients who were observed at County General Hospital. You can easily create a logical vector based on the values in the categorical arrays `Gender` and `Location`.

```
rows = T.Location=='County General Hospital' & T.Gender=='Female';
```

`rows` is a 100-by-1 logical vector with logical `true` (1) for the table rows where the gender is female and the location is County General Hospital.

Define the subset of variables.

```
vars = {'Age', 'Height', 'Weight'};
```

Use parentheses to create the subtable, `T1`.

```
T1 = T(rows,vars)
```

`T1=19×3 table`

	Age	Height	Weight
Brown	49	64	119
Taylor	31	66	132
Anderson	45	68	128
Lee	44	66	146
Walker	28	65	123
Young	25	63	114
Campbell	37	65	135
Evans	39	62	121
Morris	43	64	135
Rivera	29	63	130
Richardson	30	67	141
Cox	28	66	111
Torres	45	70	137

Peterson	32	60	136	
Ramirez	48	64	137	
Bennett	35	64	131	:

A is a 19-by-3 table.

Since ordinal categorical arrays have a mathematical ordering for their categories, you can perform element-wise comparisons of them with relational operations, such as greater than and less than.

Create a subtable, T2, of the gender, age, height, and weight of all patients who assessed their health status as poor or fair.

First, define the subset of rows to include in table T2.

```
rows = T.SelfAssessedHealthStatus<='Fair';
```

Then, define the subset of variables to include in table T2.

```
vars = {'Gender', 'Age', 'Height', 'Weight'};
```

Use parentheses to create the subtable T2.

```
T2 = T(rows,vars)
```

T2=26×4 table

	Gender	Age	Height	Weight
Johnson	Male	43	69	163
Jones	Female	40	67	133
Thomas	Female	42	66	137
Jackson	Male	25	71	174
Garcia	Female	27	69	131
Rodriguez	Female	39	64	117
Lewis	Female	41	62	137
Lee	Female	44	66	146
Hall	Male	25	70	189
Hernandez	Male	36	68	166
Lopez	Female	40	66	137
Gonzalez	Female	35	66	118
Mitchell	Male	39	71	164
Campbell	Female	37	65	135
Parker	Male	30	68	182

Stewart Male 49 68 170 :

T2 is a 26-by-4 table.

See Also

Related Examples

- “Create and Work with Tables” on page 9-2
- “Create Categorical Arrays” on page 8-2
- “Access Data in a Table” on page 9-42
- “Access Data Using Categorical Arrays” on page 8-30

More About

- “Advantages of Using Categorical Arrays” on page 8-43
- “Ordinal Categorical Arrays” on page 8-46

Plot Categorical Data

This example shows how to plot data from a categorical array.

Load Sample Data

Load sample data gathered from 100 patients.

```
load patients
```

```
whos
```

Name	Size	Bytes	Class	Attributes
Age	100x1	800	double	
Diastolic	100x1	800	double	
Gender	100x1	12212	cell	
Height	100x1	800	double	
LastName	100x1	12416	cell	
Location	100x1	15008	cell	
SelfAssessedHealthStatus	100x1	12340	cell	
Smoker	100x1	100	logical	
Systolic	100x1	800	double	
Weight	100x1	800	double	

Create Categorical Arrays from Cell Arrays of Character Vectors

The workspace variable, `Location`, is a cell array of character vectors that contains the three unique medical facilities where patients were observed.

To access and compare data more easily, convert `Location` to a categorical array.

```
Location = categorical(Location);
```

Summarize the categorical array.

```
summary(Location)
```

```
County General Hospital    39
St. Mary's Medical Center  24
VA Hospital                 37
```

39 patients were observed at County General Hospital, 24 at St. Mary's Medical Center, and 37 at the VA Hospital.

The workspace variable, `SelfAssessedHealthStatus`, contains four unique values, Excellent, Fair, Good, and Poor.

Convert `SelfAssessedHealthStatus` to an ordinal categorical array, such that the categories have the mathematical ordering `Poor < Fair < Good < Excellent`.

```
SelfAssessedHealthStatus = categorical(SelfAssessedHealthStatus,...  
    {'Poor' 'Fair' 'Good' 'Excellent'}, 'Ordinal', true);
```

Summarize the categorical array, `SelfAssessedHealthStatus`.

```
summary(SelfAssessedHealthStatus)
```

Poor	11
Fair	15
Good	40
Excellent	34

Plot Histogram

Create a histogram bar plot directly from a categorical array.

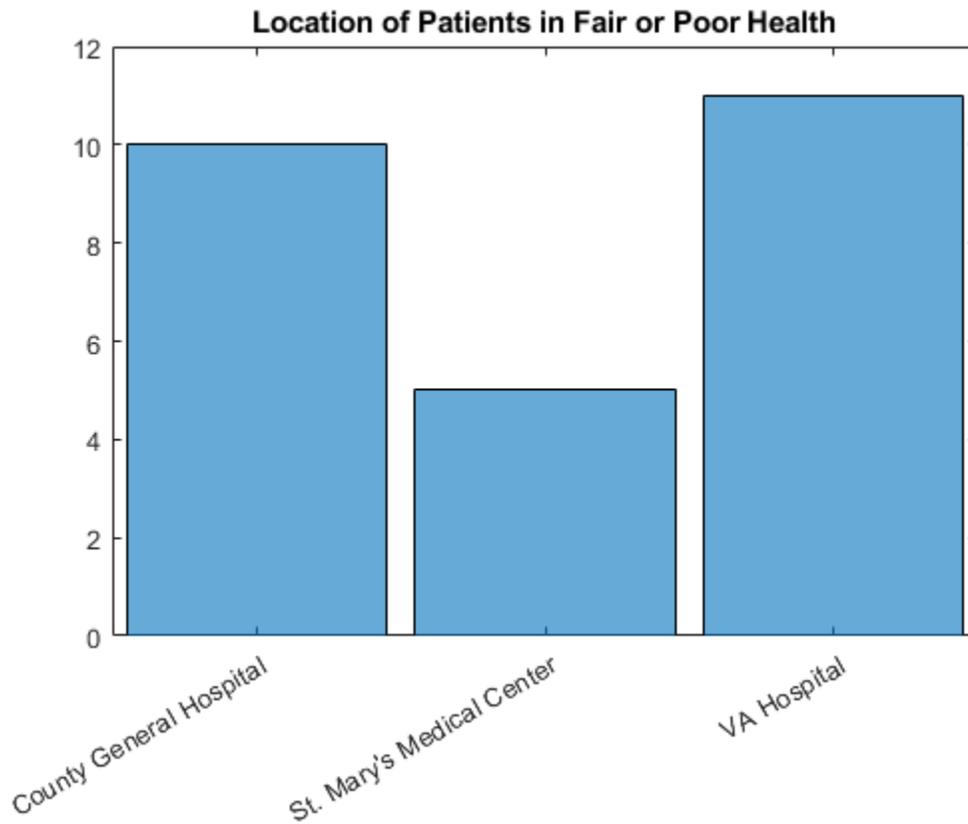
```
figure  
histogram(SelfAssessedHealthStatus)  
title('Self Assessed Health Status From 100 Patients')
```



The function `histogram` accepts the categorical array, `SelfAssessedHealthStatus`, and plots the category counts for each of the four categories.

Create a histogram of the hospital location for only the patients who assessed their health as Fair or Poor.

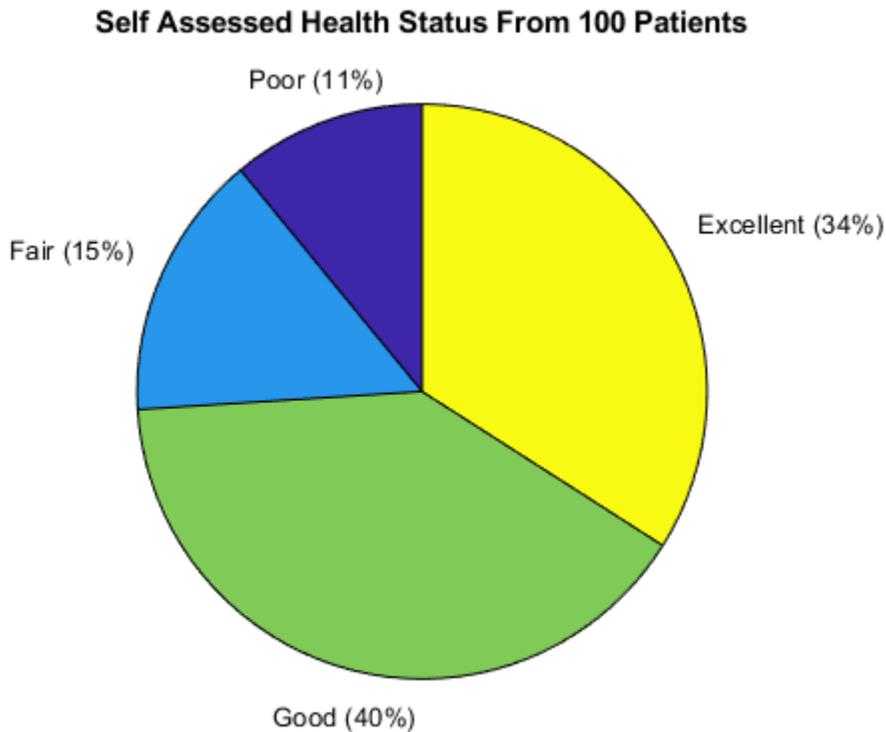
```
figure
histogram(Location(SelfAssessedHealthStatus<='Fair'))
title('Location of Patients in Fair or Poor Health')
```



Create Pie Chart

Create a pie chart directly from a categorical array.

```
figure  
pie(SelfAssessedHealthStatus);  
title('Self Assessed Health Status From 100 Patients')
```

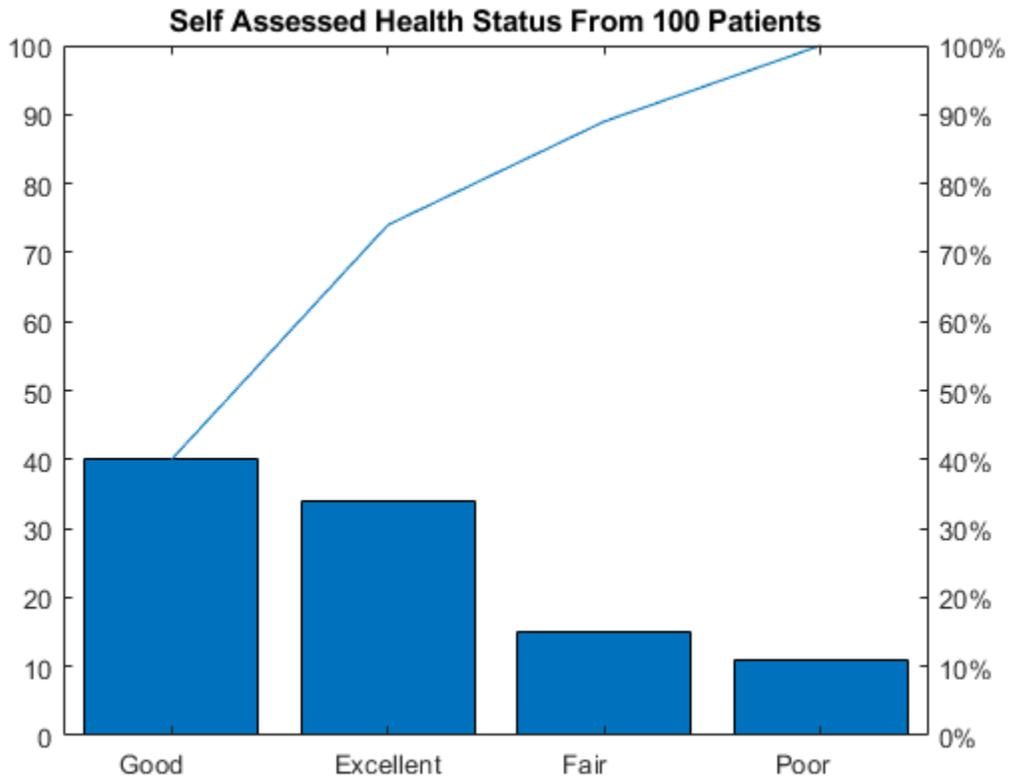


The function `pie` accepts the categorical array, `SelfAssessedHealthStatus`, and plots a pie chart of the four categories.

Create Pareto Chart

Create a Pareto chart from the category counts for each of the four categories of `SelfAssessedHealthStatus`.

```
figure
A = countcats(SelfAssessedHealthStatus);
C = categories(SelfAssessedHealthStatus);
pareto(A,C);
title('Self Assessed Health Status From 100 Patients')
```



The first input argument to `pareto` must be a vector. If a categorical array is a matrix or multidimensional array, reshape it into a vector before calling `countcats` and `pareto`.

Create Scatter Plot

Convert the cell array of character vectors to a categorical array.

```
Gender = categorical(Gender);
```

Summarize the categorical array, `Gender`.

```
summary(Gender)
```

```
Female    53
Male      47
```

Gender is a 100-by-1 categorical array with two categories, Female and Male.

Use the categorical array, Gender, to access Weight and Height data for each gender separately.

```
X1 = Weight(Gender=='Female');  
Y1 = Height(Gender=='Female');
```

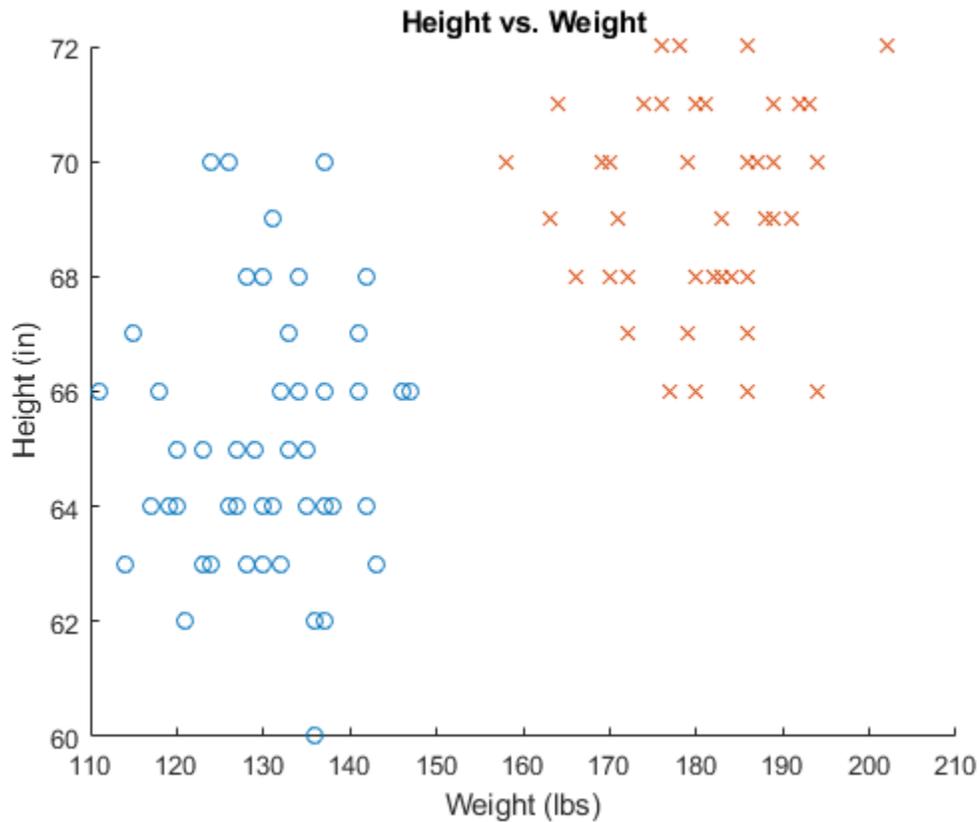
```
X2 = Weight(Gender=='Male');  
Y2 = Height(Gender=='Male');
```

X1 and Y1 are 53-by-1 numeric arrays containing data from the female patients.

X2 and Y2 are 47-by-1 numeric arrays containing data from the male patients.

Create a scatter plot of height vs. weight. Indicate data from the female patients with a circle and data from the male patients with a cross.

```
figure  
h1 = scatter(X1,Y1,'o');  
hold on  
h2 = scatter(X2,Y2,'x');  
  
title('Height vs. Weight')  
xlabel('Weight (lbs)')  
ylabel('Height (in)')
```



See Also

[bar](#) | [categorical](#) | [countcats](#) | [histogram](#) | [pie](#) | [rose](#) | [scatter](#) | [summary](#)

Related Examples

- “Access Data Using Categorical Arrays” on page 8-30

Compare Categorical Array Elements

This example shows how to use relational operations with a categorical array.

Create Categorical Array from Cell Array of Character Vectors

Create a 2-by-4 cell array of character vectors.

```
C = {'blue' 'red' 'green' 'blue';...  
    'blue' 'green' 'green' 'blue'};
```

```
colors = categorical(C)
```

```
colors = 2x4 categorical array  
    blue    red    green    blue  
    blue    green  green    blue
```

colors is a 2-by-4 categorical array.

List the categories of the categorical array.

```
categories(colors)
```

```
ans = 3x1 cell array  
    {'blue' }  
    {'green'}  
    {'red'  }
```

Determine If Elements Are Equal

Use the relational operator, eq (==), to compare the first and second rows of colors.

```
colors(1,:) == colors(2,:)
```

```
ans = 1x4 logical array  
    1    0    1    1
```

Only the values in the second column differ between the rows.

Compare Entire Array to Character Vector

Compare the entire categorical array, `colors`, to the character vector `'blue'` to find the location of all blue values.

```
colors == 'blue'
ans = 2x4 logical array

     1     0     0     1
     1     0     0     1
```

There are four blue entries in `colors`, one in each corner of the array.

Convert to an Ordinal Categorical Array

Add a mathematical ordering to the categories in `colors`. Specify the category order that represents the ordering of color spectrum, `red < green < blue`.

```
colors = categorical(colors,{'red','green','blue'},'Ordinal',true)
colors = 2x4 categorical array
      blue      red      green      blue
      blue      green     green     blue
```

The elements in the categorical array remain the same.

List the discrete categories in `colors`.

```
categories(colors)
ans = 3x1 cell array
    {'red' }
    {'green'}
    {'blue' }
```

Compare Elements Based on Order

Determine if elements in the first column of `colors` are greater than the elements in the second column.

```
colors(:,1) > colors(:,2)
```

```
ans = 2x1 logical array
```

```
1  
1
```

Both values in the first column, **blue**, are greater than the corresponding values in the second column, **red** and **green**.

Find all the elements in **colors** that are less than 'blue'.

```
colors < 'blue'
```

```
ans = 2x4 logical array
```

```
0 1 1 0  
0 1 1 0
```

The function `lt (<)` indicates the location of all green and red values with 1.

See Also

`categorical` | `categories`

Related Examples

- “Access Data Using Categorical Arrays” on page 8-30

More About

- “Relational Operations”
- “Advantages of Using Categorical Arrays” on page 8-43
- “Ordinal Categorical Arrays” on page 8-46

Combine Categorical Arrays

This example shows how to combine two categorical arrays.

Create Categorical Arrays

Create a categorical array, A, containing the preferred lunchtime beverage of 25 students in classroom A.

```
A = gallery('integerdata',3,[25,1],1);
A = categorical(A,1:3,{'milk' 'water' 'juice'});
```

A is a 25-by-1 categorical array with three distinct categories: milk, water, and juice.

Summarize the categorical array, A.

```
summary(A)

    milk      8
    water      8
    juice      9
```

Eight students in classroom A prefer milk, eight prefer water, and nine prefer juice.

Create another categorical array, B, containing the preferences of 28 students in classroom B.

```
B = gallery('integerdata',3,[28,1],3);
B = categorical(B,1:3,{'milk' 'water' 'juice'});
```

B is a 28-by-1 categorical array containing the same categories as A.

Summarize the categorical array, B.

```
summary(B)

    milk      12
    water      10
    juice       6
```

Twelve students in classroom B prefer milk, ten prefer water, and six prefer juice.

Concatenate Categorical Arrays

Concatenate the data from classrooms A and B into a single categorical array, Group1.

```
Group1 = [A;B];
```

Summarize the categorical array, Group1

```
summary(Group1)
```

```
    milk      20  
    water     18  
    juice     15
```

Group1 is a 53-by-1 categorical array with three categories: milk, water, and juice.

Create Categorical Array with Different Categories

Create a categorical array, Group2, containing data from 50 students who were given the additional beverage option of soda.

```
Group2 = gallery('integerdata',4,[50,1],2);  
Group2 = categorical(Group2,1:4,{'juice' 'milk' 'soda' 'water'});
```

Summarize the categorical array, Group2.

```
summary(Group2)
```

```
    juice     18  
    milk      10  
    soda      13  
    water      9
```

Group2 is a 50-by-1 categorical array with four categories: juice, milk, soda, and water.

Concatenate Arrays with Different Categories

Concatenate the data from Group1 and Group2.

```
students = [Group1;Group2];
```

Summarize the resulting categorical array, students.

```
summary(students)
```

```
    milk      30  
    water     27  
    juice     33  
    soda      13
```

Concatenation appends the categories exclusive to the second input, `soda`, to the end of the list of categories from the first input, `milk`, `water`, `juice`, `soda`.

Use `reordercats` to change the order of the categories in the categorical array, `students`.

```
students = reordercats(students,{'juice','milk','water','soda'});
```

```
categories(students)
```

```
ans = 4x1 cell array
    {'juice'}
    {'milk' }
    {'water'}
    {'soda' }
```

Union of Categorical Arrays

Use the function `union` to find the unique responses from `Group1` and `Group2`.

```
C = union(Group1,Group2)
```

```
C = 4x1 categorical array
    milk
    water
    juice
    soda
```

`union` returns the combined values from `Group1` and `Group2` with no repetitions. In this case, `C` is equivalent to the categories of the concatenation, `students`.

All of the categorical arrays in this example were nonordinal. To combine ordinal categorical arrays, they must have the same sets of categories including their order.

See Also

`cat` | `categorical` | `categories` | `horzcat` | `summary` | `union` | `vertcat`

Related Examples

- “Create Categorical Arrays” on page 8-2

- “Combine Categorical Arrays Using Multiplication” on page 8-27
- “Convert Text in Table Variables to Categorical” on page 8-7
- “Access Data Using Categorical Arrays” on page 8-30

More About

- “Ordinal Categorical Arrays” on page 8-46

Combine Categorical Arrays Using Multiplication

This example shows how to use the `times` function to combine categorical arrays, including ordinal categorical arrays and arrays with undefined elements. When you call `times` on two categorical arrays, the output is a categorical array with new categories. The set of new categories is the set of all the ordered pairs created from the categories of the input arrays, or the Cartesian product. `times` forms each element of the output array as the ordered pair of the corresponding elements of the input arrays. The output array has the same size as the input arrays.

Combine Two Categorical Arrays

Combine two categorical arrays using `times`. The input arrays must have the same number of elements, but can have different numbers of categories.

```
A = categorical({'blue', 'red', 'green'});
B = categorical({'+', '-', '+'});
C = A.*B
```

```
C = 1x3 categorical array
    blue +      red -      green +
```

Cartesian Product of Categories

Show the categories of C. The categories are all the ordered pairs that can be created from the categories of A and B, also known as the Cartesian product.

```
categories(C)

ans = 6x1 cell array
    {'blue '+' }
    {'blue -'  }
    {'green '+' }
    {'green -'  }
    {'red '+'  }
    {'red -'   }
```

As a consequence, `A.*B` does not equal `B.*A`.

```
D = B.*A
```

```
D = 1x3 categorical array
    + blue      - red      + green
```

```
categories(D)
```

```
ans = 6x1 cell array
    {'+ blue' }
    {'+ green' }
    {'+ red'   }
    {'- blue'  }
    {'- green' }
    {'- red'   }
```

Multiplication with Undefined Elements

Combine two categorical arrays. If either A or B have an undefined element, the corresponding element of C is undefined.

```
A = categorical({'blue', 'red', 'green', 'black'});
B = categorical({'+', '-', '+', '-'});
A = removecats(A,{'black'});
C = A.*B
```

```
C = 1x4 categorical array
    blue +      red -      green +      <undefined>
```

Cartesian Product of Ordinal Categorical Arrays

Combine two ordinal categorical arrays. C is an ordinal categorical array only if A and B are both ordinal. The ordering of the categories of C follows from the orderings of the input categorical arrays.

```
A = categorical({'blue', 'red', 'green'}, {'green', 'red', 'blue'}, 'Ordinal', true);
B = categorical({'+', '-', '+'}, 'Ordinal', true);
C = A.*B;
categories(C)
```

```
ans = 6x1 cell array
    {'green +' }
    {'green -' }
    {'red +'   }
    {'red -'   }
```

```
{'blue +' }  
{'blue -' }
```

See Also

[categorical](#) | [categories](#) | [summary](#) | [times](#)

Related Examples

- [“Create Categorical Arrays”](#) on page 8-2
- [“Combine Categorical Arrays”](#) on page 8-23
- [“Access Data Using Categorical Arrays”](#) on page 8-30

More About

- [“Ordinal Categorical Arrays”](#) on page 8-46

Access Data Using Categorical Arrays

In this section...

“Select Data By Category” on page 8-30

“Common Ways to Access Data Using Categorical Arrays” on page 8-30

Select Data By Category

Selecting data based on its values is often useful. This type of data selection can involve creating a logical vector based on values in one variable, and then using that logical vector to select a subset of values in other variables. You can create a logical vector for selecting data by finding values in a numeric array that fall within a certain range. Additionally, you can create the logical vector by finding specific discrete values. When using categorical arrays, you can easily:

- **Select elements from particular categories.** For categorical arrays, use the logical operators `==` or `~=` to select data that is in, or not in, a particular category. To select data in a particular group of categories, use the `ismember` function.

For ordinal categorical arrays, use inequalities `>`, `>=`, `<`, or `<=` to find data in categories above or below a particular category.

- **Delete data that is in a particular category.** Use logical operators to include or exclude data from particular categories.
- **Find elements that are not in a defined category.** Categorical arrays indicate which elements do not belong to a defined category by `<undefined>`. Use the `isundefined` function to find observations without a defined value.

Common Ways to Access Data Using Categorical Arrays

This example shows how to index and search using categorical arrays. You can access data using categorical arrays stored within a table in a similar manner.

Load Sample Data

Load sample data gathered from 100 patients.

```
load patients
whos
```

Name	Size	Bytes	Class	Attributes
Age	100x1	800	double	
Diastolic	100x1	800	double	
Gender	100x1	12212	cell	
Height	100x1	800	double	
LastName	100x1	12416	cell	
Location	100x1	15008	cell	
SelfAssessedHealthStatus	100x1	12340	cell	
Smoker	100x1	100	logical	
Systolic	100x1	800	double	
Weight	100x1	800	double	

Create Categorical Arrays from Cell Arrays of Character Vectors

Gender and Location contain data that belong in categories. Each cell array contains character vectors taken from a small set of unique values (indicating two genders and three locations respectively). Convert Gender and Location to categorical arrays.

```
Gender = categorical(Gender);
Location = categorical(Location);
```

Search for Members of a Single Category

For categorical arrays, you can use the logical operators == and ~= to find the data that is in, or not in, a particular category.

Determine if there are any patients observed at the location, 'Rampart General Hospital'.

```
any(Location=='Rampart General Hospital')
```

```
ans = logical
     0
```

There are no patients observed at Rampart General Hospital.

Search for Members of a Group of Categories

You can use ismember to find data in a particular group of categories. Create a logical vector for the patients observed at County General Hospital or VA Hospital.

```
VA_CountyGenIndex = ...
    ismember(Location,{'County General Hospital','VA Hospital'});
```

`VA_CountyGenIndex` is a 100-by-1 logical array containing logical true (1) for each element in the categorical array `Location` that is a member of the category `County General Hospital` or `VA Hospital`. The output, `VA_CountyGenIndex` contains 76 nonzero elements.

Use the logical vector, `VA_CountyGenIndex` to select the `LastName` of the patients observed at either `County General Hospital` or `VA Hospital`.

```
VA_CountyGenPatients = LastName(VA_CountyGenIndex);
```

`VA_CountyGenPatients` is a 76-by-1 cell array of character vectors.

Select Elements in a Particular Category to Plot

Use the `summary` function to print a summary containing the category names and the number of elements in each category.

```
summary(Location)

    County General Hospital    39
    St. Mary's Medical Center  24
    VA Hospital                37
```

`Location` is a 100-by-1 categorical array with three categories. `County General Hospital` occurs in 39 elements, `St. Mary s Medical Center` in 24 elements, and `VA Hospital` in 37 elements.

Use the `summary` function to print a summary of `Gender`.

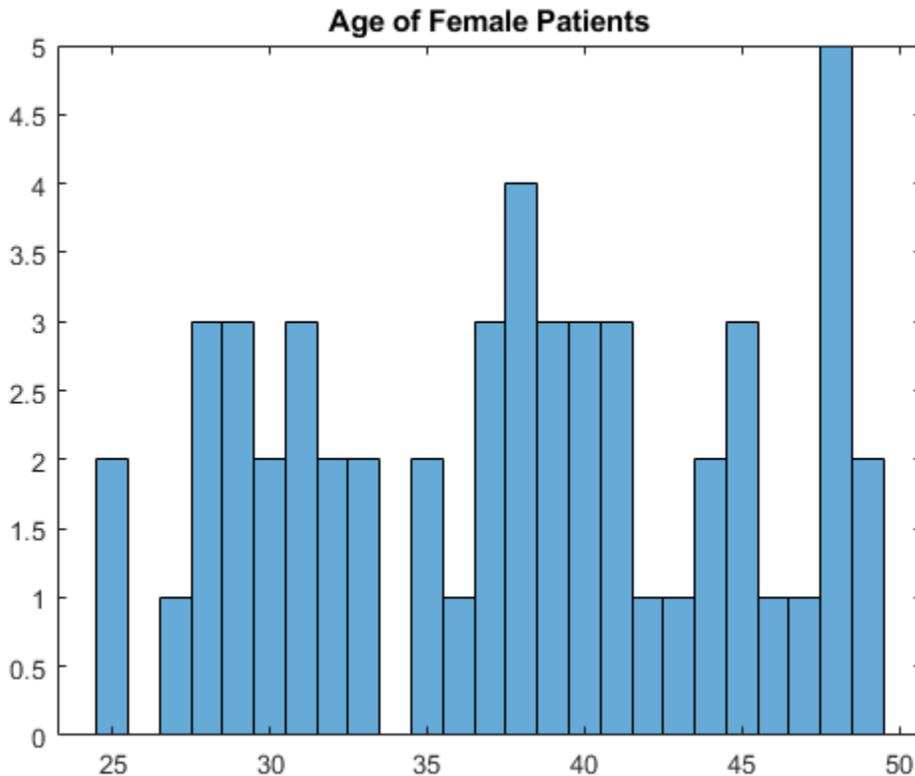
```
summary(Gender)

    Female    53
    Male      47
```

`Gender` is a 100-by-1 categorical array with two categories. `Female` occurs in 53 elements and `Male` occurs in 47 elements.

Use logical operator `==` to access the age of only the female patients. Then plot a histogram of this data.

```
figure()
histogram(Age(Gender=='Female'))
title('Age of Female Patients')
```



`histogram(Age(Gender=='Female'))` plots the age data for the 53 female patients.

Delete Data from a Particular Category

You can use logical operators to include or exclude data from particular categories. Delete all patients observed at VA Hospital from the workspace variables, `Age` and `Location`.

```
Age = Age(Location~='VA Hospital');
Location = Location(Location~='VA Hospital');
```

Now, `Age` is a 63-by-1 numeric array, and `Location` is a 63-by-1 categorical array.

List the categories of `Location`, as well as the number of elements in each category.

```
summary(Location)
```

```
County General Hospital    39
St. Mary's Medical Center  24
VA Hospital                 0
```

The patients observed at `VA Hospital` are deleted from `Location`, but `VA Hospital` is still a category.

Use the `removecats` function to remove `VA Hospital` from the categories of `Location`.

```
Location = removecats(Location, 'VA Hospital');
```

Verify that the category, `VA Hospital`, was removed.

```
categories(Location)
```

```
ans = 2x1 cell array
    {'County General Hospital' }
    {'St. Mary's Medical Center'}
```

`Location` is a 63-by-1 categorical array that has two categories.

Delete Element

You can delete elements by indexing. For example, you can remove the first element of `Location` by selecting the rest of the elements with `Location(2:end)`. However, an easier way to delete elements is to use `[]`.

```
Location(1) = [];
summary(Location)
```

```
County General Hospital    38
St. Mary's Medical Center  24
```

`Location` is a 62-by-1 categorical array that has two categories. Deleting the first element has no effect on other elements from the same category and does not delete the category itself.

Check for Undefined Data

Remove the category `County General Hospital` from `Location`.

```
Location = removecats(Location, 'County General Hospital');
```

Display the first eight elements of the categorical array, `Location`.

```
Location(1:8)
```

```
ans = 8x1 categorical array
    St. Mary's Medical Center
    <undefined>
    St. Mary's Medical Center
    St. Mary's Medical Center
    <undefined>
    <undefined>
    St. Mary's Medical Center
    St. Mary's Medical Center
```

After removing the category, `County General Hospital`, elements that previously belonged to that category no longer belong to any category defined for `Location`. Categorical arrays denote these elements as `undefined`.

Use the function `isundefined` to find observations that do not belong to any category.

```
undefinedIndex = isundefined(Location);
```

`undefinedIndex` is a 62-by-1 categorical array containing logical `true` (1) for all `undefined` elements in `Location`.

Set Undefined Elements

Use the `summary` function to print the number of `undefined` elements in `Location`.

```
summary(Location)

    St. Mary's Medical Center    24
    <undefined>                  38
```

The first element of `Location` belongs to the category, `St. Mary's Medical Center`. Set the first element to be `undefined` so that it no longer belongs to any category.

```
Location(1) = '<undefined>';
summary(Location)

    St. Mary's Medical Center    23
    <undefined>                  39
```

You can make selected elements `undefined` without removing a category or changing the categories of other elements. Set elements to be `undefined` to indicate elements with values that are unknown.

Preallocate Categorical Arrays with Undefined Elements

You can use `undefined` elements to preallocate the size of a categorical array for better performance. Create a categorical array that has elements with known locations only.

```
definedIndex = ~isundefined(Location);  
newLocation = Location(definedIndex);  
summary(newLocation)
```

```
St. Mary's Medical Center    23
```

Expand the size of `newLocation` so that it is a 200-by-1 categorical array. Set the last new element to be `undefined`. All of the other new elements also are set to be `undefined`. The 23 original elements keep the values they had.

```
newLocation(200) = '<undefined>';  
summary(newLocation)
```

```
St. Mary's Medical Center    23  
<undefined>                  177
```

`newLocation` has room for values you plan to store in the array later.

See Also

`any` | `categorical` | `categories` | `histogram` | `isundefined` | `removecats` | `summary`

Related Examples

- “Create Categorical Arrays” on page 8-2
- “Convert Text in Table Variables to Categorical” on page 8-7
- “Plot Categorical Data” on page 8-12
- “Compare Categorical Array Elements” on page 8-20
- “Work with Protected Categorical Arrays” on page 8-38

More About

- “Advantages of Using Categorical Arrays” on page 8-43
- “Ordinal Categorical Arrays” on page 8-46

Work with Protected Categorical Arrays

This example shows how to work with a categorical array with protected categories.

When you create a categorical array with the `categorical` function, you have the option of specifying whether or not the categories are protected. Ordinal categorical arrays always have protected categories, but you also can create a nonordinal categorical array that is protected using the `'Protected', true` name-value pair argument.

When you assign values that are not in the array's list of categories, the array updates automatically so that its list of categories includes the new values. Similarly, you can combine (nonordinal) categorical arrays that have different categories. The categories in the result include the categories from both arrays.

When you assign new values to a *protected* categorical array, the values must belong to one of the existing categories. Similarly, you can only combine protected arrays that have the same categories.

- If you want to combine two nonordinal categorical arrays that have protected categories, they must have the same categories, but the order does not matter. The resulting categorical array uses the category order from the first array.
- If you want to combine two ordinal categorical array (that always have protected categories), they must have the same categories, including their order.

To add new categories to the array, you must use the function `addcats`.

Create Ordinal Categorical Array

Create a categorical array containing the sizes of 10 objects. Use the names `small`, `medium`, and `large` for the values `'S'`, `'M'`, and `'L'`.

```
A = categorical({'M';'L';'S';'S';'M';'L';'M';'L';'M';'S'},...  
               {'S','M','L'},{'small','medium','large'},'Ordinal',true)
```

```
A = 10x1 categorical array  
    medium  
    large  
    small  
    small  
    medium  
    large  
    medium
```

```

large
medium
small

```

A is a 10-by-1 categorical array.

Display the categories of A.

```

categories(A)

ans = 3x1 cell array
    {'small' }
    {'medium'}
    {'large' }

```

Verify That Categories Are Protected

When you create an ordinal categorical array, the categories are always protected.

Use the `isprotected` function to verify that the categories of A are protected.

```

tf = isprotected(A)

tf = logical
    1

```

The categories of A are protected.

Assign Value in New Category

If you try to assign a new value that does not belong to one of the existing categories, then MATLAB® returns an error. For example, you cannot assign the value 'xlarge' to the categorical array, as in the expression `A(2) = 'xlarge'`, because xlarge is not a category of A. Instead, MATLAB® returns the error:

```

Error using categorical/subsasgn (line 68)
Cannot add a new category 'xlarge' to this categorical array
because its categories are protected. Use ADDCATS to
add the new category.

```

To add a new category for `xlarge`, use the `addcats` function. Since `A` is ordinal you must specify the order for the new category.

```
A = addcats(A, 'xlarge', 'After', 'large');
```

Now, assign a value for `'xlarge'`, since it has an existing category.

```
A(2) = 'xlarge'
```

```
A = 10x1 categorical array
    medium
    xlarge
    small
    small
    medium
    large
    medium
    large
    medium
    small
```

`A` is now a 10-by-1 categorical array with four categories, such that `small < medium < large < xlarge`.

Combine Two Ordinal Categorical Arrays

Create another ordinal categorical array, `B`, containing the sizes of five items.

```
B = categorical([2;1;1;2;2],1:2,{'xsmall','small'},'Ordinal',true)
```

```
B = 5x1 categorical array
    small
    xsmall
    xsmall
    small
    small
```

`B` is a 5-by-1 categorical array with two categories such that `xsmall < small`.

To combine two ordinal categorical arrays (which always have protected categories), they must have the same categories and the categories must be in the same order.

Add the category `'xsmall'` to `A` before the category `'small'`.

```
A = addcats(A, 'xsmall', 'Before', 'small');
```

```
categories(A)
```

```
ans = 5x1 cell array
    {'xsmall'}
    {'small' }
    {'medium'}
    {'large' }
    {'xlarge'}
```

Add the categories {'medium', 'large', 'xlarge'} to B after the category 'small'.

```
B = addcats(B, {'medium', 'large', 'xlarge'}, 'After', 'small');
```

```
categories(B)
```

```
ans = 5x1 cell array
    {'xsmall'}
    {'small' }
    {'medium'}
    {'large' }
    {'xlarge'}
```

The categories of A and B are now the same including their order.

Vertically concatenate A and B.

```
C = [A;B]
```

```
C = 15x1 categorical array
    medium
    xlarge
    small
    small
    medium
    large
    medium
    large
    medium
    small
    small
    xsmall
```

```
xsmall  
small  
small
```

The values from B are appended to the values from A.

List the categories of C.

```
categories(C)
```

```
ans = 5x1 cell array  
    {'xsmall'}  
    {'small' }  
    {'medium'}  
    {'large' }  
    {'xlarge'}
```

C is a 16-by-1 ordinal categorical array with five categories, such that `xsmall < small < medium < large < xlarge`.

See Also

[addcats](#) | [categorical](#) | [categories](#) | [isordinal](#) | [isprotected](#) | [summary](#)

Related Examples

- “Create Categorical Arrays” on page 8-2
- “Convert Text in Table Variables to Categorical” on page 8-7
- “Access Data Using Categorical Arrays” on page 8-30
- “Combine Categorical Arrays” on page 8-23
- “Combine Categorical Arrays Using Multiplication” on page 8-27

More About

- “Ordinal Categorical Arrays” on page 8-46

Advantages of Using Categorical Arrays

In this section...

“Natural Representation of Categorical Data” on page 8-43

“Mathematical Ordering for Character Vectors” on page 8-43

“Reduce Memory Requirements” on page 8-43

Natural Representation of Categorical Data

`categorical` is a data type to store data with values from a finite set of discrete categories. One common alternative to using categorical arrays is to use character arrays or cell arrays of character vectors. To compare values in character arrays and cell arrays of character vectors, you must use `strcmp` which can be cumbersome. With categorical arrays, you can use the logical operator `eq` (`==`) to compare elements in the same way that you compare numeric arrays. The other common alternative to using categorical arrays is to store categorical data using integers in numeric arrays. Using numeric arrays loses all the useful descriptive information from the category names, and also tends to suggest that the integer values have their usual numeric meaning, which, for categorical data, they do not.

Mathematical Ordering for Character Vectors

Categorical arrays are convenient and memory efficient containers for nonnumeric data with values from a finite set of discrete categories. They are especially useful when the categories have a meaningful mathematical ordering, such as an array with entries from the discrete set of categories `{'small', 'medium', 'large'}` where `small < medium < large`.

An ordering other than alphabetical order is not possible with character arrays or cell arrays of character vectors. Thus, inequality comparisons, such as greater and less than, are not possible. With categorical arrays, you can use relational operations to test for equality and perform element-wise comparisons that have a meaningful mathematical ordering.

Reduce Memory Requirements

This example shows how to compare the memory required to store data as a cell array of character vectors versus a categorical array. Categorical arrays have categories that are

defined as character vectors, which can be costly to store and manipulate in a cell array of character vectors or char array. Categorical arrays store only one copy of each category name, often reducing the amount of memory required to store the array.

Create a sample cell array of character vectors.

```
state = [ repmat({'MA'},25,1); repmat({'NY'},25,1); ...  
         repmat({'CA'},50,1); ...  
         repmat({'MA'},25,1); repmat({'NY'},25,1) ];
```

Display information about the variable `state`.

```
whos state
```

Name	Size	Bytes	Class	Attributes
state	150x1	17400	cell	

The variable `state` is a cell array of character vectors requiring 17,400 bytes of memory.

Convert `state` to a categorical array.

```
state = categorical(state);
```

Display the discrete categories in the variable `state`.

```
categories(state)
```

```
ans = 3x1 cell array  
    {'CA'}  
    {'MA'}  
    {'NY'}
```

`state` contains 150 elements, but only three distinct categories.

Display information about the variable `state`.

```
whos state
```

Name	Size	Bytes	Class	Attributes
state	150x1	500	categorical	

There is a significant reduction in the memory required to store the variable.

See Also

categorical | categories

Related Examples

- “Create Categorical Arrays” on page 8-2
- “Convert Text in Table Variables to Categorical” on page 8-7
- “Compare Categorical Array Elements” on page 8-20
- “Access Data Using Categorical Arrays” on page 8-30

More About

- “Ordinal Categorical Arrays” on page 8-46

Ordinal Categorical Arrays

In this section...
“Order of Categories” on page 8-46
“How to Create Ordinal Categorical Arrays” on page 8-46
“Working with Ordinal Categorical Arrays” on page 8-48

Order of Categories

`categorical` is a data type to store data with values from a finite set of discrete categories, which can have a natural order. You can specify and rearrange the order of categories in all categorical arrays. However, you only can treat *ordinal* categorical arrays as having a mathematical ordering to their categories. Use an ordinal categorical array if you want to use the functions `min`, `max`, or relational operations, such as greater than and less than.

The discrete set of pet categories `{'dog' 'cat' 'bird'}` has no meaningful mathematical ordering. You are free to use any category order and the meaning of the associated data does not change. For example, `pets = categorical({'bird','cat','dog','dog','cat'})` creates a categorical array and the categories are listed in alphabetical order, `{'bird' 'cat' 'dog'}`. You can choose to specify or change the order of the categories to `{'dog' 'cat' 'bird'}` and the meaning of the data does not change.

ordinal categorical arrays contain categories that have a meaningful mathematical ordering. For example, the discrete set of size categories `{'small', 'medium', 'large'}` has the mathematical ordering `small < medium < large`. The first category listed is the smallest and the last category is the largest. The order of the categories in an ordinal categorical array affects the result from relational comparisons of ordinal categorical arrays.

How to Create Ordinal Categorical Arrays

This example shows how to create an ordinal categorical array using the `categorical` function with the `'Ordinal', true` name-value pair argument.

Ordinal Categorical Array from a Cell Array of Character Vectors

Create an ordinal categorical array, `sizes`, from a cell array of character vectors, `A`. Use `valueset`, specified as a vector of unique values, to define the categories for `sizes`.

```
A = {'medium' 'large'; 'small' 'medium'; 'large' 'small'};
valueset = {'small', 'medium', 'large'};
```

```
sizes = categorical(A,valueset,'Ordinal',true)
```

```
sizes = 3x2 categorical array
    medium    large
    small     medium
    large     small
```

`sizes` is 3-by-2 ordinal categorical array with three categories such that `small < medium < large`. The order of the values in `valueset` becomes the order of the categories of `sizes`.

Ordinal Categorical Array from Integers

Create an equivalent categorical array from an array of integers. Use the values 1, 2, and 3 to define the categories `small`, `medium`, and `large`, respectively.

```
A2 = [2 3; 1 2; 3 1];
valueset = 1:3;
catnames = {'small', 'medium', 'large'};
```

```
sizes2 = categorical(A2,valueset,catnames,'Ordinal',true)
```

```
sizes2 = 3x2 categorical array
    medium    large
    small     medium
    large     small
```

Compare `sizes` and `sizes2`

```
isequal(sizes,sizes2)
```

```
ans = logical
     1
```

sizes and sizes2 are equivalent categorical arrays with the same ordering of categories.

Convert a Categorical Array from Nonordinal to Ordinal

Create a nonordinal categorical array from the cell array of character vectors, A.

```
sizes3 = categorical(A)

sizes3 = 3x2 categorical array
    medium    large
    small     medium
    large     small
```

Determine if the categorical array is ordinal.

```
isordinal(sizes3)

ans = logical
     0
```

sizes3 is a nonordinal categorical array with three categories, {'large', 'medium', 'small'}. The categories of sizes3 are the sorted unique values from A. You must use the input argument, valueset, to specify a different category order.

Convert sizes3 to an ordinal categorical array, such that small < medium < large.

```
sizes3 = categorical(sizes3,{'small','medium','large'},'Ordinal',true);
```

sizes3 is now a 3-by-2 ordinal categorical array equivalent to sizes and sizes2.

Working with Ordinal Categorical Arrays

In order to combine or compare two categorical arrays, the sets of categories for both input arrays must be identical, including their order. Furthermore, ordinal categorical arrays are always protected. Therefore, when you assign values to an ordinal categorical array, the values must belong to one of the existing categories. For more information see “Work with Protected Categorical Arrays” on page 8-38.

See Also

`categorical` | `categories` | `isequal` | `isordinal`

Related Examples

- “Create Categorical Arrays” on page 8-2
- “Convert Text in Table Variables to Categorical” on page 8-7
- “Compare Categorical Array Elements” on page 8-20
- “Access Data Using Categorical Arrays” on page 8-30

More About

- “Advantages of Using Categorical Arrays” on page 8-43

Core Functions Supporting Categorical Arrays

Many functions in MATLAB operate on categorical arrays in much the same way that they operate on other arrays. A few of these functions might exhibit special behavior when operating on a categorical array. If multiple input arguments are ordinal categorical arrays, the function often requires that they have the same set of categories, including order. Furthermore, a few functions, such as `max` and `gt`, require that the input categorical arrays are ordinal.

The following table lists notable MATLAB functions that operate on categorical arrays in addition to other arrays.

size	isequal	intersect	plot	double
length	isequaln	ismember	plot3	single
ndims		setdiff	scatter	int8
numel	eq	setxor	scatter3	int16
isrow	ne	unique	bar	int32
iscolumn	lt	union	barh	int64
	le		histogram	uint8
cat	ge	times		uint16
horzcat	gt	sort	pie	uint32
vertcat		sortrows	rose	uint64
	min	issorted	stem	char
	max		stairs	string
	median	permute	area	cellstr
	mode	reshape	mesh	
		transpose	surf	
		ctranspose	surface	
			semilogx	
			semilogy	
			fill	
			fill3	
			line	
			text	

Tables

- “Create and Work with Tables” on page 9-2
- “Add and Delete Table Rows” on page 9-13
- “Add, Delete, and Rearrange Table Variables” on page 9-17
- “Clean Messy and Missing Data in Tables” on page 9-26
- “Modify Units, Descriptions, and Table Variable Names” on page 9-33
- “Add Custom Properties to Tables and Timetables” on page 9-36
- “Access Data in a Table” on page 9-42
- “Calculations on Tables” on page 9-51
- “Split Data into Groups and Calculate Statistics” on page 9-55
- “Split Table Data Variables and Apply Functions” on page 9-59
- “Advantages of Using Tables” on page 9-64
- “Grouping Variables To Split Data” on page 9-71
- “Changes to DimensionNames Property in R2016b” on page 9-75

Create and Work with Tables

This example shows how to create a table from workspace variables, work with table data, and write tables to files for later use. `table` is a data type for collecting heterogeneous data and metadata properties such as variable names, row names, descriptions, and variable units, in a single container.

Tables are suitable for column-oriented or tabular data that are often stored as columns in a text file or in a spreadsheet. Each variable in a table can have a different data type, but must have the same number of rows. However, variables in a table are not restricted to column vectors. For example, a table variable can contain a matrix with multiple columns as long as it has the same number of rows as the other table variables. A typical use for a table is to store experimental data, where rows represent different observations and columns represent different measured variables.

Tables are convenient containers for collecting and organizing related data variables and for viewing and summarizing data. For example, you can extract variables to perform calculations and conveniently add the results as new table variables. When you finish your calculations, write the table to a file to save your results.

Create and View Table

Create a table from workspace variables and view it. Alternatively, use the Import Tool or the `readtable` function to create a table from a spreadsheet or a text file. When you import data from a file using these functions, each column becomes a table variable.

Load sample data for 100 patients from the `patients` MAT-file to workspace variables.

```
load patients
whos
```

Name	Size	Bytes	Class	Attributes
Age	100x1	800	double	
Diastolic	100x1	800	double	
Gender	100x1	12212	cell	
Height	100x1	800	double	
LastName	100x1	12416	cell	
Location	100x1	15008	cell	
SelfAssessedHealthStatus	100x1	12340	cell	
Smoker	100x1	100	logical	
Systolic	100x1	800	double	
Weight	100x1	800	double	

Populate a table with column-oriented variables that contain patient data. You can access and assign table variables by name. When you assign a table variable from a workspace variable, you can assign the table variable a different name.

Create a table and populate it with the `Gender`, `Smoker`, `Height`, and `Weight` workspace variables. Display the first five rows.

```
T = table(Gender,Smoker,Height,Weight);
T(1:5,:)
```

```
ans=5x4 table
   Gender   Smoker   Height   Weight
   _____
   'Male'   true       71       176
   'Male'   false      69       163
   'Female' false      64       131
   'Female' false      67       133
   'Female' false      64       119
```

As an alternative, use the `readtable` function to read the patient data from a comma-delimited file. `readtable` reads all the columns that are in a file.

Create a table by reading all columns from the file, `patients.dat`.

```
T2 = readtable('patients.dat');
T2(1:5,:)
```

```
ans=5x10 table
   LastName   Gender   Age   Location   Height   Weight
   _____
   'Smith'    'Male'   38   'County General Hospital'   71       176
   'Johnson' 'Male'   43   'VA Hospital'               69       163
   'Williams' 'Female' 38   'St. Mary's Medical Center' 64       131
   'Jones'    'Female' 40   'VA Hospital'               67       133
   'Brown'    'Female' 49   'County General Hospital'   64       119
```

You can assign more column-oriented table variables using dot notation, `T.varname`, where `T` is the table and `varname` is the desired variable name. Create identifiers that are random numbers. Then assign them to a table variable, and name the table variable `ID`. All the variables you assign to a table must have the same number of rows. Display the first five rows of `T`.

```
T.ID = randi(1e4,100,1);
T(1:5,:)
```

```
ans=5x5 table
  Gender      Smoker      Height      Weight      ID
  -----
'Male'      true         71         176         8148
'Male'      false        69         163         9058
'Female'    false        64         131         1270
'Female'    false        67         133         9134
'Female'    false        64         119         6324
```

All the variables you assign to a table must have the same number of rows.

View the data type, description, units, and other descriptive statistics for each variable by creating a table summary using the `summary` function.

```
summary(T)
```

```
Variables:
```

```
Gender: 100x1 cell array of character vectors
```

```
Smoker: 100x1 logical
```

```
Values:
```

```
True         34
```

```
False        66
```

```
Height: 100x1 double
```

```
Values:
```

```
Min          60
```

```
Median       67
```

```
Max          72
```

```
Weight: 100x1 double
```

```
Values:
```

```
Min          111
```

```
Median      142.5
```

```
Max         202
```

```
ID: 100x1 double
```

```
Values:
```

```
Min          120
```

```
Median     5485.5
```

```
Max         9706
```

Return the size of the table.

```
size(T)
```

```
ans = 1x2
```

```
100    5
```

T contains 100 rows and 5 variables.

Create a new, smaller table containing the first five rows of T and display it. You can use numeric indexing within parentheses to specify rows and variables. This method is similar to indexing into numeric arrays to create subarrays. Tnew is a 5-by-5 table.

```
Tnew = T(1:5,:)
```

```
Tnew=5x5 table
   Gender   Smoker   Height   Weight   ID
   _____
   'Male'    true      71      176     8148
   'Male'    false     69      163     9058
   'Female'  false     64      131     1270
   'Female'  false     67      133     9134
   'Female'  false     64      119     6324
```

Create a smaller table containing all rows of Tnew and the variables from the second to the last. Use the end keyword to indicate the last variable or the last row of a table. Tnew is a 5-by-4 table.

```
Tnew = Tnew(:,2:end)
```

```
Tnew=5x4 table
   Smoker   Height   Weight   ID
   _____
   true     71      176     8148
   false    69      163     9058
   false    64      131     1270
   false    67      133     9134
   false    64      119     6324
```

Access Data by Row and Variable Names

Add row names to T and index into the table using row and variable names instead of numeric indices. Add row names by assigning the LastName workspace variable to the RowNames property of T.

```
T.Properties.RowNames = LastName;
```

Display the first five rows of T with row names.

```
T(1:5,:)
```

```
ans=5x5 table
   Gender   Smoker   Height   Weight   ID
```

Smith	'Male'	true	71	176	8148
Johnson	'Male'	false	69	163	9058
Williams	'Female'	false	64	131	1270
Jones	'Female'	false	67	133	9134
Brown	'Female'	false	64	119	6324

Return the size of T. The size does not change because row and variable names are not included when calculating the size of a table.

```
size(T)
```

```
ans = 1x2
```

```
100    5
```

Select all the data for the patients with the last names 'Smith' and 'Johnson'. In this case, it is simpler to use the row names than to use numeric indices. Tnew is a 2-by-5 table.

```
Tnew = T({'Smith', 'Johnson'}, :)
```

```
Tnew=2x5 table
```

	Gender	Smoker	Height	Weight	ID
Smith	'Male'	true	71	176	8148
Johnson	'Male'	false	69	163	9058

Select the height and weight of the patient named 'Johnson' by indexing on variable names. Tnew is a 1-by-2 table.

```
Tnew = T('Johnson', {'Height', 'Weight'})
```

```
Tnew=1x2 table
```

	Height	Weight
Johnson	69	163

You can access table variables either with dot syntax, as in T.Height, or by named indexing, as in T(:, 'Height').

Calculate and Add Result as Table Variable

You can access the contents of table variables, and then perform calculations on them using MATLAB® functions. Calculate body-mass-index (BMI) based on data in the existing

table variables and add it as a new variable. Plot the relationship of BMI to a patient's status as a smoker or a nonsmoker. Add blood-pressure readings to the table, and plot the relationship of blood pressure to BMI.

Calculate BMI using the table variables, `Weight` and `Height`. You can extract `Weight` and `Height` for the calculation while conveniently keeping `Weight`, `Height`, and `BMI` in the table with the rest of the patient data. Display the first five rows of `T`.

```
T.BMI = (T.Weight*0.453592)./(T.Height*0.0254).^2;
```

```
T(1:5,:)
```

```
ans=5x6 table
```

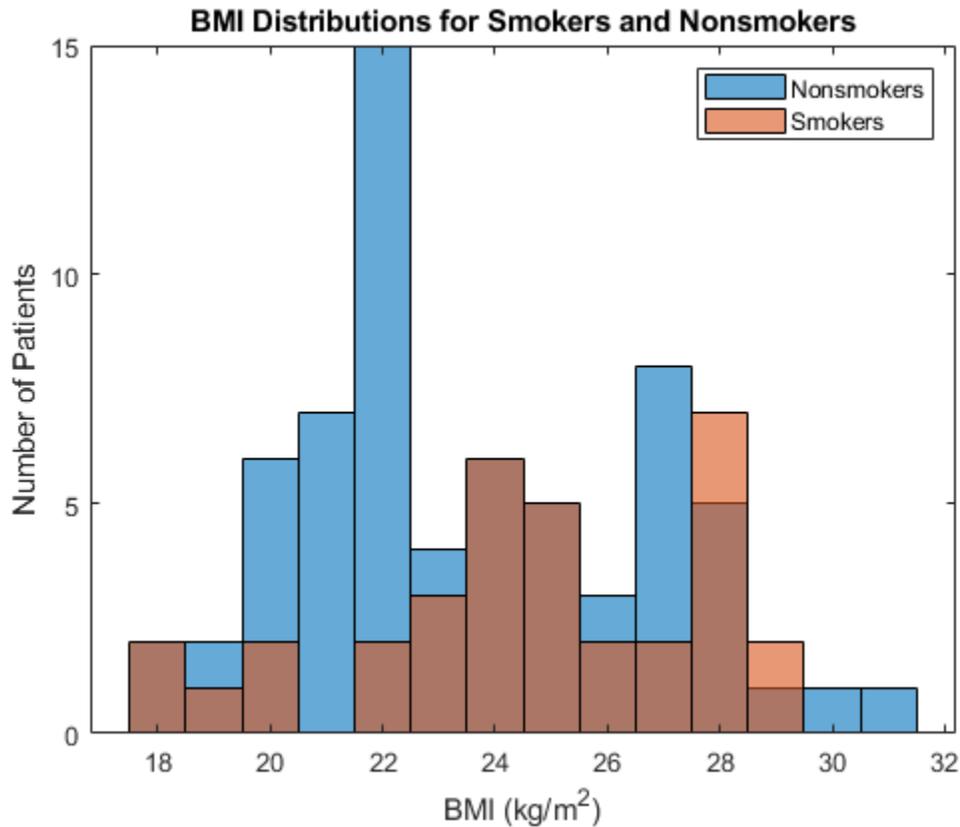
	Gender	Smoker	Height	Weight	ID	BMI
Smith	'Male'	true	71	176	8148	24.547
Johnson	'Male'	false	69	163	9058	24.071
Williams	'Female'	false	64	131	1270	22.486
Jones	'Female'	false	67	133	9134	20.831
Brown	'Female'	false	64	119	6324	20.426

Populate the variable units and variable descriptions properties for `BMI`. You can add metadata to any table variable to describe further the data contained in the variable.

```
T.Properties.VariableUnits{'BMI'} = 'kg/m^2';
T.Properties.VariableDescriptions{'BMI'} = 'Body Mass Index';
```

Create a histogram to explore whether there is a relationship between smoking and body-mass-index in this group of patients. You can index into `BMI` with the logical values from the `Smoker` table variable, because each row contains `BMI` and `Smoker` values for the same patient.

```
tf = (T.Smoker == false);
h1 = histogram(T.BMI(tf), 'BinMethod', 'integers');
hold on
tf = (T.Smoker == true);
h2 = histogram(T.BMI(tf), 'BinMethod', 'integers');
xlabel('BMI (kg/m^2)');
ylabel('Number of Patients');
legend('Nonsmokers', 'Smokers');
title('BMI Distributions for Smokers and Nonsmokers');
hold off
```



Add blood pressure readings for the patients from the workspace variables `Systolic` and `Diastolic`. Each row contains `Systolic`, `Diastolic`, and `BMI` values for the same patient.

```
T.Systolic = Systolic;
T.Diastolic = Diastolic;
```

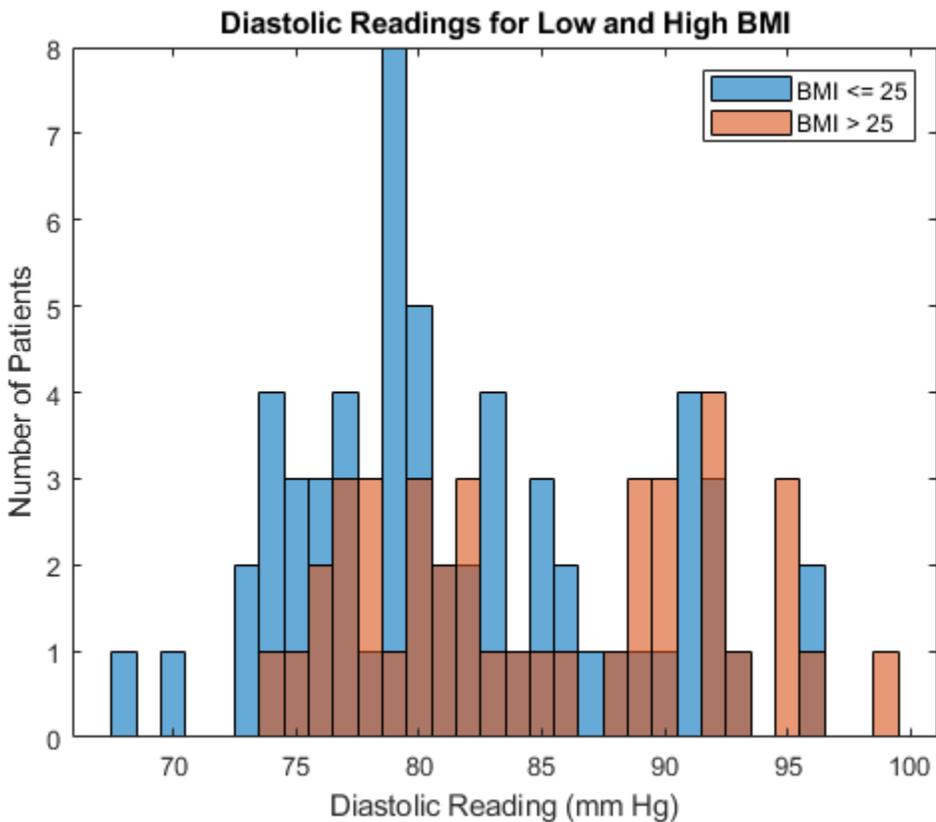
Create a histogram to show whether there is a relationship between high values of `Diastolic` and `BMI`.

```
tf = (T.BMI <= 25);
h1 = histogram(T.Diastolic(tf), 'BinMethod', 'integers');
hold on
tf = (T.BMI > 25);
```

```

h2 = histogram(T.Diastolic(tf), 'BinMethod', 'integers');
xlabel('Diastolic Reading (mm Hg)');
ylabel('Number of Patients');
legend('BMI <= 25', 'BMI > 25');
title('Diastolic Readings for Low and High BMI');
hold off

```



Reorder Table Variables and Rows for Output

To prepare the table for output, reorder the table rows by name, and table variables by position or name. Display the final arrangement of the table.

Sort the table by row names so that patients are listed in alphabetical order.

```
T = sortrows(T, 'RowNames');
```

```
T(1:5,:)
```

```
ans=5x8 table
```

	Gender	Smoker	Height	Weight	ID	BMI	Systolic
Adams	'Female'	false	66	137	8235	22.112	127
Alexander	'Male'	true	69	171	1300	25.252	128
Allen	'Female'	false	63	143	7432	25.331	113
Anderson	'Female'	false	68	128	1577	19.462	114
Bailey	'Female'	false	68	130	2239	19.766	113

Create a `BloodPressure` variable to hold blood pressure readings in a 100-by-2 table variable.

```
T.BloodPressure = [T.Systolic T.Diastolic];
```

Delete `Systolic` and `Diastolic` from the table since they are redundant.

```
T.Systolic = [];
T.Diastolic = [];
```

```
T(1:5,:)
```

```
ans=5x7 table
```

	Gender	Smoker	Height	Weight	ID	BMI	BloodPressure
Adams	'Female'	false	66	137	8235	22.112	127 8
Alexander	'Male'	true	69	171	1300	25.252	128 9
Allen	'Female'	false	63	143	7432	25.331	113 8
Anderson	'Female'	false	68	128	1577	19.462	114 7
Bailey	'Female'	false	68	130	2239	19.766	113 8

To put `ID` as the first column, reorder the table variables by position.

```
T = T(:, [5 1:4 6 7]);
```

```
T(1:5,:)
```

```
ans=5x7 table
```

	ID	Gender	Smoker	Height	Weight	BMI	BloodPressure
Adams	8235	'Female'	false	66	137	22.112	127 8
Alexander	1300	'Male'	true	69	171	25.252	128 9

Allen	7432	'Female'	false	63	143	25.331	113	8
Anderson	1577	'Female'	false	68	128	19.462	114	7
Bailey	2239	'Female'	false	68	130	19.766	113	8

You also can reorder table variables by name. To reorder the table variables so that Gender is last:

- 1 Find 'Gender' in the VariableNames property of the table.
- 2 Move 'Gender' to the end of a cell array of variable names.
- 3 Use the cell array of names to reorder the table variables.

```
varnames = T.Properties.VariableNames;
others = ~strcmp('Gender',varnames);
varnames = [varnames(others) 'Gender'];
T = T(:,varnames);
```

Display the first five rows of the reordered table.

```
T(1:5,:)
```

ans=5×7 table

	ID	Smoker	Height	Weight	BMI	BloodPressure	Gender
Adams	8235	false	66	137	22.112	127 83	'Female'
Alexander	1300	true	69	171	25.252	128 99	'Male'
Allen	7432	false	63	143	25.331	113 80	'Female'
Anderson	1577	false	68	128	19.462	114 77	'Female'
Bailey	2239	false	68	130	19.766	113 81	'Female'

Write Table to File

You can write the entire table to a file, or create a subtable to write a selected portion of the original table to a separate file.

Write T to a file with the `writetable` function.

```
writetable(T,'allPatientsBMI.txt');
```

You can use the `readtable` function to read the data in `allPatientsBMI.txt` into a new table.

Create a subtable and write the subtable to a separate file. Delete the rows that contain data on patients who are smokers. Then remove the `Smoker` variable. `nonsmokers` contains data only for the patients who are not smokers.

```
nonsmokers = T;  
toDelete = (nonsmokers.Smoker == true);  
nonsmokers(toDelete,:) = [];  
nonsmokers.Smoker = [];
```

Write nonsmokers to a file.

```
writetable(nonsmokers, 'nonsmokersBMI.txt');
```

See Also

Import Tool | array2table | cell2table | readtable | sortrows | struct2table | summary | table | writetable

Related Examples

- “Clean Messy and Missing Data in Tables” on page 9-26
- “Modify Units, Descriptions, and Table Variable Names” on page 9-33
- “Access Data in a Table” on page 9-42

More About

- “Advantages of Using Tables” on page 9-64

Add and Delete Table Rows

This example shows how to add and delete rows in a table. You can also edit tables using the Variables Editor.

Load Sample Data

Load the sample patients data and create a table, T.

```
load patients
T = table(LastName,Gender,Age,Height,Weight,Smoker,Systolic,Diastolic);
size(T)

ans = 1×2

    100     8
```

The table, T, has 100 rows and eight variables (columns).

Add Rows by Concatenation

Read data on more patients from a comma-delimited file, `morePatients.csv`, into a table, T2. Then, append the rows from T2 to the end of the table, T.

```
T2 = readtable('morePatients.csv');
Tnew = [T;T2];
size(Tnew)

ans = 1×2

    104     8
```

The table Tnew has 104 rows. In order to vertically concatenate two tables, both tables must have the same number of variables, with the same variable names. If the variable names are different, you can directly assign new rows in a table to rows from another table. For example, `T(end+1:end+4, :) = T2`.

Add Rows from Cell Array

To append new rows stored in a cell array, vertically concatenate the cell array onto the end of the table. You can concatenate directly from a cell array when it has the right

number of columns and the contents of its cells can be concatenated onto the corresponding table variables.

```
cellPatients = {'Edwards', 'Male', 42, 70, 158, 0, 116, 83;  
              'Falk', 'Female', 28, 62, 125, 1, 120, 71};  
Tnew = [Tnew; cellPatients];  
size(Tnew)
```

```
ans = 1×2
```

```
106    8
```

You also can convert a cell array to a table using the `cell2table` function.

Add Rows from Structure

You also can append new rows stored in a structure. Convert the structure to a table, and then concatenate the tables.

```
structPatients(1,1).LastName = 'George';  
structPatients(1,1).Gender = 'Male';  
structPatients(1,1).Age = 45;  
structPatients(1,1).Height = 76;  
structPatients(1,1).Weight = 182;  
structPatients(1,1).Smoker = 1;  
structPatients(1,1).Systolic = 132;  
structPatients(1,1).Diastolic = 85;
```

```
structPatients(2,1).LastName = 'Hadley';  
structPatients(2,1).Gender = 'Female';  
structPatients(2,1).Age = 29;  
structPatients(2,1).Height = 58;  
structPatients(2,1).Weight = 120;  
structPatients(2,1).Smoker = 0;  
structPatients(2,1).Systolic = 112;  
structPatients(2,1).Diastolic = 70;
```

```
Tnew = [Tnew; struct2table(structPatients)];  
size(Tnew)
```

```
ans = 1×2
```

```
108    8
```

Omit Duplicate Rows

To omit any rows in a table that are duplicated, use the `unique` function.

```
Tnew = unique(Tnew);
size(Tnew)
```

```
ans = 1×2
```

```
106    8
```

`unique` deleted two duplicate rows.

Delete Rows by Row Number

Delete rows 18, 20, and 21 from the table.

```
Tnew([18,20,21],:) = [];
size(Tnew)
```

```
ans = 1×2
```

```
103    8
```

The table contains information on 103 patients now.

Delete Rows by Row Name

First, specify the variable of identifiers, `LastName`, as row names. Then, delete the variable, `LastName`, from `Tnew`. Finally, use the row name to index and delete rows.

```
Tnew.Properties.RowNames = Tnew.LastName;
Tnew.LastName = [];
Tnew('Smith',:) = [];
size(Tnew)
```

```
ans = 1×2
```

```
102    7
```

The table now has one less row and one less variable.

Search for Rows to Delete

You also can search for observations in the table. For example, delete rows for any patients under the age of 30.

```
toDelete = Tnew.Age < 30;  
Tnew(toDelete,:) = [];  
size(Tnew)
```

```
ans = 1×2
```

```
85    7
```

The table now has 17 fewer rows.

See Also

[array2table](#) | [cell2table](#) | [readtable](#) | [struct2table](#) | [table](#)

Related Examples

- “Add, Delete, and Rearrange Table Variables” on page 9-17
- “Clean Messy and Missing Data in Tables” on page 9-26

Add, Delete, and Rearrange Table Variables

This example shows how to add, delete, and rearrange column-oriented variables in a table. You can add, move, and delete table variables using the `addvars`, `movevars`, and `removevars` functions. As alternatives, you also can modify table variables using dot syntax or by indexing into the table. Use the `splitvars` and `mergevars` functions to split multicolumn variables and combine multiple variables into one. Finally, you can reorient a table so that the rows of the table become variables of an output table, using the `rows2vars` function.

You also can modify table variables using the Variables Editor.

Load Sample Data and Create Tables

Load arrays of sample data from the `patients` MAT-file. Display the names and sizes of the variables loaded into the workspace.

```
load patients
whos -file patients
```

Name	Size	Bytes	Class	Attributes
Age	100x1	800	double	
Diastolic	100x1	800	double	
Gender	100x1	12212	cell	
Height	100x1	800	double	
LastName	100x1	12416	cell	
Location	100x1	15008	cell	
SelfAssessedHealthStatus	100x1	12340	cell	
Smoker	100x1	100	logical	
Systolic	100x1	800	double	
Weight	100x1	800	double	

Create two tables. Create one table, `T`, with information collected from a patient questionnaire and create another table, `T2`, with data measured from patients. Each table has 100 rows.

```
T = table(Age,Gender,Smoker);
T2 = table(Height,Weight,Systolic,Diastolic);
```

Display the first five rows of each table.

```
head(T,5)
```

```
ans=5x3 table
  Age      Gender      Smoker
  ----      -
  38      'Male'      true
  43      'Male'      false
  38      'Female'    false
  40      'Female'    false
  49      'Female'    false
```

```
head(T2,5)
```

```
ans=5x4 table
  Height      Weight      Systolic      Diastolic
  ----      -
  71          176          124           93
  69          163          109           77
  64          131          125           83
  67          133          117           75
  64          119          122           80
```

Add Variables Concatenated from Another Table

Add variables to the table T by horizontally concatenating it with T2.

```
T = [T T2];
```

Display the first five rows of T.

```
head(T,5)
```

```
ans=5x7 table
  Age      Gender      Smoker      Height      Weight      Systolic      Diastolic
  ----      -
  38      'Male'      true        71          176          124           93
  43      'Male'      false       69          163          109           77
  38      'Female'    false       64          131          125           83
  40      'Female'    false       67          133          117           75
  49      'Female'    false       64          119          122           80
```

The table T now has 7 variables and 100 rows.

If the tables that you are horizontally concatenating have row names, `horzcat` concatenates the tables by matching the row names. Therefore, the tables must use the same row names, but the row order does not matter.

Add Variable from Workspace to Table

Add the names of patients from the workspace variable `LastName` before the first table variable in `T`. You can specify any location in the table using the name of a variable near the new location. Use quotation marks to refer to the names of table variables. However, do not use quotation marks for input arguments that are workspace variables.

```
T = addvars(T,LastName, 'Before', 'Age');
head(T,5)
```

```
ans=5x8 table
  LastName      Age      Gender      Smoker      Height      Weight      Systolic      Diastolic
  _____  _____  _____  _____  _____  _____  _____  _____
  'Smith'      38      'Male'      true         71         176         124          93
  'Johnson'    43      'Male'      false        69         163         109          77
  'Williams'   38      'Female'    false        64         131         125          83
  'Jones'      40      'Female'    false        67         133         117          75
  'Brown'      49      'Female'    false        64         119         122          80
```

You also can specify locations in a table using numbers. For example, the equivalent syntax using a number to specify location is `T = addvars(T,LastName, 'Before', 1)`.

Add Variables Using Dot Syntax

An alternative way to add new table variables is to use dot syntax. When you use dot syntax, you always add the new variable as the last table variable. You can add a variable that has any data type, as long as it has the same number of rows as the table.

Create a new variable for blood pressure as a horizontal concatenation of the two variables `Systolic` and `Diastolic`. Add it to `T`.

```
T.BloodPressure = [Systolic Diastolic];
head(T,5)
```

```
ans=5x9 table
  LastName      Age      Gender      Smoker      Height      Weight      Systolic      Diastolic
  _____  _____  _____  _____  _____  _____  _____  _____
  'Smith'      38      'Male'      true         71         176         124          93
  'Johnson'    43      'Male'      false        69         163         109          77
  'Williams'   38      'Female'    false        64         131         125          83
  'Jones'      40      'Female'    false        67         133         117          75
  'Brown'      49      'Female'    false        64         119         122          80
```

'Smith'	38	'Male'	true	71	176	124	93
'Johnson'	43	'Male'	false	69	163	109	77
'Williams'	38	'Female'	false	64	131	125	83
'Jones'	40	'Female'	false	67	133	117	75
'Brown'	49	'Female'	false	64	119	122	80

T now has 9 variables and 100 rows. A table variable can have multiple columns. So although `BloodPressure` has two columns, it is one table variable.

Add a new variable, `BMI`, in the table `T`, that contains the body mass index for each patient. `BMI` is a function of height and weight. When you calculate `BMI`, you can refer to the `Weight` and `Height` variables that are in `T`.

```
T.BMI = (T.Weight*0.453592)./(T.Height*0.0254).^2;
```

The operators `./` and `.^` in the calculation of `BMI` indicate element-wise division and exponentiation, respectively.

Display the first five rows of the table `T`.

```
head(T,5)
```

```
ans=5x10 table
```

LastName	Age	Gender	Smoker	Height	Weight	Systolic	Diastolic
'Smith'	38	'Male'	true	71	176	124	93
'Johnson'	43	'Male'	false	69	163	109	77
'Williams'	38	'Female'	false	64	131	125	83
'Jones'	40	'Female'	false	67	133	117	75
'Brown'	49	'Female'	false	64	119	122	80

Move Variable in Table

Move the table variable `BMI` using the `movevars` function, so that it is after the variable `Weight`. When you specify table variables by name, use quotation marks.

```
T = movevars(T, 'BMI', 'After', 'Weight');
head(T,5)
```

```
ans=5x10 table
```

LastName	Age	Gender	Smoker	Height	Weight	BMI	Systolic
----------	-----	--------	--------	--------	--------	-----	----------

'Smith'	38	'Male'	true	71	176	24.547	124
'Johnson'	43	'Male'	false	69	163	24.071	109
'Williams'	38	'Female'	false	64	131	22.486	125
'Jones'	40	'Female'	false	67	133	20.831	117
'Brown'	49	'Female'	false	64	119	20.426	122

You also can specify locations in a table using numbers. For example, the equivalent syntax using a number to specify location is `T = movevars(T, 'BMI', 'After', 6)`. It is often more convenient to refer to variables by name.

Move Table Variable Using Indexing

As an alternative, you can move table variables by indexing. You can index into a table using the same syntax you use for indexing into a matrix.

Move `BloodPressure` so that it is next to `BMI`.

```
T = T(:, [1:7 10 8 9]);
head(T, 5)
```

```
ans=5x10 table
  LastName    Age    Gender    Smoker    Height    Weight    BMI    BloodPres
  _____  ___  _____  _____  _____  _____  _____  _____
  'Smith'     38    'Male'     true       71        176       24.547    124
  'Johnson'  43    'Male'     false      69        163       24.071    109
  'Williams'  38    'Female'   false      64        131       22.486    125
  'Jones'     40    'Female'   false      67        133       20.831    117
  'Brown'     49    'Female'   false      64        119       20.426    122
```

In a table with many variables, it is often more convenient to use the `movevars` function.

Delete Variables

To delete table variables, use the `removevars` function. Delete the `Systolic` and `Diastolic` table variables.

```
T = removevars(T, {'Systolic', 'Diastolic'});
head(T, 5)
```

```
ans=5x8 table
  LastName    Age    Gender    Smoker    Height    Weight    BMI    BloodPres
```

'Smith'	38	'Male'	true	71	176	24.547	124	93
'Johnson'	43	'Male'	false	69	163	24.071	109	77
'Williams'	38	'Female'	false	64	131	22.486	125	83
'Jones'	40	'Female'	false	67	133	20.831	117	75
'Brown'	49	'Female'	false	64	119	20.426	122	80

Delete Variable Using Dot Syntax

As an alternative, you can delete variables using dot syntax and the empty matrix, []. Remove the Age variable from the table.

```
T.Age = [];
head(T,5)
```

```
ans=5x7 table
```

LastName	Gender	Smoker	Height	Weight	BMI	BloodPressure	
'Smith'	'Male'	true	71	176	24.547	124	93
'Johnson'	'Male'	false	69	163	24.071	109	77
'Williams'	'Female'	false	64	131	22.486	125	83
'Jones'	'Female'	false	67	133	20.831	117	75
'Brown'	'Female'	false	64	119	20.426	122	80

Delete Variable Using Indexing

You also can delete variables using indexing and the empty matrix, []. Remove the Gender variable from the table.

```
T(:, 'Gender') = [];
head(T,5)
```

```
ans=5x6 table
```

LastName	Smoker	Height	Weight	BMI	BloodPressure	
'Smith'	true	71	176	24.547	124	93
'Johnson'	false	69	163	24.071	109	77
'Williams'	false	64	131	22.486	125	83
'Jones'	false	67	133	20.831	117	75
'Brown'	false	64	119	20.426	122	80

Split and Merge Table Variables

To split multicolumn table variables into variables that each have one column, use the `splitvars` functions. Split the variable `BloodPressure` into two variables.

```
T = splitvars(T, 'BloodPressure', 'NewVariableNames', {'Systolic', 'Diastolic'});
head(T,5)
```

```
ans=5x7 table
  LastName   Smoker   Height   Weight   BMI   Systolic   Diastolic
  _____  _____  _____  _____  _____  _____  _____
  'Smith'    true      71       176      24.547  124       93
  'Johnson' false     69       163      24.071  109       77
  'Williams' false     64       131      22.486  125       83
  'Jones'    false     67       133      20.831  117       75
  'Brown'    false     64       119      20.426  122       80
```

Similarly, you can group related table variables together in one variable, using the `mergevars` function. Combine `Systolic` and `Diastolic` back into one variable, and name it `BP`.

```
T = mergevars(T, {'Systolic', 'Diastolic'}, 'NewVariableName', 'BP');
head(T,5)
```

```
ans=5x6 table
  LastName   Smoker   Height   Weight   BMI   BP
  _____  _____  _____  _____  _____  _____
  'Smith'    true      71       176      24.547  124  93
  'Johnson' false     69       163      24.071  109  77
  'Williams' false     64       131      22.486  125  83
  'Jones'    false     67       133      20.831  117  75
  'Brown'    false     64       119      20.426  122  80
```

Reorient Rows To Become Variables

You can reorient the rows of a table or timetable, so that they become the variables the output table, using the `rows2vars` function. However, if the table has multicolumn variables, then you must split them before you can call `rows2vars`.

Reorient the rows of T. Specify that the names of the patients in T are the names of table variables in the output table. The first variable of T3 contains the names of the variables of T. Each remaining variable of T3 contains the data from the corresponding row of T.

```
T = splitvars(T, 'BP', 'NewVariableNames', {'Systolic', 'Diastolic'});
T3 = rows2vars(T, 'VariableNamesSource', 'LastName');
T3(:,1:5)
```

```
ans=6x5 table
   OriginalVariableNames   Smith   Johnson   Williams   Jones
   _____   _____   _____   _____   _____
   'Smoker'           1         0         0         0
   'Height'          71         69         64         67
   'Weight'         176        163        131        133
   'BMI'            24.547    24.071    22.486    20.831
   'Systolic'       124        109        125        117
   'Diastolic'      93         77         83         75
```

You can use dot syntax with T3 to access patient data as an array. However, if the row values of an input table cannot be concatenated, then the variables of the output table are cell arrays.

```
T3.Smith
```

```
ans = 6x1
    1.0000
   71.0000
  176.0000
   24.5467
  124.0000
   93.0000
```

See Also

addvars | inner2outer | mergevars | movevars | removevars | rows2vars | splitvars | table

Related Examples

- “Add and Delete Table Rows” on page 9-13
- “Clean Messy and Missing Data in Tables” on page 9-26
- “Modify Units, Descriptions, and Table Variable Names” on page 9-33

Clean Messy and Missing Data in Tables

This example shows how to find, clean, and delete table rows with missing data.

Load Sample Data

Load sample data from a comma-separated text file, `messy.csv`. The file contains many different missing data indicators:

- Empty character vector ('')
- period (.)
- NA
- NaN
- -99

To specify the character vectors to treat as empty values, use the `'TreatAsEmpty'` name-value pair argument with the `readtable` function.

```
T = readtable('messy.csv', 'TreatAsEmpty', {'.', 'NA'})
```

T=21x5 table

A	B	C	D	E
'afe1'	3	'yes'	3	3
'egh3'	NaN	'no'	7	7
'wth4'	3	'yes'	3	3
'atn2'	23	'no'	23	23
'arg1'	5	'yes'	5	5
'jre3'	34.6	'yes'	34.6	34.6
'wen9'	234	'yes'	234	234
'ple2'	2	'no'	2	2
'dbo8'	5	'no'	5	5
'oii4'	5	'yes'	5	5
'wnk3'	245	'yes'	245	245
'abk6'	563	''	563	563
'pnj5'	463	'no'	463	463
'wnn3'	6	'no'	6	6
'oks9'	23	'yes'	23	23
'wba3'	NaN	'yes'	NaN	14
⋮				

T is a table with 21 rows and five variables. 'TreatAsEmpty' only applies to numeric columns in the file and cannot handle numeric literals, such as '-99'.

Summarize Table

View the data type, description, units, and other descriptive statistics for each variable by creating a table summary using the `summary` function.

```
summary(T)
```

```
Variables:
```

```
A: 21x1 cell array of character vectors
```

```
B: 21x1 double
```

```
Values:
```

Min	-99
Median	14
Max	563
NumMissing	3

```
C: 21x1 cell array of character vectors
```

```
D: 21x1 double
```

```
Values:
```

Min	-99
Median	7
Max	563
NumMissing	2

```
E: 21x1 double
```

```
Values:
```

Min	-99
Median	14
Max	563

When you import data from a file, the default is for `readtable` to read any variables with nonnumeric elements as a cell array of character vectors.

Find Rows with Missing Values

Display the subset of rows from the table, T, that have at least one missing value.

```
TF = ismissing(T,{' ' '.' 'NA' NaN -99});
T(any(TF,2),:)
```

ans=5x5 table

A	B	C	D	E
'egh3'	NaN	'no'	7	7
'abk6'	563	' '	563	563
'wba3'	NaN	'yes'	NaN	14
'poj2'	-99	'yes'	-99	-99
'gry5'	NaN	'yes'	NaN	21

readtable replaced '.' and 'NA' with NaN in the numeric variables, B, D, and E.

Replace Missing Value Indicators

Clean the data so that the missing values indicated by code -99 have the standard MATLAB® numeric missing value indicator, NaN.

```
T = standardizeMissing(T,-99)
```

T=21x5 table

A	B	C	D	E
'afe1'	3	'yes'	3	3
'egh3'	NaN	'no'	7	7
'wth4'	3	'yes'	3	3
'atn2'	23	'no'	23	23
'arg1'	5	'yes'	5	5
'jre3'	34.6	'yes'	34.6	34.6
'wen9'	234	'yes'	234	234
'ple2'	2	'no'	2	2
'dbo8'	5	'no'	5	5
'oii4'	5	'yes'	5	5
'wnk3'	245	'yes'	245	245
'abk6'	563	' '	563	563
'pnj5'	463	'no'	463	463
'wnn3'	6	'no'	6	6

```
'oks9'    23    'yes'    23    23
'wba3'    NaN   'yes'    NaN   14
:
```

`standardizeMissing` replaces three instances of `-99` with `NaN`.

Create a new table, `T2`, and replace missing values with values from previous rows of the table. `fillmissing` provides a number of ways to fill in missing values.

```
T2 = fillmissing(T, 'previous')
```

`T2=21x5 table`

A	B	C	D	E
'afe1'	3	'yes'	3	3
'egh3'	3	'no'	7	7
'wth4'	3	'yes'	3	3
'atn2'	23	'no'	23	23
'arg1'	5	'yes'	5	5
'jre3'	34.6	'yes'	34.6	34.6
'wen9'	234	'yes'	234	234
'ple2'	2	'no'	2	2
'dbo8'	5	'no'	5	5
'oii4'	5	'yes'	5	5
'wnk3'	245	'yes'	245	245
'abk6'	563	'yes'	563	563
'pnj5'	463	'no'	463	463
'wnn3'	6	'no'	6	6
'oks9'	23	'yes'	23	23
'wba3'	23	'yes'	23	14
:				

Remove Rows with Missing Values

Create a new table, `T3`, that contains only the rows from `T` without missing values.

```
T3 = rmmissing(T)
```

`T3=16x5 table`

A	B	C	D	E

'afe1'	3	'yes'	3	3
'wth4'	3	'yes'	3	3
'atn2'	23	'no'	23	23
'arg1'	5	'yes'	5	5
'jre3'	34.6	'yes'	34.6	34.6
'wen9'	234	'yes'	234	234
'ple2'	2	'no'	2	2
'dbo8'	5	'no'	5	5
'oii4'	5	'yes'	5	5
'wnk3'	245	'yes'	245	245
'pnj5'	463	'no'	463	463
'wnn3'	6	'no'	6	6
'oks9'	23	'yes'	23	23
'pkn4'	2	'no'	2	2
'adw3'	22	'no'	22	22
'bas8'	23	'no'	23	23

T3 contains 16 rows and five variables.

Organize Data

Sort the rows of T3 in descending order by C, and then sort in ascending order by A.

```
T3 = sortrows(T2, {'C', 'A'}, {'descend', 'ascend'})
```

T3=21×5 table

A	B	C	D	E
'abk6'	563	'yes'	563	563
'afe1'	3	'yes'	3	3
'arg1'	5	'yes'	5	5
'gry5'	23	'yes'	23	21
'jre3'	34.6	'yes'	34.6	34.6
'oii4'	5	'yes'	5	5
'oks9'	23	'yes'	23	23
'poj2'	22	'yes'	22	22
'wba3'	23	'yes'	23	14
'wen9'	234	'yes'	234	234
'wnk3'	245	'yes'	245	245
'wth4'	3	'yes'	3	3
'adw3'	22	'no'	22	22
'atn2'	23	'no'	23	23
'bas8'	23	'no'	23	23

```
'dbo8'    5    'no'    5    5
  ⋮
```

In C, the rows are grouped first by 'yes', followed by 'no'. Then in A, the rows are listed alphabetically.

Reorder the table so that A and C are next to each other.

```
T3 = T3(:, {'A', 'C', 'B', 'D', 'E'})
```

T3=21x5 table

A	C	B	D	E
'abk6'	'yes'	563	563	563
'afe1'	'yes'	3	3	3
'arg1'	'yes'	5	5	5
'gry5'	'yes'	23	23	21
'jre3'	'yes'	34.6	34.6	34.6
'oii4'	'yes'	5	5	5
'oks9'	'yes'	23	23	23
'poj2'	'yes'	22	22	22
'wba3'	'yes'	23	23	14
'wen9'	'yes'	234	234	234
'wnk3'	'yes'	245	245	245
'wth4'	'yes'	3	3	3
'adw3'	'no'	22	22	22
'atn2'	'no'	23	23	23
'bas8'	'no'	23	23	23
'dbo8'	'no'	5	5	5
⋮				

See Also

fillmissing | ismissing | readtable | rmmissing | sortrows | standardizeMissing | summary

Related Examples

- “Add and Delete Table Rows” on page 9-13

- “Add, Delete, and Rearrange Table Variables” on page 9-17
- “Modify Units, Descriptions, and Table Variable Names” on page 9-33
- “Access Data in a Table” on page 9-42
- “Missing Data in MATLAB”

Modify Units, Descriptions, and Table Variable Names

This example shows how to access and modify table properties for variable units, descriptions and names. You also can edit these property values using the Variables Editor.

Load Sample Data

Load the sample patients data and create a table.

```
load patients
BloodPressure = [Systolic Diastolic];

T = table(Gender, Age, Height, Weight, Smoker, BloodPressure);
```

Display the first five rows of the table, T.

```
T(1:5, :)
```

```
ans=5x6 table
   Gender   Age   Height   Weight   Smoker   BloodPressure
   _____   _____   _____   _____   _____   _____
   'Male'    38     71     176     true     124     93
   'Male'    43     69     163     false    109     77
   'Female'  38     64     131     false    125     83
   'Female'  40     67     133     false    117     75
   'Female'  49     64     119     false    122     80
```

T has 100 rows and 6 variables.

Add Variable Units

Specify units for each variable in the table by modifying the table property, `VariableUnits`. Specify the variable units as a cell array of character vectors.

```
T.Properties.VariableUnits = {'' 'Yrs' 'In' 'Lbs' '' ''};
```

An individual empty character vector within the cell array indicates that the corresponding variable does not have units.

Add a Variable Description for a Single Variable

Add a variable description for the variable, `BloodPressure`. Assign a single character vector to the element of the cell array containing the description for `BloodPressure`.

```
T.Properties.VariableDescriptions{'BloodPressure'} = 'Systolic/Diastolic';
```

You can use the variable name, 'BloodPressure', or the numeric index of the variable, 6, to index into the cell array of character vectors containing the variable descriptions.

Summarize the Table

View the data type, description, units, and other descriptive statistics for each variable by using `summary` to summarize the table.

```
summary(T)
```

```
Variables:
```

```
Gender: 100x1 cell array of character vectors
```

```
Age: 100x1 double
```

```
Properties:
```

```
Units: Yrs
```

```
Values:
```

```
Min      25
```

```
Median   39
```

```
Max      50
```

```
Height: 100x1 double
```

```
Properties:
```

```
Units: In
```

```
Values:
```

```
Min      60
```

```
Median   67
```

```
Max      72
```

```
Weight: 100x1 double
```

```
Properties:
```

```
Units: Lbs
```

```
Values:
```

```
Min      111
```

```
Median   142.5
```

```
Max      202
```

```
Smoker: 100x1 logical
```

```
Values:
```

```
True     34
```

```
False    66
```

```
BloodPressure: 100x2 double
```

```
Properties:
```

```
Description: Systolic/Diastolic
```

```
Values:
```

```
BloodPressure_1
```

```
BloodPressure_2
```

Min	109	68
Median	122	81.5
Max	138	99

The BloodPressure variable has a description and the Age, Height, Weight, and BloodPressure variables have units.

Change a Variable Name

Change the variable name for the first variable from Gender to Sex.

```
T.Properties.VariableNames{'Gender'} = 'Sex';
```

Display the first five rows of the table, T.

```
T(1:5, :)
```

```
ans=5x6 table
```

Sex	Age	Height	Weight	Smoker	BloodPressure	
'Male'	38	71	176	true	124	93
'Male'	43	69	163	false	109	77
'Female'	38	64	131	false	125	83
'Female'	40	67	133	false	117	75
'Female'	49	64	119	false	122	80

In addition to properties for variable units, descriptions and names, there are table properties for row and dimension names, a table description, and user data.

See Also

[array2table](#) | [cell2table](#) | [readtable](#) | [struct2table](#) | [summary](#) | [table](#)

Related Examples

- “Add, Delete, and Rearrange Table Variables” on page 9-17
- “Access Data in a Table” on page 9-42

Add Custom Properties to Tables and Timetables

This example shows how to add custom properties to tables and timetables, set and access their values, and remove them.

All tables and timetables have properties that contain metadata about them or their variables. You can access these properties through the `T.Properties` object, where `T` is the name of the table or timetable. For example, `T.Properties.VariableNames` returns a cell array containing the names of the variables of `T`.

The properties you access through `T.Properties` are part of the definitions of the `table` and `timetable` data types. You cannot add or remove these predefined properties. But starting in R2018b, you can add and remove your own *custom* properties, by modifying the `T.Properties.CustomProperties` object of a table or timetable.

Add Properties

Read power outage data into a table. Sort it using the first variable that contains dates and times, `OutageTime`. Then display the first three rows.

```
T = readtable('outages.csv');
T = sortrows(T, 'OutageTime');
head(T,3)
```

```
ans=3×6 table
      Region      OutageTime      Loss      Customers      RestorationTime
      _____  _____  _____  _____  _____
      'SouthWest'  2002-02-01 12:18  458.98  1.8202e+06  2002-02-07 16:50
      'MidWest'    2002-03-05 17:53  96.563  2.8666e+05  2002-03-10 14:41
      'MidWest'    2002-03-16 06:18  186.44  2.1275e+05  2002-03-18 23:23
```

Display its properties. These are the properties that all tables have in common. Note that there is also a `CustomProperties` object, but that by default it has no properties.

`T.Properties`

```
ans =
      TableProperties with properties:

      Description: ''
      UserData: []
```

```

        DimensionNames: {'Row' 'Variables'}
        VariableNames: {1x6 cell}
VariableDescriptions: {}
        VariableUnits: {}
        VariableContinuity: []
        RowNames: {}
    CustomProperties: No custom properties are set.
    Use addprop and rmprop to modify CustomProperties.

```

To add custom properties, use the `addprop` function. Specify the names of the properties. For each property, also specify whether it has metadata for the whole table (similar to the `Description` property) or for its variables (similar to the `VariableNames` property). If the property has variable metadata, then its value must be a vector whose length is equal to the number of variables.

Add custom properties that contain an output file name, file type, and indicators of which variables to plot. Best practice is to assign the input table as the output argument of `addprop`, so that the custom properties are part of the same table. Specify that the output file name and file type are table metadata using the `'table'` option. Specify that the plot indicators are variable metadata using the `'variable'` option.

```

T = addprop(T,{'OutputFileName','OutputFileType','ToPlot'}, ...
            {'table','table','variable'});

```

`T.Properties`

`ans =`

TableProperties with properties:

```

        Description: ''
        UserData: []
        DimensionNames: {'Row' 'Variables'}
        VariableNames: {1x6 cell}
VariableDescriptions: {}
        VariableUnits: {}
        VariableContinuity: []
        RowNames: {}

```

```

Custom Properties (access using t.Properties.CustomProperties.<name>):
    OutputFileName: []
    OutputFileType: []
        ToPlot: []

```

Set and Access Values of Custom Properties

When you add custom properties using `addprop`, their values are empty arrays by default. You can set and access the values of the custom properties using dot syntax.

Set the output file name and type. These properties contain metadata for the table. Then assign a logical array to the `ToPlot` property. This property contains metadata for the variables. In this example, the elements of the value of the `ToPlot` property are `true` for each variable to be included in a plot, and `false` for each variable to be excluded.

```
T.Properties.CustomProperties.OutputFileName = 'outageResults';
T.Properties.CustomProperties.OutputFileType = '.mat';
T.Properties.CustomProperties.ToPlot = [false false true true true false];
T.Properties
```

```
ans =
```

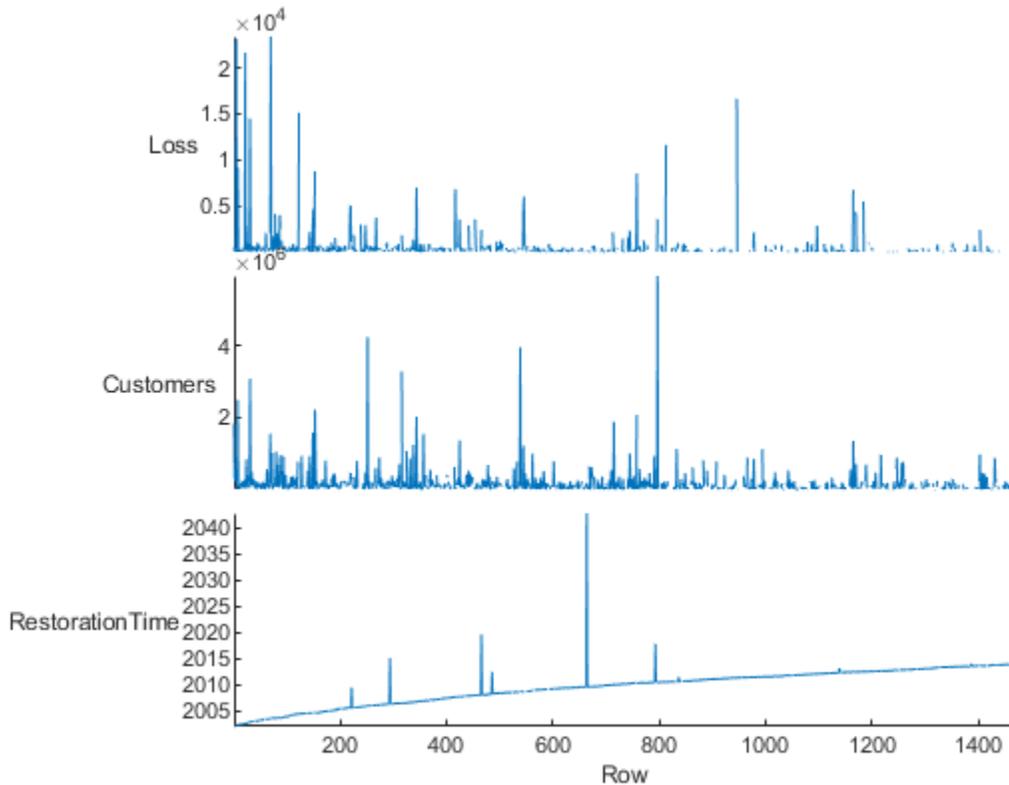
```
TableProperties with properties:
```

```
    Description: ''
      UserData: []
DimensionNames: {'Row' 'Variables'}
VariableNames: {1x6 cell}
VariableDescriptions: {}
  VariableUnits: {}
VariableContinuity: []
      RowNames: {}
```

```
Custom Properties (access using t.Properties.CustomProperties.<name>):
  OutputFileName: 'outageResults'
  OutputFileType: '.mat'
      ToPlot: [0 0 1 1 1 0]
```

Plot variables from `T` in a stacked plot using the `stackedplot` function. To plot only the `Loss`, `Customers`, and `RestorationTime` values, use the `ToPlot` custom property as the second input argument.

```
stackedplot(T,T.Properties.CustomProperties.ToPlot);
```



When you move or delete table variables, both the predefined and custom properties are reordered so that their values correspond to the same variables. In this example, the values of the `ToPlot` custom property stay aligned with the variables marked for plotting, just as the values of the `VariableNames` predefined property stay aligned.

Remove the `Customers` variable and display the properties.

```
T.Customers = [];
T.Properties
```

```
ans =
  TableProperties with properties:
```

```
  Description: ''
```

```
        UserData: []
        DimensionNames: {'Row' 'Variables'}
        VariableNames: {1x5 cell}
VariableDescriptions: {}
        VariableUnits: {}
VariableContinuity: []
        RowNames: {}

Custom Properties (access using t.Properties.CustomProperties.<name>):
    OutputFileName: 'outageResults'
    OutputFileType: '.mat'
        ToPlot: [0 0 1 1 0]
```

Convert the table to a timetable, using the outage times as row times. Move `Region` to the end of the table, and `RestorationTime` before the first variable, using the `movevars` function. Note that the properties are reordered appropriately. The `RestorationTime` and `Loss` variables still have indicators for inclusion in a plot.

```
T = table2timetable(T);
T = movevars(T, 'Region', 'After', 'Cause');
T = movevars(T, 'RestorationTime', 'Before', 1);
T.Properties

ans =
    TimetableProperties with properties:

        Description: ''
        UserData: []
        DimensionNames: {'OutageTime' 'Variables'}
        VariableNames: {'RestorationTime' 'Loss' 'Cause' 'Region'}
VariableDescriptions: {}
        VariableUnits: {}
VariableContinuity: []
        RowTimes: [1468x1 datetime]
        StartTime: 2002-02-01 12:18
        SampleRate: NaN
        TimeStep: NaN

Custom Properties (access using t.Properties.CustomProperties.<name>):
    OutputFileName: 'outageResults'
    OutputFileType: '.mat'
        ToPlot: [1 1 0 0]
```

Remove Properties

You can remove any or all of the custom properties of a table using the `rmprop` function. However, you cannot use it to remove predefined properties from `T.Properties`, because those properties are part of the definition of the table data type.

Remove the `OutputFileName` and `OutputFileType` custom properties. Display the remaining table properties.

```
T = rmprop(T,{'OutputFileName','OutputFileType'});
T.Properties
```

```
ans =
  TimetableProperties with properties:

    Description: ''
    UserData: []
    DimensionNames: {'OutageTime' 'Variables'}
    VariableNames: {'RestorationTime' 'Loss' 'Cause' 'Region'}
    VariableDescriptions: {}
    VariableUnits: {}
    VariableContinuity: []
    RowTimes: [1468x1 datetime]
    StartTime: 2002-02-01 12:18
    SampleRate: NaN
    TimeStep: NaN

    Custom Properties (access using t.Properties.CustomProperties.<name>):
        ToPlot: [1 1 0 0]
```

See Also

`addprop` | `head` | `movevars` | `readtable` | `rmprop` | `sortrows` | `stackedplot` | `table` | `table2timetable`

Related Examples

- “Modify Units, Descriptions, and Table Variable Names” on page 9-33
- “Access Data in a Table” on page 9-42
- “Add, Delete, and Rearrange Table Variables” on page 9-17

Access Data in a Table

In this section...

“Ways to Index into a Table” on page 9-42

“Create Table from Subset of Larger Table” on page 9-44

“Create Array from the Contents of Table” on page 9-47

Ways to Index into a Table

A table is a container for storing column-oriented variables that have the same number of rows. Parentheses allow you to select a subset of the data in a table and preserve the table container. Curly braces and dot indexing allow you to extract data from a table.

If you use curly braces, the resulting array is the horizontal concatenation of the specified table variables containing only the specified rows. The data types of all the specified variables must be compatible for concatenation. You can then perform calculations using MATLAB functions.

Dot indexing extracts data from one table variable. The result is an array of the same data type as extracted variable. You can follow the dot indexing with parentheses to specify a subset of rows to extract from a variable.

`T.Variables` horizontally concatenates all table variables into an array. `T.Variables` is equivalent to `T{: , :}`.

To subscript into a table and select variables of a specified type, use the `vartype` function.

Summary of Table Indexing Methods

Consider a table, `T`.

Type of Indexing	Result	Syntax	Rows	Variables
Parentheses	table	<code>T(rows, vars)</code>	One or more rows, specified by <i>rows</i>	One or more variables, specified by <i>vars</i>
Curly Braces	extracted data	<code>T{rows, vars}</code>	One or more rows, specified by <i>rows</i>	One or more variables, specified by <i>vars</i>

Type of Indexing	Result	Syntax	Rows	Variables
Dot Indexing	extracted data	<code>T.var</code> <code>T.</code> <code>(varindex)</code>	All rows	One variable, specified by <i>var</i> (a name) or by <i>varindex</i> (a number)
Dot Indexing	extracted data	<code>T.var(rows)</code>	One or more rows, specified by <i>rows</i>	One variable, specified by <i>var</i> (a name)
Variables Property	extracted data	<code>T.Variables</code>	All rows	All variables when they can be horizontally concatenated into an array
Subscripting by Variable Type	table	<code>S = vartype(type);</code> <code>T(rows,S)</code>	One or more rows, specified by <i>rows</i>	One or more variables of the specified <i>type</i> (for example, 'numeric')
Subscripting by Variable Type	extracted data	<code>S = vartype(type);</code> <code>T{rows,S}</code>	One or more rows, specified by <i>rows</i>	One or more variables of the specified <i>type</i> (for example, 'numeric')

How to Specify Rows to Access

When indexing into a table with parentheses, curly braces, or dot indexing, you can specify *rows* as a colon, numeric indices, or logical expressions. Furthermore, you can index by name using a single row name or a cell array of row names.

A logical expression can contain curly braces or dot indexing to extract data from which you can define the subset of rows. For example, `rows = T.Var2>0` returns a logical array with logical `true` (1) for rows where the value in the variable `Var2` is greater than zero.

How to Specify Variables to Access

When indexing into a table with parentheses or curly braces, you can specify *vars* as a colon, numeric indices, logical expressions, a single variable name, a cell array of variable names, or as the output of the `vartype` function..

When using dot indexing, you must specify a single variable to access. For a single variable name, use `T.var`. For a single variable index, specified as a positive integer, use `T.(varindex)`.

Create Table from Subset of Larger Table

This example shows how to create a table from a subset of a larger table.

Load Sample Data

Load the sample patients data and create a table. Use the unique identifiers in `LastName` as row names.

```
load patients

patients = table(Age,Gender,Height,Weight,Smoker,...
    'RowNames',LastName);
```

The table, `patients`, contains 100 rows and 5 variables.

View the data type, description, units, and other descriptive statistics for each variable by using `summary` to summarize the table.

```
summary(patients)
```

Variables:

Age: 100x1 double

Values:

Min	25
Median	39
Max	50

Gender: 100x1 cell array of character vectors

Height: 100x1 double

Values:

Min	60
Median	67
Max	72

Weight: 100x1 double

Values:

Min	111
Median	142.5
Max	202

Smoker: 100x1 logical

Values:

True	34
False	66

Index Using Numeric Indices

Create a subtable containing the first five rows and all the variables from the table, `patients`. Use numeric indexing within the parentheses to specify the desired rows and variables. This is similar to indexing with numeric arrays.

```
T1 = patients(1:5,:)
```

T1=5x5 table

	Age	Gender	Height	Weight	Smoker
Smith	38	'Male'	71	176	true
Johnson	43	'Male'	69	163	false
Williams	38	'Female'	64	131	false
Jones	40	'Female'	67	133	false
Brown	49	'Female'	64	119	false

`T1` is a 5-by-5 table. In addition to numeric indices, you can use row or variable names inside the parentheses. In this case, using row indices and a colon is more compact than using row or variable names.

Index Using Names

Select all the data for the patients with the last names 'Adams' and 'Brown'. In this case, it is simpler to use the row names than to use the numeric index.

```
T2 = patients({'Adams', 'Brown'},:)
```

T2=2×5 table

	Age	Gender	Height	Weight	Smoker
Adams	48	'Female'	66	137	false
Brown	49	'Female'	64	119	false

T2 is a 2-by-5 table.

Index Using a Logical Expression

Create a new table, T3, containing the gender, height, and weight of the patients under the age of 30. Select only the rows where the value in the variable, *Age*, is less than 30.

Use dot notation to extract data from a table variable and a logical expression to define the subset of rows based on that extracted data.

```
rows = patients.Age<30;
vars = {'Gender', 'Height', 'Weight'};
```

rows is a 100-by-1 logical array containing logical `true` (1) for rows where the value in the variable, *Age*, is less than 30.

Use parentheses to return a table containing the desired subset of the data.

```
T3 = patients(rows,vars)
```

T3=15×3 table

	Gender	Height	Weight
Moore	'Male'	68	183
Jackson	'Male'	71	174
Garcia	'Female'	69	131
Walker	'Female'	65	123
Hall	'Male'	70	189
Young	'Female'	63	114
Hill	'Female'	64	138
Rivera	'Female'	63	130
Cooper	'Female'	65	127
Cox	'Female'	66	111
Howard	'Female'	68	134
James	'Male'	66	186
Jenkins	'Male'	69	189

Perry	'Female'	64	120
Alexander	'Male'	69	171

T3 is a 15-by-3 table.

Create Array from the Contents of Table

This example shows how to extract the contents of a table using curly braces or dot indexing.

Load Sample Data

Load the sample patients data and create a table. Use the unique identifiers in `LastName` as row names.

```
load patients
```

```
patients = table(Age,Gender,Height,Weight,Smoker,...
    'RowNames',LastName);
```

The table, `patients`, contains 100 rows and 5 variables.

Extract Multiple Rows and Multiple Variables

Extract data from multiple variables in the table, `patients` by using curly braces. Since dot indexing extracts data from a single variable at a time, braces are more convenient when you want to extract more than one variable.

Extract the height and weight for the first five patients. Use numeric indices to select the subset of rows, `1:5`, and variable names to select the subset of variables, `{Height,Weight}`.

```
A = patients{1:5,{'Height','Weight'}}
```

```
A = 5x2
```

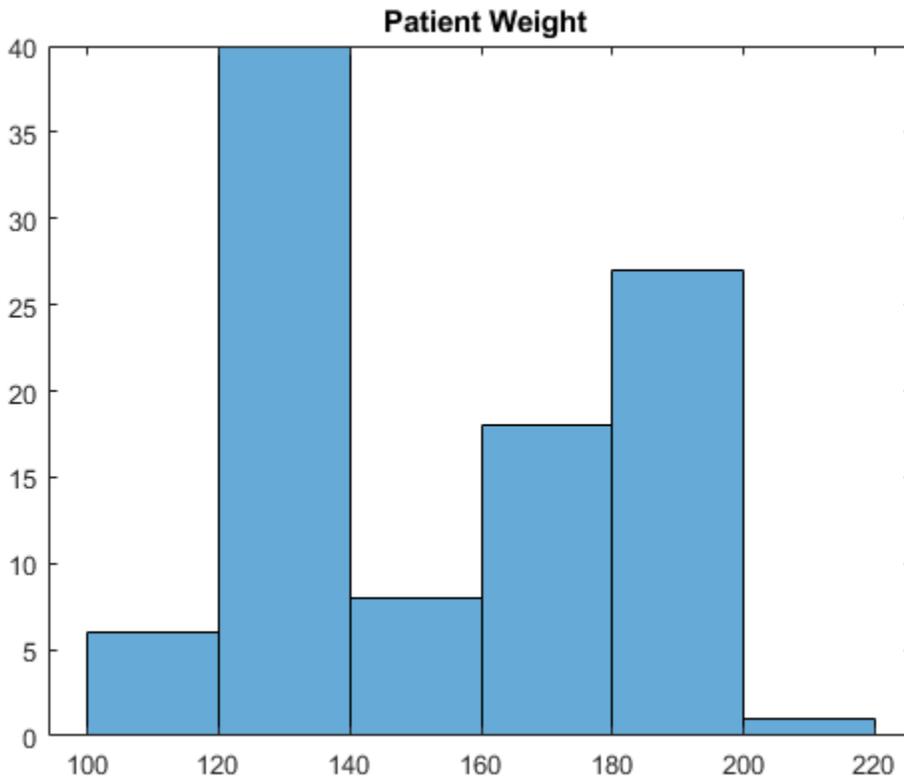
71	176
69	163
64	131
67	133
64	119

A is a 5-by-2 numeric array.

Extract Data from One Variable

Use dot indexing to easily extract the contents of a single variable. Plot a histogram of the numeric data in the variable, `Weight`.

```
figure()  
histogram(patients.Weight)  
title(' Patient Weight')
```



`patients.Weight` is a double-precision column vector with 100 rows. Alternatively, you can use curly braces, `patients{:,'Weight'}`, to extract all the rows for the variable `Weight`.

To specify a subset of rows for a single variable, you can follow the dot indexing with parentheses or curly braces. Extract the heights of the nonsmoker patients under the age of 30.

Use dot notation to extract data from table variables and a logical expression to define the subset of rows based on that extracted data.

```
rows = patients.Smoker==false & patients.Age<30;
```

Use dot notation to extract the desired rows from the variable, Height.

```
patients.Height(rows)
```

```
ans = 11x1
```

```
68
71
70
63
64
63
65
66
68
66
:
```

The output is a 11-by-1 numeric array. Alternatively, you can specify the single variable, Height, within curly braces to extract the desired data, `patients{rows, 'Height'}`.

See Also

`histogram` | `summary` | `table`

Related Examples

- “Create and Work with Tables” on page 9-2
- “Modify Units, Descriptions, and Table Variable Names” on page 9-33
- “Calculations on Tables” on page 9-51

More About

- “Advantages of Using Tables” on page 9-64

Calculations on Tables

This example shows how to perform calculations on tables.

The functions `rowfun` and `varfun` each apply a specified function to a table, yet many other functions require numeric or homogeneous arrays as input arguments. You can extract data from individual variables using dot indexing or from one or more variables using curly braces. The extracted data is then an array that you can use as input to other functions.

Read Sample Data into Table

Read data from a comma-separated text file, `testScores.csv`, into a table using the `readtable` function. `testScores.csv` contains test scores for several students. Use the student names in the first column of the text file as row names in the table.

```
T = readtable('testScores.csv', 'ReadRowNames', true)
```

T=10×4 table

	Gender	Test1	Test2	Test3
HOWARD	'male'	90	87	93
WARD	'male'	87	85	83
TORRES	'male'	86	85	88
PETERSON	'female'	75	80	72
GRAY	'female'	89	86	87
RAMIREZ	'female'	96	92	98
JAMES	'male'	78	75	77
WATSON	'female'	91	94	92
BROOKS	'female'	86	83	85
KELLY	'male'	79	76	82

T is a table with 10 rows and four variables.

Summarize the Table

View the data type, description, units, and other descriptive statistics for each variable by using the `summary` function to summarize the table.

```
summary(T)
```

Variables:

Gender: 10x1 cell array of character vectors

Test1: 10x1 double

Values:

```

      Min      75
     Median  86.5
      Max     96

```

Test2: 10x1 double

Values:

```

      Min      75
     Median  85
      Max     94

```

Test3: 10x1 double

Values:

```

      Min      72
     Median  86
      Max     98

```

The summary contains the minimum, median, and maximum score for each test.

Find the Average Across Each Row

Extract the data from the second, third, and fourth variables using curly braces, {}, find the average of each row, and store it in a new variable, TestAvg.

```
T.TestAvg = mean(T{: ,2:end},2)
```

T=10x5 table

	Gender	Test1	Test2	Test3	TestAvg
HOWARD	'male'	90	87	93	90
WARD	'male'	87	85	83	85
TORRES	'male'	86	85	88	86.333
PETERSON	'female'	75	80	72	75.667
GRAY	'female'	89	86	87	87.333
RAMIREZ	'female'	96	92	98	95.333

JAMES	'male'	78	75	77	76.667
WATSON	'female'	91	94	92	92.333
BROOKS	'female'	86	83	85	84.667
KELLY	'male'	79	76	82	79

Alternatively, you can use the variable names, `T{:, {'Test1', 'Test2', 'Test3'}}` or the variable indices, `T{:, 2:4}` to select the subset of data.

Compute Statistics Using Grouping Variable

Compute the mean and maximum of `TestAvg` by gender of the students.

```
varfun(@mean, T, 'InputVariables', 'TestAvg', ...
      'GroupingVariables', 'Gender')
```

`ans=2x3 table`

Gender	GroupCount	mean_TestAvg
'female'	5	87.067
'male'	5	83.4

Replace Data Values

The maximum score for each test is 100. Use curly braces to extract the data from the table and convert the test scores to a 25 point scale.

```
T{:, 2:end} = T{:, 2:end} * 25 / 100
```

`T=10x5 table`

	Gender	Test1	Test2	Test3	TestAvg
HOWARD	'male'	22.5	21.75	23.25	22.5
WARD	'male'	21.75	21.25	20.75	21.25
TORRES	'male'	21.5	21.25	22	21.583
PETERSON	'female'	18.75	20	18	18.917
GRAY	'female'	22.25	21.5	21.75	21.833
RAMIREZ	'female'	24	23	24.5	23.833
JAMES	'male'	19.5	18.75	19.25	19.167
WATSON	'female'	22.75	23.5	23	23.083
BROOKS	'female'	21.5	20.75	21.25	21.167
KELLY	'male'	19.75	19	20.5	19.75

Change Variable Name

Change the variable name from `TestAvg` to `Final`.

```
T.Properties.VariableNames{end} = 'Final'
```

T=10x5 table

	Gender	Test1	Test2	Test3	Final
HOWARD	'male'	22.5	21.75	23.25	22.5
WARD	'male'	21.75	21.25	20.75	21.25
TORRES	'male'	21.5	21.25	22	21.583
PETERSON	'female'	18.75	20	18	18.917
GRAY	'female'	22.25	21.5	21.75	21.833
RAMIREZ	'female'	24	23	24.5	23.833
JAMES	'male'	19.5	18.75	19.25	19.167
WATSON	'female'	22.75	23.5	23	23.083
BROOKS	'female'	21.5	20.75	21.25	21.167
KELLY	'male'	19.75	19	20.5	19.75

See Also

`findgroups` | `rowfun` | `splitapply` | `summary` | `table` | `varfun`

Related Examples

- “Access Data in a Table” on page 9-42
- “Split Table Data Variables and Apply Functions” on page 9-59

Split Data into Groups and Calculate Statistics

This example shows how to split data from the `patients.mat` data file into groups. Then it shows how to calculate mean weights and body mass indices, and variances in blood pressure readings, for the groups of patients. It also shows how to summarize the results in a table.

Load Patient Data

Load sample data gathered from 100 patients.

```
load patients
```

Convert `Gender` and `SelfAssessedHealthStatus` to categorical arrays.

```
Gender = categorical(Gender);
SelfAssessedHealthStatus = categorical(SelfAssessedHealthStatus);
whos
```

Name	Size	Bytes	Class	Attributes
Age	100x1	800	double	
Diastolic	100x1	800	double	
Gender	100x1	346	categorical	
Height	100x1	800	double	
LastName	100x1	12416	cell	
Location	100x1	15008	cell	
SelfAssessedHealthStatus	100x1	592	categorical	
Smoker	100x1	100	logical	
Systolic	100x1	800	double	
Weight	100x1	800	double	

Calculate Mean Weights

Split the patients into nonsmokers and smokers using the `Smoker` variable. Calculate the mean weight for each group.

```
[G,smoker] = findgroups(Smoker);
meanWeight = splitapply(@mean,Weight,G)
```

```
meanWeight = 2x1
```

```
149.9091
161.9412
```

The `findgroups` function returns `G`, a vector of group numbers created from `Smoker`. The `splitapply` function uses `G` to split `Weight` into two groups. `splitapply` applies the `mean` function to each group and concatenates the mean weights into a vector.

`findgroups` returns a vector of group identifiers as the second output argument. The group identifiers are logical values because `Smoker` contains logical values. The patients in the first group are nonsmokers, and the patients in the second group are smokers.

```
smoker
```

```
smoker = 2x1 logical array
```

```
0
1
```

Split the patient weights by both gender and status as a smoker and calculate the mean weights.

```
G = findgroups(Gender,Smoker);
meanWeight = splitapply(@mean,Weight,G)
```

```
meanWeight = 4x1
```

```
130.3250
130.9231
180.0385
181.1429
```

The unique combinations across `Gender` and `Smoker` identify four groups of patients: female nonsmokers, female smokers, male nonsmokers, and male smokers. Summarize the four groups and their mean weights in a table.

```
[G,gender,smoker] = findgroups(Gender,Smoker);
T = table(gender,smoker,meanWeight)
```

```
T=4x3 table
```

gender	smoker	meanWeight
Female	false	130.32
Female	true	130.92
Male	false	180.04

```
Male      true      181.14
```

T. `gender` contains categorical values, and T. `smoker` contains logical values. The data types of these table variables match the data types of `Gender` and `Smoker` respectively.

Calculate body mass index (BMI) for the four groups of patients. Define a function that takes `Height` and `Weight` as its two input arguments, and that calculates BMI.

```
meanBMIfcn = @(h,w)mean((w ./ (h.^2)) * 703);
BMI = splitapply(meanBMIfcn,Height,Weight,G)
```

```
BMI = 4×1

    21.6721
    21.6686
    26.5775
    26.4584
```

Group Patients Based on Self-Reports

Calculate the fraction of patients who report their health as either `Poor` or `Fair`. First, use `splitapply` to count the number of patients in each group: female nonsmokers, female smokers, male nonsmokers, and male smokers. Then, count only those patients who report their health as either `Poor` or `Fair`, using logical indexing on `S` and `G`. From these two sets of counts, calculate the fraction for each group.

```
[G,gender,smoker] = findgroups(Gender,Smoker);
S = SelfAssessedHealthStatus;
I = ismember(S,{'Poor','Fair'});
numPatients = splitapply(@numel,S,G);
numPF = splitapply(@numel,S(I),G(I));
numPF./numPatients
```

```
ans = 4×1

    0.2500
    0.3846
    0.3077
    0.1429
```

Compare the standard deviation in `Diastolic` readings of those patients who report `Poor` or `Fair` health, and those patients who report `Good` or `Excellent` health.

```
stdDiastolicPF = splitapply(@std,Diastolic(I),G(I));
stdDiastolicGE = splitapply(@std,Diastolic(~I),G(~I));
```

Collect results in a table. For these patients, the female nonsmokers who report Poor or Fair health show the widest variation in blood pressure readings.

```
T = table(gender,smoker,numPatients,numPF,stdDiastolicPF,stdDiastolicGE,BMI)
```

T=4×7 table

gender	smoker	numPatients	numPF	stdDiastolicPF	stdDiastolicGE	BMI
Female	false	40	10	6.8872	3.9012	21
Female	true	13	5	5.4129	5.0409	21
Male	false	26	8	4.2678	4.8159	26
Male	true	21	3	5.6862	5.258	26

See Also

findgroups | splitapply

Related Examples

- “Split Table Data Variables and Apply Functions” on page 9-59

More About

- “Grouping Variables To Split Data” on page 9-71

Split Table Data Variables and Apply Functions

This example shows how to split power outage data from a table into groups by region and cause of the power outages. Then it shows how to apply functions to calculate statistics for each group and collect the results in a table.

Load Power Outage Data

The sample file, `outages.csv`, contains data representing electric utility outages in the United States. The file contains six columns: `Region`, `OutageTime`, `Loss`, `Customers`, `RestorationTime`, and `Cause`. Read `outages.csv` into a table.

```
T = readtable('outages.csv');
```

Convert `Region` and `Cause` to categorical arrays, and `OutageTime` and `RestorationTime` to datetime arrays. Display the first five rows.

```
T.Region = categorical(T.Region);
T.Cause = categorical(T.Cause);
T.OutageTime = datetime(T.OutageTime);
T.RestorationTime = datetime(T.RestorationTime);
T(1:5,:)
```

ans=5x6 table

Region	OutageTime	Loss	Customers	RestorationTime	
SouthWest	2002-02-01 12:18	458.98	1.8202e+06	2002-02-07 16:50	winter
SouthEast	2003-01-23 00:49	530.14	2.1204e+05	NaT	winter
SouthEast	2003-02-07 21:15	289.4	1.4294e+05	2003-02-17 08:14	winter
West	2004-04-06 05:44	434.81	3.4037e+05	2004-04-06 06:10	equipm
MidWest	2002-03-16 06:18	186.44	2.1275e+05	2002-03-18 23:23	severe

Calculate Maximum Power Loss

Determine the greatest power loss due to a power outage in each region. The `findgroups` function returns `G`, a vector of group numbers created from `T.Region`. The `splitapply` function uses `G` to split `T.Loss` into five groups, corresponding to the five regions. `splitapply` applies the `max` function to each group and concatenates the maximum power losses into a vector.

```
G = findgroups(T.Region);
maxLoss = splitapply(@max,T.Loss,G)
```

```
maxLoss = 5×1
104 ×
```

```
2.3141
2.3418
0.8767
0.2796
1.6659
```

Calculate the maximum power loss due to a power outage by cause. To specify that **Cause** is the grouping variable, use table indexing. Create a table that contains the maximum power losses and their causes.

```
T1 = T(:, 'Cause');
[G, powerLosses] = findgroups(T1);
powerLosses.maxLoss = splitapply(@max, T.Loss, G)
```

```
powerLosses=10×2 table
      Cause      maxLoss
-----
attack          582.63
earthquake      258.18
energy emergency 11638
equipment fault 16659
fire            872.96
severe storm   8767.3
thunder storm  23418
unknown        23141
wind           2796
winter storm   2883.7
```

`powerLosses` is a table because `T1` is a table. You can append the maximum losses as another table variable.

Calculate the maximum power loss by cause in each region. To specify that **Region** and **Cause** are the grouping variables, use table indexing. Create a table that contains the maximum power losses and display the first 15 rows.

```
T1 = T(:, {'Region', 'Cause'});
[G, powerLosses] = findgroups(T1);
powerLosses.maxLoss = splitapply(@max, T.Loss, G);
powerLosses(1:15, :)
```

```
ans=15x3 table
  Region      Cause      maxLoss
  _____  _____  _____
  Midwest    attack          0
  Midwest    energy emergency 2378.7
  Midwest    equipment fault  903.28
  Midwest    severe storm     6808.7
  Midwest    thunder storm    15128
  Midwest    unknown          23141
  Midwest    wind             2053.8
  Midwest    winter storm     669.25
  NorthEast  attack          405.62
  NorthEast  earthquake       0
  NorthEast  energy emergency 11638
  NorthEast  equipment fault  794.36
  NorthEast  fire             872.96
  NorthEast  severe storm     6002.4
  NorthEast  thunder storm    23418
```

Calculate Number of Customers Impacted

Determine power-outage impact on customers by cause and region. Because `T.Loss` contains NaN values, wrap `sum` in an anonymous function to use the `'omitnan'` input argument.

```
osumFcn = @(x)(sum(x,'omitnan'));
powerLosses.totalCustomers = splitapply(osumFcn,T.Customers,G);
powerLosses(1:15,:)

```

```
ans=15x4 table
  Region      Cause      maxLoss      totalCustomers
  _____  _____  _____  _____
  Midwest    attack          0              0
  Midwest    energy emergency 2378.7        6.3363e+05
  Midwest    equipment fault  903.28        1.7822e+05
  Midwest    severe storm     6808.7        1.3511e+07
  Midwest    thunder storm    15128         4.2563e+06
  Midwest    unknown          23141         3.9505e+06
  Midwest    wind             2053.8        1.8796e+06
  Midwest    winter storm     669.25        4.8887e+06
  NorthEast  attack          405.62         2181.8
  NorthEast  earthquake       0              0
```

NorthEast	energy emergency	11638	1.4391e+05
NorthEast	equipment fault	794.36	3.9961e+05
NorthEast	fire	872.96	6.1292e+05
NorthEast	severe storm	6002.4	2.7905e+07
NorthEast	thunder storm	23418	2.1885e+07

Calculate Mean Durations of Power Outages

Determine the mean durations of all U.S. power outages in hours. Add the mean durations of power outages to `powerLosses`. Because `T.RestorationTime` has `NaT` values, omit the resulting `NaN` values when calculating the mean durations.

```
D = T.RestorationTime - T.OutageTime;
H = hours(D);
omeanFcn = @(x)(mean(x,'omitnan'));
powerLosses.meanOutage = splitapply(omeanFcn,H,G);
powerLosses(1:15,:)

```

`ans=15x5 table`

Region	Cause	maxLoss	totalCustomers	meanOutage
MidWest	attack	0	0	335.02
MidWest	energy emergency	2378.7	6.3363e+05	5339.3
MidWest	equipment fault	903.28	1.7822e+05	17.863
MidWest	severe storm	6808.7	1.3511e+07	78.906
MidWest	thunder storm	15128	4.2563e+06	51.245
MidWest	unknown	23141	3.9505e+06	30.892
MidWest	wind	2053.8	1.8796e+06	73.761
MidWest	winter storm	669.25	4.8887e+06	127.58
NorthEast	attack	405.62	2181.8	5.5117
NorthEast	earthquake	0	0	0
NorthEast	energy emergency	11638	1.4391e+05	77.345
NorthEast	equipment fault	794.36	3.9961e+05	87.204
NorthEast	fire	872.96	6.1292e+05	4.0267
NorthEast	severe storm	6002.4	2.7905e+07	2163.5
NorthEast	thunder storm	23418	2.1885e+07	46.098

See Also

`findgroups` | `rowfun` | `splitapply` | `varfun`

Related Examples

- “Access Data in a Table” on page 9-42
- “Calculations on Tables” on page 9-51
- “Split Data into Groups and Calculate Statistics” on page 9-55

More About

- “Grouping Variables To Split Data” on page 9-71

Advantages of Using Tables

Conveniently Store Mixed-Type Data in Single Container

You can use the `table` data type to collect mixed-type data and metadata properties, such as variable name, row names, descriptions, and variable units, in a single container. Tables are suitable for column-oriented or tabular data that is often stored as columns in a text file or in a spreadsheet. For example, you can use a table to store experimental data, with rows representing different observations and columns representing different measured variables.

Tables consist of rows and column-oriented variables. Each variable in a table can have a different data type and a different size, but each variable must have the same number of rows.

For example, load sample patients data.

```
load patients
```

Then, combine the workspace variables, `Systolic` and `Diastolic` into a single `BloodPressure` variable and convert the workspace variable, `Gender`, from a cell array of character vectors to a categorical array.

```
BloodPressure = [Systolic Diastolic];
Gender = categorical(Gender);
```

```
whos('Gender', 'Age', 'Smoker', 'BloodPressure')
```

Name	Size	Bytes	Class	Attributes
Age	100x1	800	double	
BloodPressure	100x2	1600	double	
Gender	100x1	346	categorical	
Smoker	100x1	100	logical	

The variables `Age`, `BloodPressure`, `Gender`, and `Smoker` have varying data types and are candidates to store in a table since they all have the same number of rows, 100.

Now, create a table from the variables and display the first five rows.

```
T = table(Gender, Age, Smoker, BloodPressure);
T(1:5, :)
```

```
ans=5x4 table
  Gender    Age    Smoker    BloodPressure
  _____  _____  _____  _____
  Male      38     true      124      93
  Male      43     false     109      77
  Female    38     false     125      83
  Female    40     false     117      75
  Female    49     false     122      80
```

The table displays in a tabular format with the variable names at the top.

Each variable in a table is a single data type. If you add a new row to the table, MATLAB® forces consistency of the data type between the new data and the corresponding table variables. For example, if you try to add information for a new patient where the first column contains the patient's age instead of gender, as in the expression `T(end+1,:) = {37,{'Female'},true,[130 84]}`, then you receive the error:

```
Invalid RHS for assignment to a categorical array.
```

The error occurs because MATLAB® cannot assign numeric data, `37`, to the categorical array, `Gender`.

For comparison of tables with structures, consider the structure array, `StructArray`, that is equivalent to the table, `T`.

```
StructArray = table2struct(T)
```

```
StructArray = 100x1 struct array with fields:
  Gender
  Age
  Smoker
  BloodPressure
```

Structure arrays organize records using named fields. Each field's value can have a different data type or size. Now, display the named fields for the first element of `StructArray`.

```
StructArray(1)
```

```
ans = struct with fields:
  Gender: Male
```

```
Age: 38
Smoker: 1
BloodPressure: [124 93]
```

Fields in a structure array are analogous to variables in a table. However, unlike with tables, you cannot enforce homogeneity within a field. For example, you can have some values of `S.Gender` that are categorical array elements, `Male` or `Female`, others that are character vectors, `'Male'` or `'Female'`, and others that are integers, `0` or `1`.

Now consider the same data stored in a *scalar* structure, with four fields each containing one variable from the table.

```
ScalarStruct = struct(...
    'Gender', {Gender}, ...
    'Age', Age, ...
    'Smoker', Smoker, ...
    'BloodPressure', BloodPressure)

ScalarStruct = struct with fields:
    Gender: [100x1 categorical]
    Age: [100x1 double]
    Smoker: [100x1 logical]
    BloodPressure: [100x2 double]
```

Unlike with tables, you cannot enforce that the data is rectangular. For example, the field `ScalarStruct.Age` can be a different length than the other fields.

A table allows you to maintain the rectangular structure (like a structure array) and enforce homogeneity of variables (like fields in a scalar structure). Although cell arrays do not have named fields, they have many of the same disadvantages as structure arrays and scalar structures. If you have rectangular data that is homogeneous in each variable, consider using a table. Then you can use numeric or named indexing, and you can use table properties to store metadata.

Access Data Using Numeric or Named Indexing

You can index into a table using parentheses, curly braces, or dot indexing. Parentheses allow you to select a subset of the data in a table and preserve the table container. Curly braces and dot indexing allow you to extract data from a table. Within each table indexing method, you can specify the rows or variables to access by name or by numeric index.

Consider the sample table from above. Each row in the table, `T`, represents a different patient. The workspace variable, `LastName`, contains unique identifiers for the 100 rows. Add row names to the table by setting the `RowNames` property to `LastName` and display the first five rows of the updated table.

```
T.Properties.RowNames = LastName;
T(1:5,:)
```

`ans=5x4 table`

	Gender	Age	Smoker	BloodPressure	
Smith	Male	38	true	124	93
Johnson	Male	43	false	109	77
Williams	Female	38	false	125	83
Jones	Female	40	false	117	75
Brown	Female	49	false	122	80

In addition to labeling the data, you can use row and variable names to access data in the table. For example, use named indexing to display the age and blood pressure of the patients `Williams` and `Brown`.

```
T({'Williams', 'Brown'}, {'Age', 'BloodPressure'})
```

`ans=2x2 table`

	Age	BloodPressure	
Williams	38	125	83
Brown	49	122	80

Now, use numeric indexing to return an equivalent subtable. Return the third and fifth row from the second and fourth variables.

```
T(3:2:5, 2:2:4)
```

`ans=2x2 table`

	Age	BloodPressure	
Williams	38	125	83
Brown	49	122	80

With cell arrays or structures, you do not have the same flexibility to use named or numeric indexing.

- With a cell array, you must use `strcmp` to find desired named data, and then you can index into the array.
- With a scalar structure or structure array, it is not possible to refer to a field by number. Furthermore, with a scalar structure, you cannot easily select a subset of variables or a subset of observations. With a structure array, you can select a subset of observations, but you cannot select a subset of variables.
- With a table, you can access data by named index or by numeric index. Furthermore, you can easily select a subset of variables and a subset of rows.

For more information on table indexing, see “Access Data in a Table” on page 9-42.

Use Table Properties to Store Metadata

In addition to storing data, tables have properties to store metadata, such as variable names, row names, descriptions, and variable units. You can access a property using `T.Properties.PropName`, where *T* is the name of the table and *PropName* is one of the table properties.

For example, add a table description, variable descriptions, and variable units for `Age`.

```
T.Properties.Description = 'Simulated Patient Data';
```

```
T.Properties.VariableDescriptions = ...  
    {'Male or Female' ...  
    '' ...  
    'true or false' ...  
    'Systolic/Diastolic'};
```

```
T.Properties.VariableUnits{'Age'} = 'Yrs';
```

Individual empty character vectors within the cell array for `VariableDescriptions` indicate that the corresponding variable does not have a description. For more information, see the `Properties` section of `table`.

To print a table summary, use the `summary` function.

```
summary(T)
```

```
Description: Simulated Patient Data
```

Variables:

Gender: 100x1 categorical

Properties:

Description: Male or Female

Values:

Female	53
Male	47

Age: 100x1 double

Properties:

Units: Yrs

Values:

Min	25
Median	39
Max	50

Smoker: 100x1 logical

Properties:

Description: true or false

Values:

True	34
False	66

BloodPressure: 100x2 double

Properties:

Description: Systolic/Diastolic

Values:

	BloodPressure_1	BloodPressure_2
Min	109	68
Median	122	81.5
Max	138	99

Structures and cell arrays do not have properties for storing metadata.

See Also

summary | table

Related Examples

- “Create and Work with Tables” on page 9-2
- “Modify Units, Descriptions, and Table Variable Names” on page 9-33
- “Access Data in a Table” on page 9-42

Grouping Variables To Split Data

You can use grouping variables to split data variables into groups. Typically, selecting grouping variables is the first step in the *Split-Apply-Combine* workflow. You can split data into groups, apply a function to each group, and combine the results. You also can denote missing values in grouping variables, so that corresponding values in data variables are ignored.

Grouping Variables

Grouping variables are variables used to group, or categorize, observations—that is, data values in other variables. A grouping variable can be any of these data types:

- Numeric, logical, categorical, `datetime`, or `duration` vector
- Cell array of character vectors
- Table, with table variables of any data type in this list

Data variables are the variables that contain observations. A grouping variable must have a value corresponding to each value in the data variables. Data values belong to the same group when the corresponding values in the grouping variable are the same.

This table shows examples of data variables, grouping variables, and the groups that you can create when you split the data variables using the grouping variables.

Data Variable	Grouping Variable	Groups of Data
[5 10 15 20 25 30]	[0 0 0 0 1 1]	[5 10 15 20] [25 30]
[10 20 30 40 50 60]	[1 3 3 1 2 1]	[10 40 60] [50] [20 30]
[64 72 67 69 64 68]	{'F', 'M', 'F', 'M', 'F', 'F'} '}	[64 67 64 68] [72 69]

You can give groups of data meaningful names when you use cell arrays of character vectors or categorical arrays as grouping variables. A categorical array is an efficient and flexible choice of grouping variable.

Group Definition

Typically, there are as many groups as there are unique values in the grouping variable. (A categorical array also can include categories that are not represented in the data.) The groups and the order of the groups depend on the data type of the grouping variable.

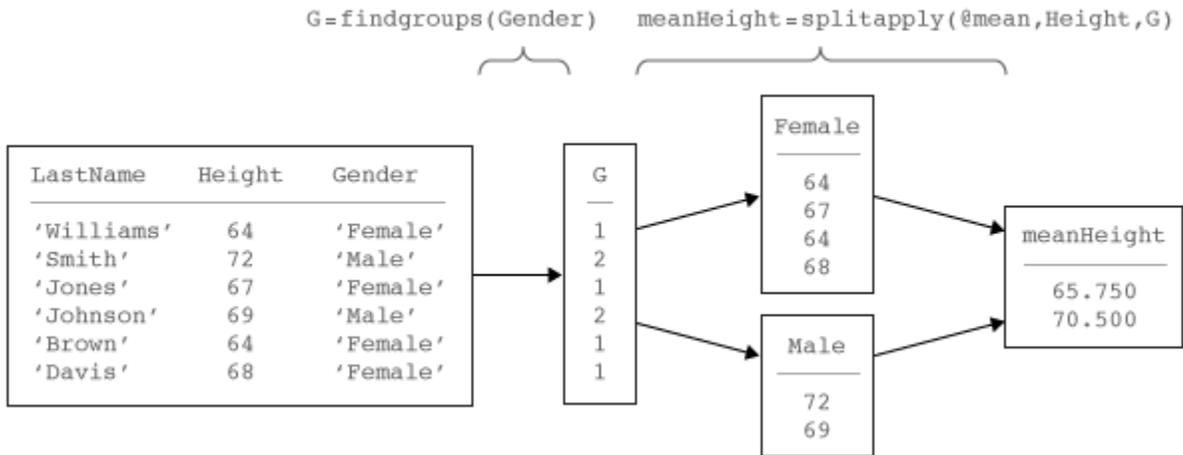
- For numeric, logical, `datetime`, or `duration` vectors, or cell arrays of character vectors, the groups correspond to the unique values sorted in ascending order.
- For categorical arrays, the groups correspond to the unique values observed in the array, sorted in the order returned by the `categories` function.

The `findgroups` function can accept multiple grouping variables, for example `G = findgroups(A1,A2)`. You also can include multiple grouping variables in a table, for example `T = table(A1,A2); G = findgroups(T)`. The `findgroups` function defines groups by the unique combinations of values across corresponding elements of the grouping variables. `findgroups` decides the order by the order of the first grouping variable, and then by the order of the second grouping variable, and so on. For example, if `A1 = {'a','a','b','b'}` and `A2 = [0 1 0 0]`, then the unique values across the grouping variables are 'a' 0, 'a' 1, and 'b' 0, defining three groups.

The Split-Apply-Combine Workflow

After you select grouping variables and split data variables into groups, you can apply functions to the groups and combine the results. This workflow is called the Split-Apply-Combine workflow. You can use the `findgroups` and `splitapply` functions together to analyze groups of data in this workflow. This diagram shows a simple example using the grouping variable `Gender` and the data variable `Height` to calculate the mean height by gender.

The `findgroups` function returns a vector of *group numbers* that define groups based on the unique values in the grouping variables. `splitapply` uses the group numbers to split the data into groups efficiently before applying a function.



Missing Group Values

Grouping variables can have missing values. This table shows the missing value indicator for each data type. If a grouping variable has missing values, then `findgroups` assigns NaN as the group number, and `splitapply` ignores the corresponding values in the data variables.

Grouping Variable Data Type	Missing Value Indicator
Numeric	NaN
Logical	(Cannot be missing)
Categorical	<undefined>
datetime	NaT
duration	NaN
Cell array of character vectors	''
String	<missing>

See Also

`findgroups` | `rowfun` | `splitapply` | `varfun`

Related Examples

- “Access Data in a Table” on page 9-42
- “Split Table Data Variables and Apply Functions” on page 9-59
- “Split Data into Groups and Calculate Statistics” on page 9-55

Changes to DimensionNames Property in R2016b

The table data type is suitable for collecting column-oriented, heterogeneous data in a single container. Tables also contain metadata properties such as variable names, row names, dimension names, descriptions, and variable units. Starting in R2016b, you can use the dimension names to access table data and metadata using dot subscripting. To support that, the dimension names must satisfy the same requirements as the variable names. For backwards compatibility, tables enforce those restrictions by automatically modifying dimension names when needed.

Create a table that has row names and variable names.

```
Number = [8; 21; 13; 20; 11];
Name = {'Van Buren'; 'Arthur'; 'Fillmore'; 'Garfield'; 'Polk'};
Party = categorical({'Democratic'; 'Republican'; 'Whig'; 'Republican'; 'Republican'});
T = table(Number,Party,'RowNames',Name)
```

T =

	Number	Party
Van Buren	8	Democratic
Arthur	21	Republican
Fillmore	13	Whig
Garfield	20	Republican
Polk	11	Republican

Display its properties, including the dimension names. The default values of the dimension names are 'Row' and 'Variables'.

T.Properties

ans =

struct with fields:

```

    Description: ''
    UserData: []
    DimensionNames: {'Row' 'Variables'}
    VariableNames: {'Number' 'Party'}
    VariableDescriptions: {}
    VariableUnits: {}
    RowNames: {5x1 cell}
```

Starting in R2016b, you can assign new names to the dimension names, and use them to access table data. Dimension names must be valid MATLAB identifiers, and must not be one of the reserved names, 'Properties', 'RowNames', or 'VariableNames'.

Assign a new name to the first dimension name, and use it to access the row names of the table.

```
T.Properties.DimensionNames{1} = 'Name';
```

```
T.Name
```

```
ans =
```

```
5x1 cell array
```

```
'Van Buren'  
'Arthur'  
'Fillmore'  
'Garfield'  
'Polk'
```

Create a new table variable called Name. When you create the variable, the table modifies its first dimension name to prevent a conflict. The updated dimension name becomes Name_1.

```
T{:, 'Name'} = {'Martin'; 'Chester'; 'Millard'; 'James'; 'James'}
```

```
Warning: DimensionNames property was modified to avoid conflicting dimension and variable names:  
'Name'. See Compatibility Considerations for Using Tables for more details. This will become an  
error in a future release.
```

```
T =
```

	Number	Party	Name
Van Buren	8	Democratic	'Martin'
Arthur	21	Republican	'Chester'
Fillmore	13	Whig	'Millard'
Garfield	20	Republican	'James'
Polk	11	Republican	'James'

```
T.Properties.DimensionNames
```

```
ans =
```

```
1x2 cell array
```

```
'Name_1' 'Data'
```

Similarly, if you assign a dimension name that is not a valid MATLAB identifier, the name is modified.

```
T.Properties.DimensionNames{1} = 'Last Name';  
T.Properties.DimensionNames
```

```
Warning: DimensionNames property was modified to make the name 'Last Name' a valid MATLAB  
identifier. See Compatibility Considerations for Using Tables for more details. This will  
become an error in a future release.
```

```
ans =
```

```
1x2 cell array
```

```
'LastName' 'Data'
```

In R2016b, tables raise warnings when dimension names are not valid identifiers, or conflict with variable names or reserved names, so that you can continue to work with code and tables created with previous releases. If you encounter these warnings, it is recommended that you update your code to avoid them.

Timetables

- “Create Timetables” on page 10-2
- “Resample and Aggregate Data in Timetable” on page 10-7
- “Combine Timetables and Synchronize Their Data” on page 10-11
- “Retime and Synchronize Timetable Variables Using Different Methods” on page 10-18
- “Select Timetable Data by Row Time and Variable Type” on page 10-24
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times” on page 10-31
- “Using Row Labels in Table and Timetable Operations” on page 10-42

Create Timetables

This example shows how to create a timetable, combine timetables, and adjust the data from multiple timetables to a common time vector. The common time vector can contain the times from either or both timetables, or it can be an entirely new time vector that you specify. The example shows how to compute and display a daily mean for weather measurements contained in different timetables.

A timetable is a type of table that associates a time with each row. A timetable can store column-oriented data variables that have different data types and sizes, so long as each variable has the same number of rows. In addition, timetables provide time-specific functions to combine, subscript into, and adjust their data.

Import Timetables from Files

Load air quality data and weather measurements into two different timetables. The dates of the measurements range from November 15, 2015, to November 19, 2015. The air quality data come from a sensor inside a building, while the weather measurements come from sensors outside.

Read the air quality data from a table with the `readtable` function. Then convert it from a table to a timetable with the `table2timetable` function. The `readtable` function returns a table only, not a timetable.

```
indoors = readtable('indoors.csv');  
indoors = table2timetable(indoors);
```

You also can create a timetable from an M-by-N array with the `array2timetable` function, or from workspace variables with the `timetable` function.

Display the first five rows of `indoors`. Each row of the timetable has a time that labels that row of data.

```
indoors(1:5,:)
```

```
ans=5x2 timetable  
      Time      Humidity      AirQuality  
-----  
2015-11-15 00:00:24      36          80  
2015-11-15 01:13:35      36          80  
2015-11-15 02:26:47      37          79  
2015-11-15 03:39:59      37          82
```

```
2015-11-15 04:53:11      36      80
```

Load the timetable with weather measurements. Display the first five rows of outdoors.

```
load outdoors
outdoors(1:5,:)
```

```
ans=5x3 timetable
      Time      Humidity      TemperatureF      PressureHg
-----
2015-11-15 00:00:24      49      51.3      29.61
2015-11-15 01:30:24     48.9      51.5      29.61
2015-11-15 03:00:24     48.9      51.5      29.61
2015-11-15 04:30:24     48.8      51.5      29.61
2015-11-15 06:00:24     48.7      51.5      29.6
```

Synchronize Timetables

The timetables, `indoors` and `outdoors`, contain different measurements taken inside and outside a building at different times. Combine all the data into one timetable with the `synchronize` function.

```
tt = synchronize(indoors,outdoors);
tt(1:5,:)
```

```
ans=5x5 timetable
      Time      Humidity_indoors      AirQuality      Humidity_outdoors      Temperature
-----
2015-11-15 00:00:24      36      80      49      51.3
2015-11-15 01:13:35      36      80      NaN      51.5
2015-11-15 01:30:24      NaN      NaN      48.9      51.5
2015-11-15 02:26:47      37      79      NaN      51.5
2015-11-15 03:00:24      NaN      NaN      48.9      51.5
```

The output timetable, `tt` contains all the times from both timetables. `synchronize` puts a missing data indicator where there are no data values to place in `tt`. When both input timetables have a variable with the same name, such as `Humidity`, `synchronize` renames both variables and adds both to the output timetable.

Synchronize the timetables again, and this time fill in missing data values with linear interpolation.

```
ttLinear = synchronize(indoors,outdoors,'union','linear');
ttLinear(1:5,:)
```

```
ans=5x5 timetable
```

Time	Humidity_indoors	AirQuality	Humidity_outdoors	Tempo
2015-11-15 00:00:24	36	80	49	
2015-11-15 01:13:35	36	80	48.919	5
2015-11-15 01:30:24	36.23	79.77	48.9	
2015-11-15 02:26:47	37	79	48.9	
2015-11-15 03:00:24	37	80.378	48.9	

Adjust Data in One Timetable

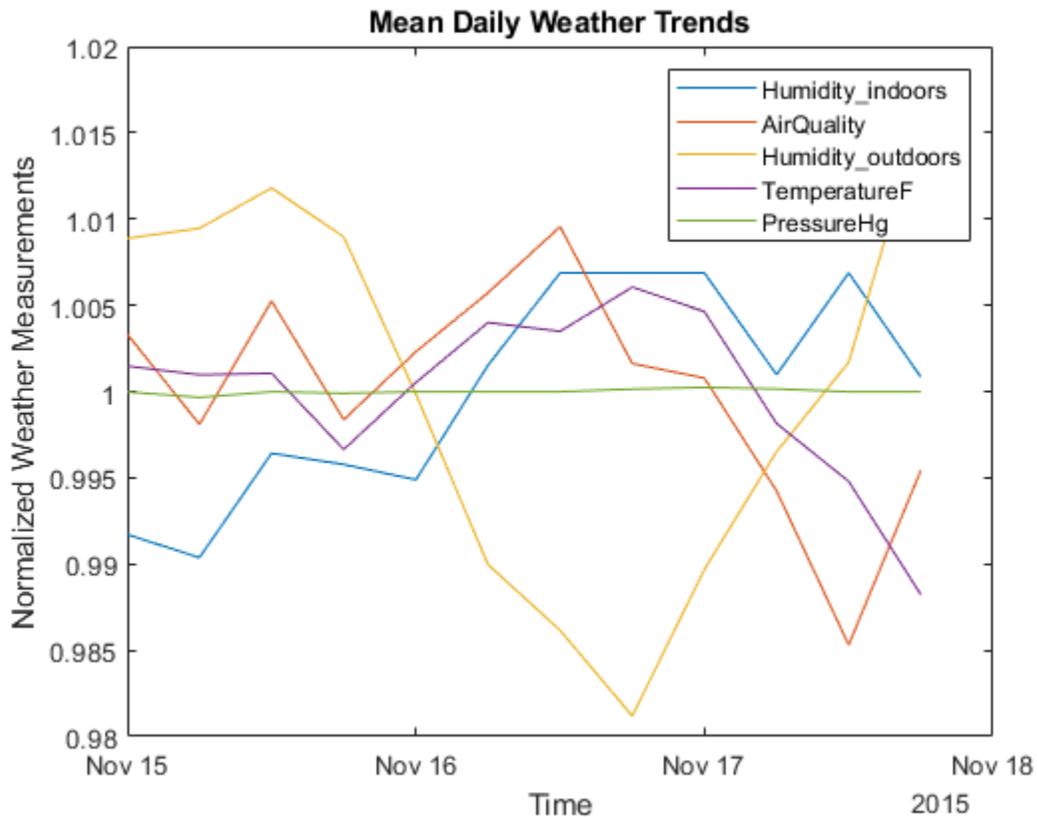
You also can adjust the data in a single timetable to a new time vector. Calculate the means of the variables in `ttLinear` over six-hour intervals with the `retime` function. If any rows have NaN values after you adjust the data, remove them with the `rmmissing` function.

```
tv = [datetime(2015,11,15):hours(6):datetime(2015,11,18)];
ttHourly = retime(ttLinear,tv,'mean');
ttHourly = rmmissing(ttHourly);
```

Plot Timetable Data

Normalize the data in `ttHourly` to the mean for each variable in the timetable. Plot the mean daily values of these measurements. You can use the `Variables` property of a timetable to access the variables. `ttHourly.Variables` returns the same variables as `ttHourly{:, :}`.

```
ttMeanVars = ttHourly.Variables./mean(ttHourly.Variables);
plot(ttHourly.Time,ttMeanVars);
legend(ttHourly.Properties.VariableNames,'Interpreter','none');
xlabel('Time');
ylabel('Normalized Weather Measurements');
title('Mean Daily Weather Trends');
```



See Also

`retime` | `rmmissing` | `synchronize` | `table2timetable` | `timerange` | `timetable`

Related Examples

- “Resample and Aggregate Data in Timetable” on page 10-7
- “Combine Timetables and Synchronize Their Data” on page 10-11
- “Retime and Synchronize Timetable Variables Using Different Methods” on page 10-18
- “Select Timetable Data by Row Time and Variable Type” on page 10-24

- “Clean Timetable with Missing, Duplicate, or Nonuniform Times” on page 10-31

Resample and Aggregate Data in Timetable

This example shows how to resample and aggregate data in a timetable. A timetable is a type of table that associates a time with each row. A timetable can store column-oriented data variables that have different data types and sizes, provided that each variable has the same number of rows. With the `resample` function, you can resample timetable data, or aggregate timetable data into time bins you specify.

Import Timetable

Load a timetable containing weather measurements taken from November 15, 2015, to November 19, 2015. The timetable contains humidity, temperature, and pressure readings taken over this time period.

```
load outdoors
outdoors(1:5,:)
```

```
ans=5x3 timetable
      Time           Humidity   TemperatureF   PressureHg
      _____ _____ _____ _____
2015-11-15 00:00:24         49         51.3         29.61
2015-11-15 01:30:24        48.9         51.5         29.61
2015-11-15 03:00:24        48.9         51.5         29.61
2015-11-15 04:30:24        48.8         51.5         29.61
2015-11-15 06:00:24        48.7         51.5         29.6
```

Determine if the timetable is regular. A regular timetable is one in which the differences between all consecutive row times are the same. `outdoors` is not a regular timetable.

```
TF = isregular(outdoors)
```

```
TF = logical
    0
```

Find the differences in the time steps. They vary between half a minute and an hour and a half.

```
dt = unique(diff(outdoors.Time))
```

```
dt = 3x1 duration array
    00:00:24
```

```
01:29:36
01:30:00
```

Resample Timetable with Interpolation

Adjust the data in the timetable with the `retime` function. Specify an hourly time vector. Interpolate the timetable data to the new row times.

```
TT = retime(outdoors, 'hourly', 'spline');
TT(1:5,:)
```

```
ans=5x3 timetable
      Time      Humidity      TemperatureF      PressureHg
-----
2015-11-15 00:00:00      49.001      51.298      29.61
2015-11-15 01:00:00      48.909      51.467      29.61
2015-11-15 02:00:00      48.902      51.51      29.61
2015-11-15 03:00:00      48.9      51.5      29.61
2015-11-15 04:00:00      48.844      51.498      29.611
```

Resample Timetable with Nearest Neighbor Values

Specify an hourly time vector for `TT`. For each row in `TT`, copy values from the corresponding row in `outdoors` whose row time is nearest.

```
TT = retime(outdoors, 'hourly', 'nearest');
TT(1:5,:)
```

```
ans=5x3 timetable
      Time      Humidity      TemperatureF      PressureHg
-----
2015-11-15 00:00:00      49      51.3      29.61
2015-11-15 01:00:00      48.9      51.5      29.61
2015-11-15 02:00:00      48.9      51.5      29.61
2015-11-15 03:00:00      48.9      51.5      29.61
2015-11-15 04:00:00      48.8      51.5      29.61
```

Aggregate Timetable Data and Calculate Daily Mean

The `retime` function provides aggregation methods, such as `mean`. Calculate the daily means for the data in `outdoors`.

```
TT = retime(outdoors, 'daily', 'mean');
TT
```

```
TT=4x3 timetable
      Time                Humidity    TemperatureF    PressureHg
      _____    _____    _____    _____
2015-11-15 00:00:00    48.931        51.394        29.607
2015-11-16 00:00:00    47.924        51.571        29.611
2015-11-17 00:00:00    48.45         51.238        29.613
2015-11-18 00:00:00    49.5          50.8          29.61
```

Adjust Timetable Data to Regular Times

Calculate the means over six-hour time intervals. Specify a regular time step using the 'regular' input argument and the 'TimeStep' name-value pair argument.

```
TT = retime(outdoors, 'regular', 'mean', 'TimeStep', hours(6));
TT(1:5,:)
```

```
ans=5x3 timetable
      Time                Humidity    TemperatureF    PressureHg
      _____    _____    _____    _____
2015-11-15 00:00:00    48.9         51.45         29.61
2015-11-15 06:00:00    48.9         51.45         29.6
2015-11-15 12:00:00    49.025       51.45         29.61
2015-11-15 18:00:00    48.9         51.225        29.607
2015-11-16 00:00:00    48.5         51.4          29.61
```

As an alternative, you can specify a time vector that has the same six-hour time intervals. Specify a format for the time vector to display both date and time when you display the timetable.

```
tv = datetime(2015,11,15):hours(6):datetime(2015,11,18);
tv.Format = 'dd-MMM-yyyy HH:mm:ss';
TT = retime(outdoors, tv, 'mean');
TT(1:5,:)
```

```
ans=5x3 timetable
      Time                Humidity    TemperatureF    PressureHg
      _____    _____    _____    _____
```

15-Nov-2015 00:00:00	48.9	51.45	29.61
15-Nov-2015 06:00:00	48.9	51.45	29.6
15-Nov-2015 12:00:00	49.025	51.45	29.61
15-Nov-2015 18:00:00	48.9	51.225	29.607
16-Nov-2015 00:00:00	48.5	51.4	29.61

See Also

`retime` | `synchronize` | `table2timetable` | `timetable`

Related Examples

- “Create Timetables” on page 10-2
- “Combine Timetables and Synchronize Their Data” on page 10-11
- “Retime and Synchronize Timetable Variables Using Different Methods” on page 10-18
- “Select Timetable Data by Row Time and Variable Type” on page 10-24
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times” on page 10-31

Combine Timetables and Synchronize Their Data

You can combine timetables and synchronize their data in a variety of ways. You can concatenate timetables vertically or horizontally, but only when they contain the same row times or timetable variables. Use the `synchronize` function to combine timetables with different row times and timetable variables. `synchronize` creates a timetable that contains all variables from all input timetables. It then synchronizes the data from the input timetables to the row times of the output timetable. `synchronize` can fill in missing elements of the output timetable with missing data indicators, with values copied from their nearest neighbors, or with interpolated values. `synchronize` also can aggregate timetable data over time bins you specify.

Concatenate Timetables Vertically

Load timetables from `openPricesSmall` and concatenate them vertically. The timetables are `opWeek1` and `opWeek2`. They contain opening prices for some stocks during the first and second weeks of January 2016.

load `openPricesSmall`

Display the two timetables.

`opWeek1`

```
opWeek1=5x2 timetable
      Time          AAPL      FB
-----
08-Jan-2016 09:00:00   98.55   99.88
07-Jan-2016 09:00:00   98.68  100.5
06-Jan-2016 09:00:00  100.56  101.13
05-Jan-2016 09:00:00  105.75  102.89
04-Jan-2016 09:00:00  102.61  101.95
```

`opWeek2`

```
opWeek2=5x2 timetable
      Time          AAPL      FB
-----
14-Jan-2016 09:00:00   97.96   95.85
13-Jan-2016 09:00:00  100.32  100.58
12-Jan-2016 09:00:00  100.55    99
```

```

11-Jan-2016 09:00:00    98.97    97.91
08-Jan-2016 09:00:00    98.55    99.88

```

Concatenate the timetables. You can concatenate timetables vertically when they have the same variables. The row times label the rows and are not contained in a timetable variable. Note that the row times of a timetable can be out of order and do not need to be regularly spaced. For example, `op` does not include days that fall on weekends. A timetable also can contain duplicate times. `op` contains two rows for 08-Jan-2016 09:00:00.

```
op = [opWeek2;opWeek1]
```

```
op=10x2 timetable
```

```

      Time          AAPL          FB
-----
14-Jan-2016 09:00:00    97.96    95.85
13-Jan-2016 09:00:00   100.32   100.58
12-Jan-2016 09:00:00   100.55     99
11-Jan-2016 09:00:00    98.97    97.91
08-Jan-2016 09:00:00    98.55    99.88
08-Jan-2016 09:00:00    98.55    99.88
07-Jan-2016 09:00:00    98.68   100.5
06-Jan-2016 09:00:00   100.56   101.13
05-Jan-2016 09:00:00   105.75   102.89
04-Jan-2016 09:00:00   102.61   101.95

```

Concatenate Timetables Horizontally

You also can concatenate timetables horizontally. The timetables must have the same row times and different variables.

Display the timetable `opOtherStocks`. The timetable has the same row times as `opWeek1`, but variables for different stocks.

```
opOtherStocks
```

```
opOtherStocks=5x2 timetable
```

```

      Time          MSFT          TWTR
-----
08-Jan-2016 09:00:00    52.37    20.51
07-Jan-2016 09:00:00    52.7     21

```

```

06-Jan-2016 09:00:00    54.32    21.62
05-Jan-2016 09:00:00    54.93    22.79
04-Jan-2016 09:00:00    54.32    22.64

```

Concatenate `opWeek1` and `opOtherStock`. The output timetable has one set of row times and the variables from both timetables.

```
op = [opWeek1 opOtherStocks]
```

```

op=5x4 timetable
      Time          AAPL          FB          MSFT          TWTR
-----
08-Jan-2016 09:00:00    98.55    99.88    52.37    20.51
07-Jan-2016 09:00:00    98.68    100.5    52.7     21
06-Jan-2016 09:00:00   100.56   101.13   54.32   21.62
05-Jan-2016 09:00:00   105.75   102.89   54.93   22.79
04-Jan-2016 09:00:00   102.61   101.95   54.32   22.64

```

Synchronize Timetables and Indicate Missing Data

Load air quality data and weather measurements from two different timetables and synchronize them. The dates of the measurements range from November 15, 2015, to November 19, 2015. The air quality data come from a sensor inside a building, while the weather measurements come from sensors outside.

```
load indoors
load outdoors
```

Display the first five lines of each timetable. They contain measurements of different quantities taken at different times.

```

indoors(1:5,:)

ans=5x2 timetable
      Time          Humidity          AirQuality
-----
2015-11-15 00:00:24         36           80
2015-11-15 01:13:35         36           80
2015-11-15 02:26:47         37           79
2015-11-15 03:39:59         37           82

```

```
2015-11-15 04:53:11      36      80
```

```
outdoors(1:5,:)
```

```
ans=5x3 timetable
```

Time	Humidity	TemperatureF	PressureHg
2015-11-15 00:00:24	49	51.3	29.61
2015-11-15 01:30:24	48.9	51.5	29.61
2015-11-15 03:00:24	48.9	51.5	29.61
2015-11-15 04:30:24	48.8	51.5	29.61
2015-11-15 06:00:24	48.7	51.5	29.6

Synchronize the timetables. The output timetable `tt` contains all the times from both timetables. `synchronize` puts a missing data indicator where there are no data values to place in `tt`. When both input timetables have a variable with the same name, such as `Humidity`, `synchronize` renames both variables and adds both to the output timetable.

```
tt = synchronize(indoors,outdoors);
tt(1:5,:)
```

```
ans=5x5 timetable
```

Time	Humidity_indoors	AirQuality	Humidity_outdoors	Tempo
2015-11-15 00:00:24	36	80	49	5
2015-11-15 01:13:35	36	80	NaN	5
2015-11-15 01:30:24	NaN	NaN	48.9	5
2015-11-15 02:26:47	37	79	NaN	5
2015-11-15 03:00:24	NaN	NaN	48.9	5

Synchronize and Interpolate Data Values

Synchronize the timetables, and fill in missing timetable elements with linear interpolation. To synchronize on a time vector that includes all times from both timetables, specify `'union'` for the output times.

```
ttLinear = synchronize(indoors,outdoors,'union','linear');
ttLinear(1:5,:)
```

```
ans=5x5 timetable
```

Time	Humidity_indoors	AirQuality	Humidity_outdoors	Tempo
------	------------------	------------	-------------------	-------

Time	Humidity_indoors	AirQuality	Humidity_outdoors	Temperature
2015-11-15 00:00:24	36	80	49	53.1
2015-11-15 01:13:35	36	80	48.919	53.1
2015-11-15 01:30:24	36.23	79.77	48.9	53.1
2015-11-15 02:26:47	37	79	48.9	53.1
2015-11-15 03:00:24	37	80.378	48.9	53.1

Synchronize to Regular Times

Synchronize the timetables to an hourly time vector. The input timetables had irregular row times. The output timetable has regular row times with one hour as the time step.

```
ttHourly = synchronize(indoors,outdoors,'hourly','linear');
ttHourly(1:5,:)
```

ans=5x5 timetable

Time	Humidity_indoors	AirQuality	Humidity_outdoors	Temperature
2015-11-15 00:00:00	36	80	49	53.1
2015-11-15 01:00:00	36	80	48.934	53.1
2015-11-15 02:00:00	36.634	79.366	48.9	53.1
2015-11-15 03:00:00	37	80.361	48.9	53.1
2015-11-15 04:00:00	36.727	81.453	48.834	53.1

Synchronize the timetables to a 30-minute time step. Specify a regular time step using the 'regular' input argument and the 'TimeStep' name-value pair argument.

```
ttHalfHour = synchronize(indoors,outdoors,'regular','linear','TimeStep',minutes(30));
ttHalfHour(1:5,:)
```

ans=5x5 timetable

Time	Humidity_indoors	AirQuality	Humidity_outdoors	Temperature
2015-11-15 00:00:00	36	80	49	53.1
2015-11-15 00:30:00	36	80	48.967	53.1
2015-11-15 01:00:00	36	80	48.934	53.1
2015-11-15 01:30:00	36.224	79.776	48.9	53.1
2015-11-15 02:00:00	36.634	79.366	48.9	53.1

As an alternative, you can synchronize the timetables to a time vector that specifies half-hour intervals.

```
tv = [datetime(2015,11,15):minutes(30):datetime(2015,11,18)];
tv.Format = indoors.Time.Format;
ttHalfHour = synchronize(indoors,outdoors,tv,'linear');
ttHalfHour(1:5,:)
```

```
ans=5x5 timetable
      Time          Humidity_indoors    AirQuality    Humidity_outdoors    Temperature
-----
2015-11-15 00:00:00          36          80          49          53
2015-11-15 00:30:00          36          80         48.967          53
2015-11-15 01:00:00          36          80         48.934          53
2015-11-15 01:30:00         36.224         79.776          48.9          53
2015-11-15 02:00:00         36.634         79.366          48.9
```

Synchronize and Aggregate Data Values

Synchronize the timetables and calculate the daily means for all variables in the output timetable.

```
ttDaily = synchronize(indoors,outdoors,'daily','mean');
ttDaily
```

```
ttDaily=4x5 timetable
      Time          Humidity_indoors    AirQuality    Humidity_outdoors    Temperature
-----
2015-11-15 00:00:00          36.5          80.05         48.931          53
2015-11-16 00:00:00          36.85          80.35         47.924          53
2015-11-17 00:00:00          36.85          79.45         48.45          53
2015-11-18 00:00:00          NaN          NaN          49.5
```

Synchronize the timetables to six-hour time intervals and calculate a mean for each interval.

```
tt6Hours = synchronize(indoors,outdoors,'regular','mean','TimeStep',hours(6));
tt6Hours(1:5,:)
```

```
ans=5x5 timetable
      Time          Humidity_indoors    AirQuality    Humidity_outdoors    Temperature
```

2015-11-15 00:00:00	36.4	80.2	48.9	5
2015-11-15 06:00:00	36.4	79.8	48.9	5
2015-11-15 12:00:00	36.6	80.4	49.025	5
2015-11-15 18:00:00	36.6	79.8	48.9	5
2015-11-16 00:00:00	36.6	80.2	48.5	5

As an alternative, specify a time vector that has the same six-hour time intervals.

```
tv = [datetime(2015,11,15):hours(6):datetime(2015,11,18)];
tv.Format = indoors.Time.Format;
tt6Hours = synchronize(indoors,outdoors,tv,'mean');
tt6Hours(1:5,:)
```

```
ans=5x5 timetable
      Time          Humidity_indoors    AirQuality    Humidity_outdoors    Temp
-----
2015-11-15 00:00:00          36.4          80.2          48.9          5
2015-11-15 06:00:00          36.4          79.8          48.9          5
2015-11-15 12:00:00          36.6          80.4          49.025          5
2015-11-15 18:00:00          36.6          79.8          48.9          5
2015-11-16 00:00:00          36.6          80.2          48.5          5
```

See Also

[retime](#) | [synchronize](#) | [table2timetable](#) | [timetable](#)

Related Examples

- “Create Timetables” on page 10-2
- “Resample and Aggregate Data in Timetable” on page 10-7
- “Retime and Synchronize Timetable Variables Using Different Methods” on page 10-18
- “Select Timetable Data by Row Time and Variable Type” on page 10-24
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times” on page 10-31

Retime and Synchronize Timetable Variables Using Different Methods

This example shows how to fill in gaps in timetable variables, using different methods for different variables. You can specify whether each timetable variable contains continuous or discrete data using the `VariableContinuity` property of the timetable. When you resample the timetable using the `retime` function, `retime` either interpolates, fills in with previous values, or fills in with missing data indicators, depending on the values in the `VariableContinuity` property. Similarly, the `synchronize` function interpolates or fills in values based on the `VariableContinuity` property of the input timetables.

Create Timetable

Create a timetable that has simulated weather measurements for several days in May 2017. The timetable variables `Tmax` and `Tmin` contain maximum and minimum temperature readings for each day, and `PrecipTotal` contains total precipitation for the day. `WXEvent` is a categorical array, recording whether certain kinds of weather events, such as thunder or hail, happened on any given day. The timetable has simulated data from May 4 to May 10, 2017, but is missing data for two days, May 6th and May 7th.

```
Date = [datetime(2017,5,4) datetime(2017,5,5) datetime(2017,5,8:10)];
Tmax = [60 62 56 59 60]';
Tmin = [44 45 40 42 45]';
PrecipTotal = [0.2 0 0 0.15 0]';
WXEvent = [2 0 0 1 0]';
WXEvent = categorical(WXEvent,[0 1 2 3 4],{'None','Thunder','Hail','Fog','Tornado'});
Station1 = timetable(Date,Tmax,Tmin,PrecipTotal,WXEvent)
```

Station1=5x4 timetable

Date	Tmax	Tmin	PrecipTotal	WXEvent
04-May-2017	60	44	0.2	Hail
05-May-2017	62	45	0	None
08-May-2017	56	40	0	None
09-May-2017	59	42	0.15	Thunder
10-May-2017	60	45	0	None

Resample Continuous and Discrete Timetable Variables

One way to fill in data for the two missing days is to use the `retime` function. If you call `retime` without specifying a method, then `retime` fills in gaps with missing data

indicators. For instance, `retime` fills gaps in numeric variables with NaN values, and gaps in the categorical variable with undefined elements.

```
Station1Daily = retime(Station1, 'daily')
```

```
Station1Daily=7x4 timetable
      Date      Tmax    Tmin    PrecipTotal    WXEvent
      -----
04-May-2017    60     44      0.2           Hail
05-May-2017    62     45      0             None
06-May-2017   NaN     NaN     NaN           <undefined>
07-May-2017   NaN     NaN     NaN           <undefined>
08-May-2017    56     40      0             None
09-May-2017    59     42     0.15          Thunder
10-May-2017    60     45      0             None
```

If you specify a method when you call `retime`, it uses the same method to fill gaps in every variable. To apply different methods to different variables, you can call `retime` multiple times, each time indexing into the timetable to access a different subset of variables.

However, you also can apply different methods by specifying the `VariableContinuity` property of the timetable. You can specify whether each variable contains continuous or discrete data. Then the `retime` function applies a different method to each timetable variable, depending on the corresponding `VariableContinuity` value.

If you specify `VariableContinuity`, then the `retime` function fills in the output timetable variables using the following methods:

- 'unset' — Fill in values using the missing data indicator for that type (such as NaN for numeric variables).
- 'continuous' — Fill in values using linear interpolation.
- 'step' — Fill in values using previous value.
- 'event' — Fill in values using the missing data indicator for that type.

Specify that the temperature data in `Station1` is continuous, that `PrecipTotal` is step data, and that `WXEvent` is event data.

```
Station1.Properties.VariableContinuity = {'continuous', 'continuous', 'step', 'event'};
Station1.Properties
```

```
ans =
  TimetableProperties with properties:

    Description: ''
    UserData: []
    DimensionNames: {'Date' 'Variables'}
    VariableNames: {'Tmax' 'Tmin' 'PrecipTotal' 'WXEvent'}
    VariableDescriptions: {}
    VariableUnits: {}
    VariableContinuity: [continuous continuous step event]
    RowTimes: [5x1 datetime]
    StartTime: 04-May-2017
    SampleRate: NaN
    TimeStep: NaN
    CustomProperties: No custom properties are set.
    Use addprop and rmprop to modify CustomProperties.
```

Resample the data in `Station1`. Given the values assigned to `VariableContinuity`, the `retime` function interpolates the temperature data, fills in the previous day's values in `PrecipTotal`, and fills in `WXEvent` with undefined elements.

```
Station1Daily = retime(Station1,'daily')
```

```
Station1Daily=7x4 timetable
    Date      Tmax    Tmin    PrecipTotal    WXEvent
    _____  _____  _____  _____  _____
    04-May-2017    60        44        0.2        Hail
    05-May-2017    62        45         0        None
    06-May-2017    60       43.333         0    <undefined>
    07-May-2017    58       41.667         0    <undefined>
    08-May-2017    56        40         0        None
    09-May-2017    59        42        0.15    Thunder
    10-May-2017    60        45         0        None
```

If you specify a method, then `retime` applies that method to all variables, overriding the values in `VariableContinuity`.

```
Station1Missing = retime(Station1,'daily','fillwithmissing')
```

```
Station1Missing=7x4 timetable
    Date      Tmax    Tmin    PrecipTotal    WXEvent
    _____  _____  _____  _____  _____
```

04-May-2017	60	44	0.2	Hail
05-May-2017	62	45	0	None
06-May-2017	NaN	NaN	NaN	<undefined>
07-May-2017	NaN	NaN	NaN	<undefined>
08-May-2017	56	40	0	None
09-May-2017	59	42	0.15	Thunder
10-May-2017	60	45	0	None

Synchronize Timetables That Contain Continuous and Discrete Data

The `synchronize` function also fills in output timetable variables using different methods, depending on the values specified in the `VariableContinuity` property of each input timetable.

Create a second timetable that contains pressure readings in millibars from a second weather station. The timetable has simulated readings from May 4 to May 8, 2017.

```
Date = datetime(2017,5,4:8)';
Pressure = [995 1003 1013 1018 1006]';
Station2 = timetable(Date,Pressure)
```

```
Station2=5x1 timetable
      Date      Pressure
      _____  _____
      04-May-2017      995
      05-May-2017     1003
      06-May-2017     1013
      07-May-2017     1018
      08-May-2017     1006
```

Synchronize the data from the two stations using the `synchronize` function. `synchronize` fills in values for variables from `Station1` according to the values in the `VariableContinuity` property of `Station1`. However, since the `VariableContinuity` property of `Station2` is empty, `synchronize` fills in `Pressure` with `NaN` values.

```
BothStations = synchronize(Station1,Station2)
```

```
BothStations=7x5 timetable
      Date      Tmax      Tmin      PrecipTotal      WXEvent      Pressure
      _____  _____  _____  _____  _____  _____
```

04-May-2017	60	44	0.2	Hail	995
05-May-2017	62	45	0	None	1003
06-May-2017	60	43.333	0	<undefined>	1013
07-May-2017	58	41.667	0	<undefined>	1018
08-May-2017	56	40	0	None	1006
09-May-2017	59	42	0.15	Thunder	NaN
10-May-2017	60	45	0	None	NaN

To indicate that `Station2.Pressure` contains continuous data, specify the `VariableContinuity` property of `Station2`. Though `Station2` contains only one variable, you must specify `VariableContinuity` using a cell array, not a character vector.

```
Station2.Properties.VariableContinuity = {'continuous'};
Station2.Properties
```

```
ans =
```

```
TimetableProperties with properties:
```

```

    Description: ''
    UserData: []
    DimensionNames: {'Date' 'Variables'}
    VariableNames: {'Pressure'}
VariableDescriptions: {}
    VariableUnits: {}
    VariableContinuity: continuous
    RowTimes: [5x1 datetime]
    StartTime: 04-May-2017
    SampleRate: NaN
    TimeStep: 1d
    CustomProperties: No custom properties are set.
    Use addprop and rmprop to modify CustomProperties.
```

Synchronize the data from the two stations. `synchronize` fills in values in `BothStations.Pressure` because `Station2.Pressure` has continuous data.

```
BothStations = synchronize(Station1,Station2)
```

```
BothStations=7x5 timetable
```

Date	Tmax	Tmin	PrecipTotal	WXEvent	Pressure
_____	_____	_____	_____	_____	_____

04-May-2017	60	44	0.2	Hail	995
05-May-2017	62	45	0	None	1003
06-May-2017	60	43.333	0	<undefined>	1013
07-May-2017	58	41.667	0	<undefined>	1018
08-May-2017	56	40	0	None	1006
09-May-2017	59	42	0.15	Thunder	994
10-May-2017	60	45	0	None	982

If you specify a method as an input argument to `synchronize`, then `synchronize` applies that method to all variables, just as the `retime` function does.

See Also

`retime` | `synchronize` | `timetable`

Related Examples

- “Create Timetables” on page 10-2
- “Resample and Aggregate Data in Timetable” on page 10-7
- “Combine Timetables and Synchronize Their Data” on page 10-11
- “Select Timetable Data by Row Time and Variable Type” on page 10-24
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times” on page 10-31

Select Timetable Data by Row Time and Variable Type

A timetable is a type of table that associates a time with each row. You can subscript into a timetable to select subsets of its data in a number of different ways. To select timetable rows with row times that fall within a given time range, specify the time range using the `timerange` function. Since a timetable is a table, you can index on rows and variables using either smooth parentheses or curly braces. You can index on specific row times, or select rows with row times that match specified times within a tolerance you set using the `withtol` function. You can also subscript into a table or timetable to select all the variables that match a type you specify with the `vartype` function. Finally, extract data from a timetable into a matrix using the `Variables` property.

Create Timetable from File

Create a timetable from the sample file `outages.csv`, containing data representing electric utility outages in the United States. Read the table from the file with the `readtable` function. Convert `T.Cause` and `T.Region` into categorical arrays. Then convert the table to a timetable using the `table2timetable` function. Display the first five rows of the timetable. `TT` is a timetable containing outage data from February 2002 to January 2014.

```
T = readtable('outages.csv');
T.Cause = categorical(T.Cause);
T.Region = categorical(T.Region);
TT = table2timetable(T);
TT(1:5,:)
```

```
ans=5x5 timetable
      OutageTime
```

OutageTime	Region	Loss	Customers	RestorationTime	
2002-02-01 12:18	SouthWest	458.98	1.8202e+06	2002-02-07 16:50	winter
2003-01-23 00:49	SouthEast	530.14	2.1204e+05	NaT	winter
2003-02-07 21:15	SouthEast	289.4	1.4294e+05	2003-02-17 08:14	winter
2004-04-06 05:44	West	434.81	3.4037e+05	2004-04-06 06:10	equipm
2002-03-16 06:18	MidWest	186.44	2.1275e+05	2002-03-18 23:23	severe

Summarize Timetable and Access Row Times

Display a summary of `TT`. It is a timetable that contains 1468 rows and five variables.

```
summary(TT)
```

RowTimes:

OutageTime: 1468x1 datetime

Values:

Min	2002-02-01 12:18
Median	2010-03-18 21:05
Max	2014-01-15 02:41

Variables:

Region: 1468x1 categorical

Values:

MidWest	142
NorthEast	557
SouthEast	389
SouthWest	26
West	354

Loss: 1468x1 double

Values:

Min	0
Median	180.26
Max	23418
NumMissing	604

Customers: 1468x1 double

Values:

Min	0
Median	75765
Max	5.9689e+06
NumMissing	328

RestorationTime: 1468x1 datetime

Values:

Min	2002-02-07 16:50
-----	------------------

```
Median      2010-03-31 10:54
Max         2042-09-18 23:31
NumMissing  29
```

Cause: 1468x1 categorical

Values:

```
attack      294
earthquake  2
energy emergency 188
equipment fault 156
fire        25
severe storm 338
thunder storm 201
unknown     24
wind        95
winter storm 145
```

Access the row times. The row times are not in a variable. Instead, the vector of row times is a property of the timetable. However, you can access the row times with dot syntax. `TT.OutageTime` is a 1468-by-1 vector of datetime values. Display the first five rows of `TT.OutageTime`.

```
TT.OutageTime(1:5)
```

```
ans = 5x1 datetime array
2002-02-01 12:18
2003-01-23 00:49
2003-02-07 21:15
2004-04-06 05:44
2002-03-16 06:18
```

Subscript on Time Range

To select all timetable rows that fall in a time range, create a subscript as a helper, using the `timerange` function. The start and end times you specify do not have to match any of the row times in the timetable.

Select all rows with outages that occurred between January 2002 and December 2003. Display the first five rows of `TT2`.

```
TR = timerange('2002-01-01','2003-12-31');
TT2 = TT(TR,:);
TT2(1:5,:)
```

```
ans=5x5 timetable
```

OutageTime	Region	Loss	Customers	RestorationTime	Ca
2002-02-01 12:18	SouthWest	458.98	1.8202e+06	2002-02-07 16:50	winter
2003-01-23 00:49	SouthEast	530.14	2.1204e+05	NaT	winter
2003-02-07 21:15	SouthEast	289.4	1.4294e+05	2003-02-17 08:14	winter
2002-03-16 06:18	MidWest	186.44	2.1275e+05	2002-03-18 23:23	severe
2003-06-18 02:49	West	0	0	2003-06-18 10:54	attach

Display the last five rows of TT2.

```
TT2(end-4:end,:)
```

```
ans=5x5 timetable
```

OutageTime	Region	Loss	Customers	RestorationTime	(
2003-09-02 19:46	SouthEast	0	0	2003-09-16 22:25	severe
2003-09-15 14:56	MidWest	418.7	61045	2003-09-22 04:21	thunde
2003-09-24 22:43	SouthWest	2576.9	9.4873e+05	2003-09-25 14:46	severe
2003-09-18 10:40	SouthWest	301.8	2.3973e+05	2003-09-27 08:17	severe
2003-10-11 19:36	SouthEast	309.8	93582	2003-10-11 19:49	energy

TT2 is a timetable with only 98 rows, containing outage data only from 2002 and 2003.

Index on Specified Times

You can index into TT with datetime values or character vectors representing specific times in TT.OutageTime. But when you do so, the times you specify must have exact matches in the time vector, and only those times are selected. Index into TT on times for the first and third rows of TT.

```
TT({'2002-02-01 12:18:00','2003-02-07 21:15:00'},:)
```

```
ans=2x5 timetable
```

OutageTime	Region	Loss	Customers	RestorationTime	Ca
------------	--------	------	-----------	-----------------	----

```

2002-02-01 12:18   SouthWest   458.98   1.8202e+06   2002-02-07 16:50   winter
2003-02-07 21:15   SouthEast   289.4    1.4294e+05   2003-02-17 08:14   winter

```

Index on Specified Times with Tolerances

Specify a tolerance when indexing on times. You can use the `withtol` function to create a subscript as a helper. With the output of `withtol`, you can select row times that match within the specified tolerance.

Index into `TT` on specified dates. Specify a tolerance of one day to return rows with row times that are within one day of the specified dates. The times must be in a datetime or duration vector, or in a cell array of character vectors that can be converted to datetime or duration values. The tolerance must be specified as a duration, using a function such as `seconds`, `minutes`, `hours`, or `days`.

```

rowTimes = {'2002-02-01', '2003-02-07'};
S = withtol(rowTimes, days(1));
TT(S, :)

```

```

ans=2x5 timetable
      OutageTime      Region      Loss      Customers      RestorationTime      Ca
-----
2002-02-01 12:18   SouthWest   458.98   1.8202e+06   2002-02-07 16:50   winter
2003-02-07 21:15   SouthEast   289.4    1.4294e+05   2003-02-17 08:14   winter

```

Subscript by Variable Type

To select all timetable variables that are of a given type, create a subscript as a helper, using the `vartype` function. You can specify variable types without having to specify their names or positions in the timetable.

Select all variables that contain numeric data. `TT2` contains only the variables `Loss` and `Customers`. The other three variables of `TT` are either categorical or datetime variables. Display the first five rows of `TT2`.

```

S = vartype('numeric');
TT2 = TT(:, S);
TT2(1:5, :)

```

```

ans=5x2 timetable
      OutageTime      Loss      Customers

```

2002-02-01 12:18	458.98	1.8202e+06
2003-01-23 00:49	530.14	2.1204e+05
2003-02-07 21:15	289.4	1.4294e+05
2004-04-06 05:44	434.81	3.4037e+05
2002-03-16 06:18	186.44	2.1275e+05

Subscript both on a time range and by variable type.

```
TR = timerange('2002-01-01', '2003-12-31');
TT2 = TT(TR,S);
TT2(1:5,:)
```

```
ans=5x2 timetable
      OutageTime      Loss      Customers
```

	OutageTime	Loss	Customers
	2002-02-01 12:18	458.98	1.8202e+06
	2003-01-23 00:49	530.14	2.1204e+05
	2003-02-07 21:15	289.4	1.4294e+05
	2002-03-16 06:18	186.44	2.1275e+05
	2003-06-18 02:49	0	0

Extract Data with Variables Property

Tables and timetables have a property, `Variables`, that you can use to extract data from variables into a matrix, as long as the variables can be concatenated together.

Extract the numeric data from `TT2` using the `Variables` property. `A` is a 1468-by-2 matrix of doubles. When you extract data from a timetable into an array, the row times are not included.

```
A = TT2.Variables;
A(1:5,:)
```

```
ans = 5x2
106 ×
```

0.0005	1.8202
0.0005	0.2120
0.0003	0.1429
0.0002	0.2128

```
0      0
```

The result of `TT2.Variables` is the same as the result of using curly braces to extract data, using the `TT2{:,:}` syntax.

You can concatenate the variables in `TT2` into an array of doubles. However, `TT` contains numeric, categorical, and datetime variables that cannot be concatenated. The `Variables` property returns an error when variables cannot be concatenated. To avoid such an error, you can subscript by variable type before using the `Variables` property.

Subscript into `TT` to select numeric variables and extract them into a matrix.

```
A = TT(:,vartype('numeric')).Variables;  
A(1:5,:)
```

```
ans = 5×2  
106 ×
```

```
0.0005    1.8202  
0.0005    0.2120  
0.0003    0.1429  
0.0004    0.3404  
0.0002    0.2128
```

See Also

`retime` | `synchronize` | `table2timetable` | `timerange` | `timetable` | `vartype` | `withtol`

Related Examples

- “Create Timetables” on page 10-2
- “Resample and Aggregate Data in Timetable” on page 10-7
- “Combine Timetables and Synchronize Their Data” on page 10-11
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times” on page 10-31

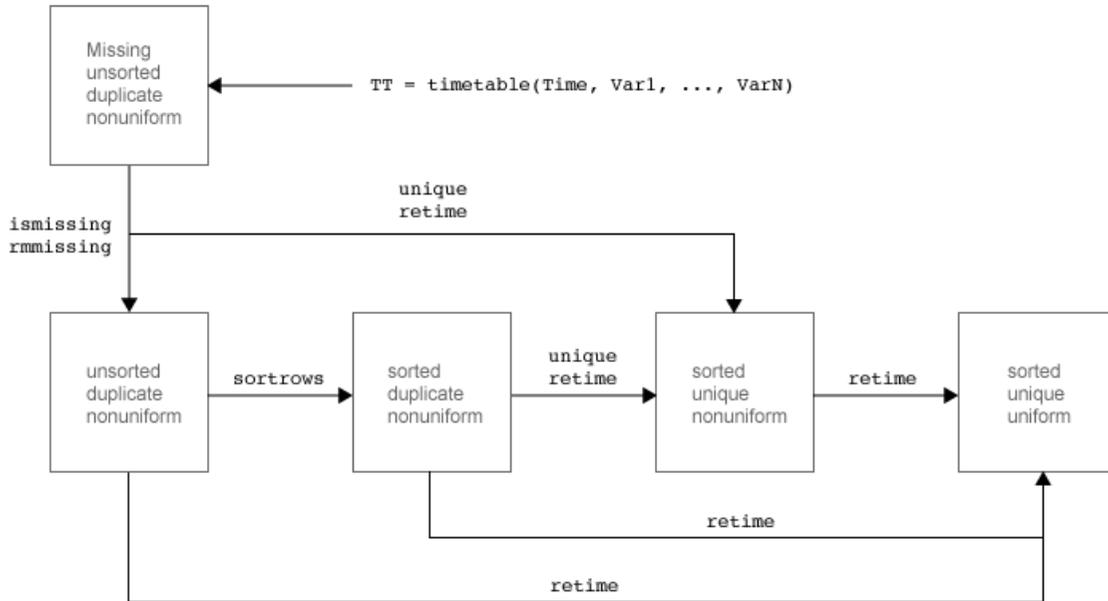
Clean Timetable with Missing, Duplicate, or Nonuniform Times

This example shows how to create a *regular* timetable from one that has missing, duplicate, or nonuniform times. A timetable is a type of table that associates a time-stamp, or *row time*, with each row of data. In a regular timetable, the row times are sorted and unique, and differ by the same regular time step. The example also shows how to export the data from a timetable for use with other functions.

Timetables can be irregular. They can contain rows that are not sorted by their row times. Timetables can contain multiple rows with the same row time, though the rows can have different data values. Even when row times are sorted and unique, they can differ by time steps of different sizes. Timetables can even contain `NaN` or `NaT` values to indicate missing row times.

Timetables provide a number of different ways to resolve missing, duplicate, or nonuniform times, and to resample or aggregate data to regular row times.

- To find missing row times, use **`ismissing`**.
- To remove missing times and data, use **`rmmissing`**.
- To sort a timetable by its row times, use **`sortrows`**.
- To make a timetable with unique and sorted row times, use **`unique`** and **`retime`**.
- To remove duplicate times, specify a unique time vector and use **`retime`**.
- To make a regular timetable, specify a regular time vector and use **`retime`**.



Load Timetable

Load a sample timetable from the MAT-file `badTimes` that contains weather measurements taken over several hours on June 9, 2016. The timetable includes temperature, rainfall, and wind speed measurements taken at irregular times during that day.

```
load badTimes
TT
```

```
TT=12x3 timetable
```

Time	Temp	Rain	WindSpeed
09-Jun-2016 06:01:04	73	0.01	2.3
09-Jun-2016 07:59:23	59	0.08	0.9
09-Jun-2016 09:53:57	59	0.03	3.4
09-Jun-2016 09:53:57	67	0.03	3.4

NaT		56	0	0
09-Jun-2016	09:53:57	67	0.03	3.4
09-Jun-2016	08:49:10	62	0.01	2.7
09-Jun-2016	08:49:10	75.8	0.01	2.7
09-Jun-2016	08:49:10	82	0.01	2.7
09-Jun-2016	05:03:11	66.2	0.05	3
09-Jun-2016	08:49:10	67.2	0.01	2.7
09-Jun-2016	04:12:00	58.8	NaN	NaN

Remove Rows with Missing Times

Remove rows that have NaT, or a missing value, as the row time. To find missing values in the vector of row times, use the `ismissing` function. `ismissing` returns a logical vector that contains 1 wherever `TT.Time` has a missing value. Index back into the timetable to keep only those rows that do not have missing values as row times. Assign those rows to `TT2`.

```
TF = ismissing(TT.Time);
TT2 = TT(~TF,:);
TT2
```

```
TT2=11x3 timetable
      Time      Temp  Rain  WindSpeed
    _____  _____  _____  _____
    09-Jun-2016 06:01:04    73    0.01    2.3
    09-Jun-2016 07:59:23    59    0.08    0.9
    09-Jun-2016 09:53:57    59    0.03    3.4
    09-Jun-2016 09:53:57    67    0.03    3.4
    09-Jun-2016 09:53:57    67    0.03    3.4
    09-Jun-2016 08:49:10    62    0.01    2.7
    09-Jun-2016 08:49:10   75.8    0.01    2.7
    09-Jun-2016 08:49:10    82    0.01    2.7
    09-Jun-2016 05:03:11   66.2    0.05    3
    09-Jun-2016 08:49:10   67.2    0.01    2.7
    09-Jun-2016 04:12:00   58.8    NaN    NaN
```

This method removes only the rows that have missing row times. The table variables might still have missing data values. For example, the last row of `TT2` has NaN values for the `Rain` and `WindSpeed` variables.

Remove Rows with Missing Times or Missing Data

You can remove missing row times and missing data values using the `rmmissing` function. `rmmissing` removes any timetable row that has a missing row time, missing data values, or both.

Display the missing row time and missing data values of `TT`. Then remove all missing values from `TT`.

`TT`

`TT=12x3 timetable`

Time	Temp	Rain	WindSpeed
09-Jun-2016 06:01:04	73	0.01	2.3
09-Jun-2016 07:59:23	59	0.08	0.9
09-Jun-2016 09:53:57	59	0.03	3.4
09-Jun-2016 09:53:57	67	0.03	3.4
NaT	56	0	0
09-Jun-2016 09:53:57	67	0.03	3.4
09-Jun-2016 08:49:10	62	0.01	2.7
09-Jun-2016 08:49:10	75.8	0.01	2.7
09-Jun-2016 08:49:10	82	0.01	2.7
09-Jun-2016 05:03:11	66.2	0.05	3
09-Jun-2016 08:49:10	67.2	0.01	2.7
09-Jun-2016 04:12:00	58.8	NaN	NaN

`TT = rmmissing(TT)`

`TT=10x3 timetable`

Time	Temp	Rain	WindSpeed
09-Jun-2016 06:01:04	73	0.01	2.3
09-Jun-2016 07:59:23	59	0.08	0.9
09-Jun-2016 09:53:57	59	0.03	3.4
09-Jun-2016 09:53:57	67	0.03	3.4
09-Jun-2016 09:53:57	67	0.03	3.4
09-Jun-2016 08:49:10	62	0.01	2.7
09-Jun-2016 08:49:10	75.8	0.01	2.7
09-Jun-2016 08:49:10	82	0.01	2.7
09-Jun-2016 05:03:11	66.2	0.05	3

```
09-Jun-2016 08:49:10    67.2    0.01    2.7
```

Sort Timetable and Determine Whether It Is Regular

Determine whether TT is sorted. Then, sort the timetable on its row times using the `sortrows` function.

```
TF = issorted(TT)
```

```
TF = logical
    0
```

```
TT = sortrows(TT)
```

```
TT=10x3 timetable
           Time           Temp    Rain    WindSpeed
-----
09-Jun-2016 05:03:11    66.2    0.05         3
09-Jun-2016 06:01:04     73    0.01         2.3
09-Jun-2016 07:59:23     59    0.08         0.9
09-Jun-2016 08:49:10     62    0.01         2.7
09-Jun-2016 08:49:10    75.8    0.01         2.7
09-Jun-2016 08:49:10     82    0.01         2.7
09-Jun-2016 08:49:10    67.2    0.01         2.7
09-Jun-2016 09:53:57     59    0.03         3.4
09-Jun-2016 09:53:57     67    0.03         3.4
09-Jun-2016 09:53:57     67    0.03         3.4
```

Determine whether TT is regular. A regular timetable has the same time interval between consecutive row times. Even a sorted timetable can have time steps that are not uniform.

```
TF = isregular(TT)
```

```
TF = logical
    0
```

Display the differences between row times.

```
diff(TT.Time)
```

```
ans = 9x1 duration array
    00:57:53
```

```
01:58:19
00:49:47
00:00:00
00:00:00
00:00:00
01:04:47
00:00:00
00:00:00
```

Remove Duplicate Rows

Timetables can have duplicate rows. Timetable rows are duplicates if they have the same row times and the same data values. In this example, the last two rows of TT are duplicates.

To remove the duplicate rows, use the `unique` function. `unique` returns the unique rows and sorts them by their row times.

```
TT = unique(TT)
```

```
TT=9x3 timetable
      Time           Temp   Rain   WindSpeed
-----
09-Jun-2016 05:03:11   66.2   0.05         3
09-Jun-2016 06:01:04    73   0.01         2.3
09-Jun-2016 07:59:23    59   0.08         0.9
09-Jun-2016 08:49:10    62   0.01         2.7
09-Jun-2016 08:49:10   67.2  0.01         2.7
09-Jun-2016 08:49:10   75.8  0.01         2.7
09-Jun-2016 08:49:10    82   0.01         2.7
09-Jun-2016 09:53:57    59   0.03         3.4
09-Jun-2016 09:53:57    67   0.03         3.4
```

Find Rows with Duplicate Times and Different Data

Timetables can have rows with duplicate row times but different data values. In this example, TT has several rows with the same row times but different values.

Find the rows that have duplicate row times. First, sort the row times and find consecutive times that have no difference between them. Times with no difference between them are the duplicates. Index back into the vector of row times and return a unique set of times that identify the duplicate row times in TT.

```

dupTimes = sort(TT.Time);
TF = (diff(dupTimes) == 0);
dupTimes = dupTimes(TF);
dupTimes = unique(dupTimes)

dupTimes = 2x1 datetime array
    09-Jun-2016 08:49:10
    09-Jun-2016 09:53:57

```

Index into the timetable to display the rows with duplicate row times. When you index on times, the output timetable contains all rows with matching row times.

```
TT(dupTimes,:)
```

```
ans=6x3 timetable
      Time           Temp    Rain    WindSpeed
-----
09-Jun-2016 08:49:10     62    0.01         2.7
09-Jun-2016 08:49:10    67.2    0.01         2.7
09-Jun-2016 08:49:10    75.8    0.01         2.7
09-Jun-2016 08:49:10     82    0.01         2.7
09-Jun-2016 09:53:57     59    0.03         3.4
09-Jun-2016 09:53:57     67    0.03         3.4
```

Select First and Last Rows with Duplicate Times

Select either the first and the last of the rows with duplicate row times using the `unique` and `retime` functions.

First, create a vector of unique row times from `TT` using the `unique` function.

```
uniqueTimes = unique(TT.Time);
```

Select the first row from each set of rows that have duplicate times.

```
TT2 = retime(TT,uniqueTimes)
```

```
TT2=5x3 timetable
      Time           Temp    Rain    WindSpeed
-----
09-Jun-2016 05:03:11    66.2    0.05         3
```

```

09-Jun-2016 06:01:04    73    0.01    2.3
09-Jun-2016 07:59:23    59    0.08    0.9
09-Jun-2016 08:49:10    62    0.01    2.7
09-Jun-2016 09:53:57    59    0.03    3.4

```

Select the last rows from each set of rows that have duplicate times. Specify the 'previous' method of `retime` to copy data from the last row. When you specify 'previous', then `retime` starts at the end of the vector of row times and stops when it encounters a duplicate row time. Then it copies the data from that row.

```
TT2 = retime(TT,uniqueTimes,'previous')
```

```

TT2=5x3 timetable
      Time           Temp    Rain    WindSpeed
      _____  _____  _____  _____
09-Jun-2016 05:03:11    66.2    0.05         3
09-Jun-2016 06:01:04     73    0.01         2.3
09-Jun-2016 07:59:23     59    0.08         0.9
09-Jun-2016 08:49:10     82    0.01         2.7
09-Jun-2016 09:53:57     67    0.03         3.4

```

Aggregate Data from All Rows with Duplicate Times

Aggregate data from rows that have duplicate row times. For example, you can calculate the means of several measurements of the same quantity taken at the same time.

Calculate the mean temperature, rainfall, and wind speed for rows with duplicate row times using the `retime` function.

```
TT = retime(TT,uniqueTimes,'mean')
```

```

TT=5x3 timetable
      Time           Temp    Rain    WindSpeed
      _____  _____  _____  _____
09-Jun-2016 05:03:11    66.2    0.05         3
09-Jun-2016 06:01:04     73    0.01         2.3
09-Jun-2016 07:59:23     59    0.08         0.9
09-Jun-2016 08:49:10    71.75    0.01         2.7
09-Jun-2016 09:53:57     63    0.03         3.4

```

Make Timetable Regular

Create a regular timetable using `retime`. Interpolate the data onto a regular hourly time vector. To use linear interpolation, specify `'linear'`. Each row time in `TT` begins on the hour, and there is a one-hour interval between consecutive row times.

```
TT = retime(TT, 'hourly', 'linear')
```

```
TT=6x3 timetable
      Time           Temp      Rain      WindSpeed
      _____ _____ _____ _____
09-Jun-2016 05:00:00  65.826    0.0522    3.0385
09-Jun-2016 06:00:00  72.875    0.010737   2.3129
09-Jun-2016 07:00:00  66.027    0.044867   1.6027
09-Jun-2016 08:00:00  59.158    0.079133   0.9223
09-Jun-2016 09:00:00  70.287    0.013344   2.8171
09-Jun-2016 10:00:00  62.183    0.031868   3.4654
```

Instead of using a predefined time step such as `'hourly'`, you can specify a time step of your own. To specify a time step of 30 minutes, use the `'regular'` input argument and the `'TimeStep'` name-value pair argument. You can specify a time step of any size as a duration or calendar duration value.

```
TT = retime(TT, 'regular', 'linear', 'TimeStep', minutes(30))
```

```
TT=11x3 timetable
      Time           Temp      Rain      WindSpeed
      _____ _____ _____ _____
09-Jun-2016 05:00:00  65.826    0.0522    3.0385
09-Jun-2016 05:30:00   69.35    0.031468   2.6757
09-Jun-2016 06:00:00  72.875    0.010737   2.3129
09-Jun-2016 06:30:00  69.451    0.027802   1.9578
09-Jun-2016 07:00:00  66.027    0.044867   1.6027
09-Jun-2016 07:30:00  62.592     0.062    1.2625
09-Jun-2016 08:00:00  59.158    0.079133   0.9223
09-Jun-2016 08:30:00  64.722    0.046239   1.8697
09-Jun-2016 09:00:00  70.287    0.013344   2.8171
09-Jun-2016 09:30:00  66.235    0.022606   3.1412
09-Jun-2016 10:00:00  62.183    0.031868   3.4654
```

Extract Regular Timetable Data

You can export the timetable data for use with functions to analyze data that is regularly spaced in time. For example, the Econometrics Toolbox™ and the Signal Processing Toolbox™ have functions you can use for further analysis on regularly spaced data.

Extract the timetable data as an array. You can use the `Variables` property to return the data as an array when the table variables can be concatenated.

```
A = TT.Variables
```

```
A = 11×3
```

```
65.8260    0.0522    3.0385
69.3504    0.0315    2.6757
72.8747    0.0107    2.3129
69.4507    0.0278    1.9578
66.0266    0.0449    1.6027
62.5923    0.0620    1.2625
59.1579    0.0791    0.9223
64.7224    0.0462    1.8697
70.2868    0.0133    2.8171
66.2348    0.0226    3.1412
    ⋮
```

`TT.Variables` is equivalent to using curly brace syntax, `TT{:, :}`, to access all variables.

```
A2 = TT{:, :}
```

```
A2 = 11×3
```

```
65.8260    0.0522    3.0385
69.3504    0.0315    2.6757
72.8747    0.0107    2.3129
69.4507    0.0278    1.9578
66.0266    0.0449    1.6027
62.5923    0.0620    1.2625
59.1579    0.0791    0.9223
64.7224    0.0462    1.8697
70.2868    0.0133    2.8171
66.2348    0.0226    3.1412
```

⋮

See Also

`diff` | `fillmissing` | `isregular` | `issorted` | `retime` | `rmmissing` | `sortrows` | `table2timetable` | `timetable` | `unique`

Related Examples

- “Create Timetables” on page 10-2
- “Resample and Aggregate Data in Timetable” on page 10-7
- “Combine Timetables and Synchronize Their Data” on page 10-11
- “Retime and Synchronize Timetable Variables Using Different Methods” on page 10-18
- “Select Timetable Data by Row Time and Variable Type” on page 10-24

Using Row Labels in Table and Timetable Operations

Tables and timetables provide ways to label the rows in your data. In tables, you can label the rows with names. In timetables, you must label the rows with dates, times, or both. Row names are optional for tables, but row times are required for timetables. These row labels are part of the metadata in a table or timetable. In some functions you also can use row labels as key variables, grouping variables, and so on, just as you can use the data variables in a table or timetable. These functions are `sortrows`, `join`, `innerjoin`, `outerjoin`, `varfun`, `rowfun`, `stack`, and `unstack`. There are some limitations on using these table functions and on using row labels as key variables.

Sort on Row Labels

For example, you can sort a timetable on its row times, on one or more of its data variables, or on row times and data variables together.

Create a timetable using the `timetable` function. A timetable has row times along its first dimension, labeling the rows. The row times are a property of the timetable, not a timetable variable.

```
Date = datetime(2016,7,[10;10;11;11;10;10;11;11]);
X = [1;1;1;1;2;2;2;2];
Y = {'a';'b';'a';'b';'a';'b';'a';'b'};
Z = [1;2;3;4;5;6;7;8];
TT = timetable(X,Y,Z,'RowTimes',Date)
```

```
TT=8x3 timetable
      Time      X      Y      Z
    _____
10-Jul-2016    1    'a'    1
10-Jul-2016    1    'b'    2
11-Jul-2016    1    'a'    3
11-Jul-2016    1    'b'    4
10-Jul-2016    2    'a'    5
10-Jul-2016    2    'b'    6
11-Jul-2016    2    'a'    7
11-Jul-2016    2    'b'    8
```

Rename the first dimension. By default, the name of the first dimension of a timetable is `Time`. You can access the `Properties.DimensionNames` property to rename a dimension.

```
TT.Properties.DimensionNames{1} = 'Date';
TT.Properties.DimensionNames
```

```
ans = 1x2 cell array
```

```
    {'Date'}    {'Variables'}
```

As an alternative, you can specify the row times as the first input argument to `timetable`, without specifying `'RowTimes'`. The `timetable` function names the row times, or the first dimension, after the first input argument, just as it names the timetable variables after the other input arguments.

```
TT = timetable(Date,X,Y,Z)
```

```
TT=8x3 timetable
```

Date	X	Y	Z
10-Jul-2016	1	'a'	1
10-Jul-2016	1	'b'	2
11-Jul-2016	1	'a'	3
11-Jul-2016	1	'b'	4
10-Jul-2016	2	'a'	5
10-Jul-2016	2	'b'	6
11-Jul-2016	2	'a'	7
11-Jul-2016	2	'b'	8

Sort the timetable by row times. To sort on row times, refer to the first dimension of the timetable by name.

```
sortrows(TT, 'Date')
```

```
ans=8x3 timetable
```

Date	X	Y	Z
10-Jul-2016	1	'a'	1
10-Jul-2016	1	'b'	2
10-Jul-2016	2	'a'	5
10-Jul-2016	2	'b'	6
11-Jul-2016	1	'a'	3
11-Jul-2016	1	'b'	4
11-Jul-2016	2	'a'	7
11-Jul-2016	2	'b'	8

Sort by the data variables X and Y. `sortrows` sorts on X first, then on Y.

```
sortrows(TT, {'X' 'Y'})
```

```
ans=8×3 timetable
  Date      X      Y      Z
  -----
  10-Jul-2016  1      'a'    1
  11-Jul-2016  1      'a'    3
  10-Jul-2016  1      'b'    2
  11-Jul-2016  1      'b'    4
  10-Jul-2016  2      'a'    5
  11-Jul-2016  2      'a'    7
  10-Jul-2016  2      'b'    6
  11-Jul-2016  2      'b'    8
```

Sort by row times and X together.

```
sortrows(TT,{'Date' 'X'})
```

```
ans=8×3 timetable
  Date      X      Y      Z
  -----
  10-Jul-2016  1      'a'    1
  10-Jul-2016  1      'b'    2
  10-Jul-2016  2      'a'    5
  10-Jul-2016  2      'b'    6
  11-Jul-2016  1      'a'    3
  11-Jul-2016  1      'b'    4
  11-Jul-2016  2      'a'    7
  11-Jul-2016  2      'b'    8
```

Use Row Labels as Grouping or Key Variables

When you group rows together using the `rowfun`, `varfun`, `stack`, and `unstack` functions, you can specify row labels as grouping variables. When you join tables or timetable together using the `join`, `innerjoin`, and `outerjoin` functions, you can specify row labels as key variables.

For example, you can perform an inner join two tables together, using row names and a table variable together as key variables. An inner join keeps only those table rows that match with respect to the key variables.

Create two tables of patient data. A table can have row names along its first dimension, labeling the rows, but is not required to have them. Specify the last names of patients as the row names of the tables. Add the first names of the patients as table variables.

```
A = table({'Michael';'Louis';'Alice';'Rosemary';'Julie'},[38;43;45;40;49],...
         'VariableNames',{'FirstName' 'Age'},...
         'RowNames',{'Garcia' 'Johnson' 'Wu' 'Jones' 'Picard'})
```

A=5×2 table

	FirstName	Age
Garcia	'Michael'	38
Johnson	'Louis'	43
Wu	'Alice'	45
Jones	'Rosemary'	40
Picard	'Julie'	49

```
B = table({'Michael';'Beverly';'Alice'},...
         [64;69;67],...
         [119;163;133],...
         [122 80; 109 77; 117 75],...
         'VariableNames',{'FirstName' 'Height' 'Weight' 'BloodPressure'},...
         'RowNames',{'Garcia' 'Johnson' 'Wu'})
```

B=3×4 table

	FirstName	Height	Weight	BloodPressure
Garcia	'Michael'	64	119	122 80
Johnson	'Beverly'	69	163	109 77
Wu	'Alice'	67	133	117 75

If a table has row names, then you can index into it by row name. Indexing by row names is a convenient way to select rows of a table. Index into B by a patient's last name to retrieve information about the patient.

```
B('Garcia',:)
```

ans=1×4 table

	FirstName	Height	Weight	BloodPressure
Garcia	'Michael'	64	119	122 80

Perform an inner join on the two tables. Both tables use the last names of patients as row names, and contain the first names as a table variable. Some patients in the two tables have matching last names but different first names. To ensure that both last and first names match, use the row names and `FirstName` as key variables. To specify the row names as a key or grouping variable, use the name of the first dimension of the table. By default, the name of the first dimension is `'Row'`.

```
C = innerjoin(A,B, 'Keys', {'Row', 'FirstName'})
```

C=2x5 table

	FirstName	Age	Height	Weight	BloodPressure	
Garcia	'Michael'	38	64	119	122	80
Wu	'Alice'	45	67	133	117	75

If you rename the first dimension of a table, then you can refer to the row names by that name instead of using 'Row'. Perform the same inner join as above but use a different name to refer to the row names.

Show the dimension names of A by accessing its `Properties.DimensionNames` property.

```
A.Properties.DimensionNames
```

ans = 1x2 cell array

```
    {'Row'}    {'Variables'}
```

Change the name of the first dimension of the table by using its `Properties.DimensionNames` property. Then use the new name as a key variable.

```
A.Properties.DimensionNames{1} = 'LastName';
```

```
A.Properties.DimensionNames
```

ans = 1x2 cell array

```
    {'LastName'}    {'Variables'}
```

Perform an inner join on A and B using `LastName` and `FirstName` as key variables.

```
B.Properties.DimensionNames{1} = 'LastName';
```

```
D = innerjoin(A,B, 'Keys', {'LastName', 'FirstName'})
```

D=2x5 table

	FirstName	Age	Height	Weight	BloodPressure	
Garcia	'Michael'	38	64	119	122	80
Wu	'Alice'	45	67	133	117	75

Notes on Use of Table Functions and Row Labels

- You cannot stack or unstack row labels using the `stack` and `unstack` functions. However, you can use row labels as grouping variables.

- You cannot perform a join using the `join`, `innerjoin`, or `outerjoin` functions when the first argument is a table and the second argument is a timetable. However, you can perform a join when both arguments are tables, both are timetables, or the first argument is a timetable and the second is a table.
- The output of a join operation can have row labels if you specify row labels as key variables. For more details on row labels from a join operation, see the documentation on the `'Keys'`, `'LeftKeys'`, and `'RightKeys'` arguments of the `join`, `innerjoin`, and `outerjoin` functions.

See Also

`innerjoin` | `join` | `outerjoin` | `rowfun` | `sortrows` | `stack` | `unstack` | `varfun`

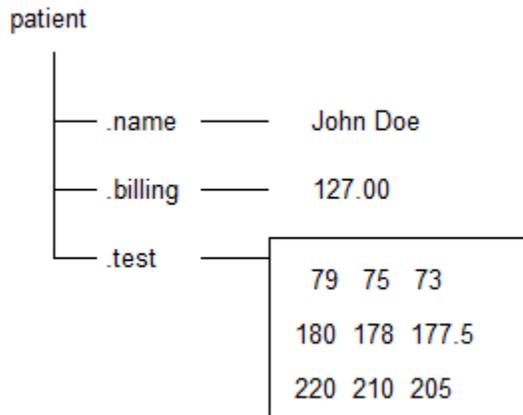
Structures

- “Create Structure Array” on page 11-2
- “Access Data in a Structure Array” on page 11-7
- “Concatenate Structures” on page 11-11
- “Generate Field Names from Variables” on page 11-13
- “Access Data in Nested Structures” on page 11-14
- “Access Elements of a Nonscalar Struct Array” on page 11-16
- “Ways to Organize Data in Structure Arrays” on page 11-18
- “Memory Requirements for a Structure Array” on page 11-22

Create 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.

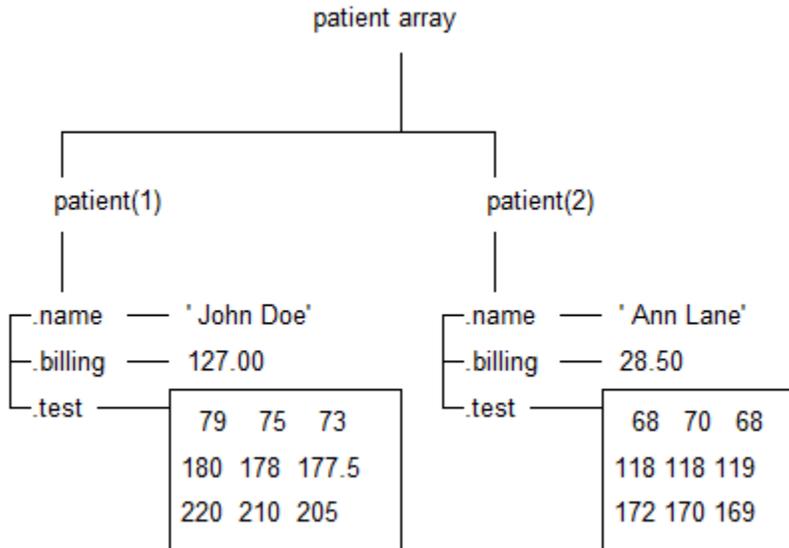
Store a patient record in a scalar structure with fields `name`, `billing`, and `test`.



```
patient(1).name = 'John Doe';
patient(1).billing = 127.00;
patient(1).test = [79, 75, 73; 180, 178, 177.5; 220, 210, 205];
patient
```

```
patient = struct with fields:
    name: 'John Doe'
    billing: 127
    test: [3x3 double]
```

Add records for other patients to the array by including subscripts after the array name.



```

patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68, 70, 68; 118, 118, 119; 172, 170, 169];
patient

```

```

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)
```

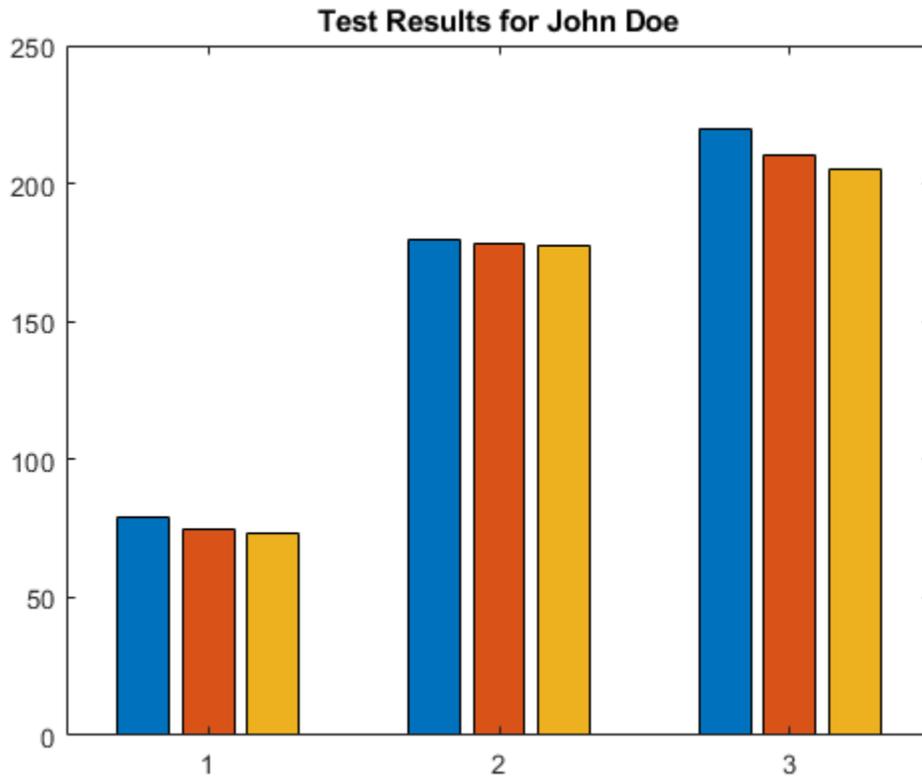
```
ans = struct with fields:  
    name: 'New Name'  
    billing: []  
    test: []
```

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
```

```
amount_due = 127
```

```
bar(patient(1).test)  
title(['Test Results for ', patient(1).name])
```



See Also

Related Examples

- "Access Data in a Structure Array" on page 11-7
- "Create Cell Array" on page 12-3
- "Create and Work with Tables" on page 9-2

More About

- “Cell vs. Struct Arrays” on page 12-19
- “Advantages of Using Tables” on page 9-64

Access Data in a Structure Array

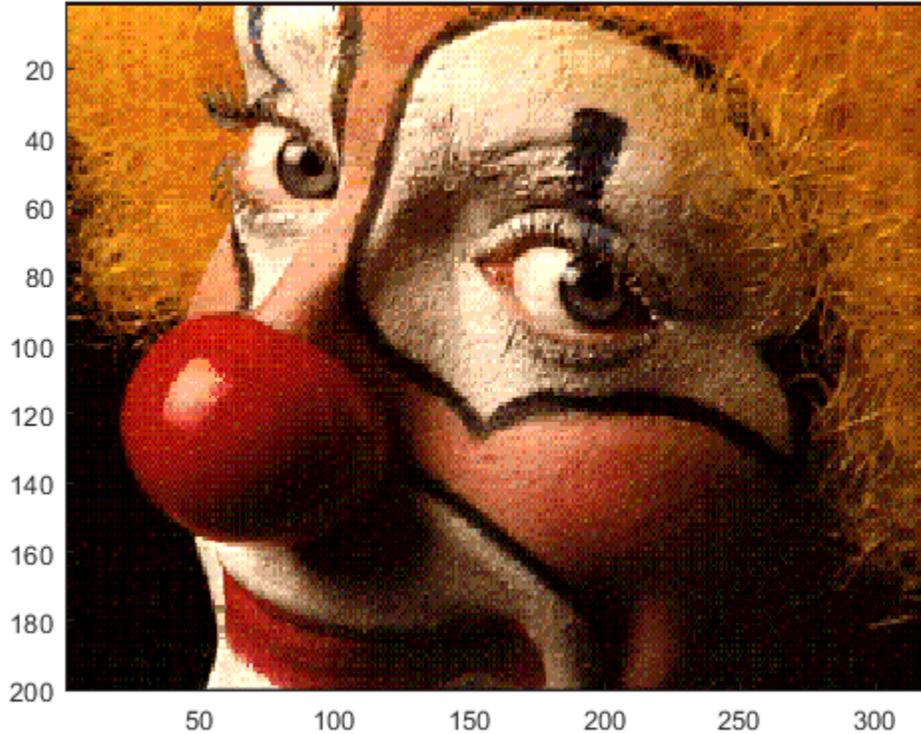
This example shows how to access the contents of a structure array. To run the code in this example, load several variables into a scalar (1-by-1) structure named `S`.

```
S = load('clown.mat')  
  
S = struct with fields:  
    X: [200x320 double]  
    map: [81x3 double]  
    caption: [2x1 char]
```

The variables from the file (`X`, `caption`, and `map`) are now fields in the struct.

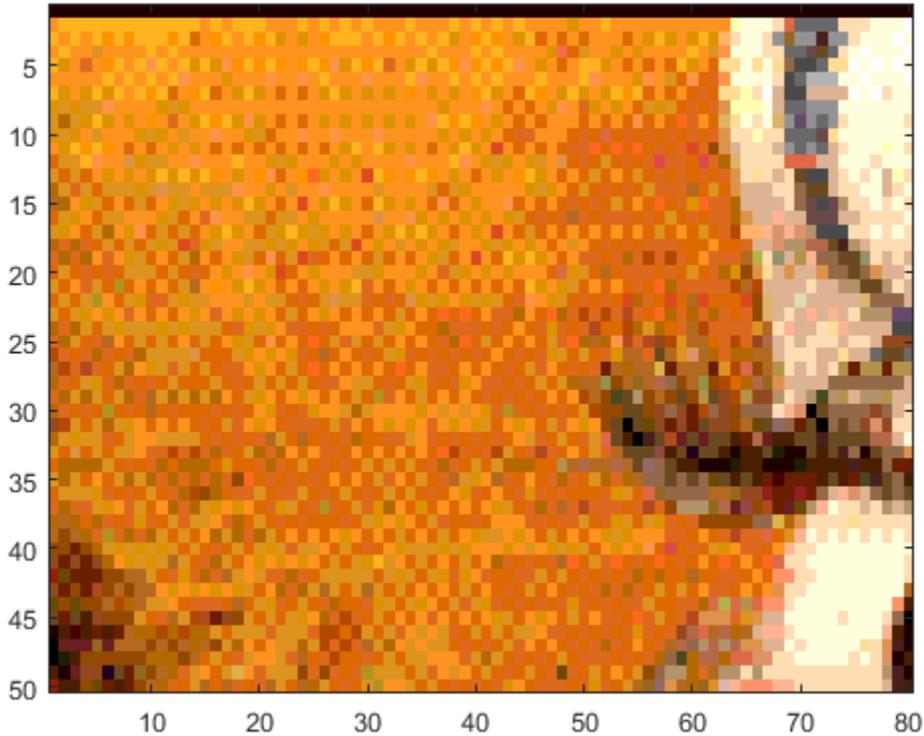
Access the data using dot notation of the form `structName.fieldName`. For example, pass the numeric data in field `X` to the `image` function:

```
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.

See Also

Related Examples

- “Access Data in Nested Structures” on page 11-14
- “Access Elements of a Nonscalar Struct Array” on page 11-16
- “Generate Field Names from Variables” on page 11-13

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);
struct1,struct2

struct1 = struct with fields:
  a: 'first'
  b: [1 2 3]

struct2 = struct with fields:
  a: 'second'
  b: [5x5 double]
```

Just as concatenating two scalar values such as `[1,2]` creates a 1-by-2 numeric array, concatenating `struct1` and `struct2` creates a 1-by-2 structure array.

```
combined = [struct1,struct2]

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

ans =
'first'
```

Concatenation also applies to nonscalar structure arrays. For example, create a 2-by-2 structure array named `new`. 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.

```
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;
```

```
larger = [combined; new]
```

```
larger = 3x2 struct array with fields:  
    a  
    b
```

Access field `a` of the structure `larger(2,1)`. It contains the same value as `new(1,1).a`.

```
larger(2,1).a
```

```
ans = 1
```

See Also

Related Examples

- “Creating, Concatenating, and Expanding Matrices”
- “Access Data in a Structure Array” on page 11-7
- “Access Elements of a Nonscalar Struct Array” on page 11-16

Generate Field Names from Variables

This example shows how to derive a structure field name at run time from a variable or expression. The general syntax is

```
structName.(dynamicExpression)
```

where `dynamicExpression` is a variable or expression that, when evaluated, returns a character vector or, starting in R2017b, a string scalar. 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, 'mmdd');
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]
```

The dynamic field name can return either a character vector or a string scalar. For example, you can specify the field `Feb29` using either single or, starting in R2017b, double quotes.

```
myStruct.('Feb29')
```

```
ans =
     1     2     3
```

```
myStruct.("Feb29")
```

```
ans =
     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 1-5.

Access Data in Nested Structures

This example shows how to index into a structure that is nested within another structure. The general syntax for accessing data in a particular field is

```
structName(index).nestedStructName(index).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)`.

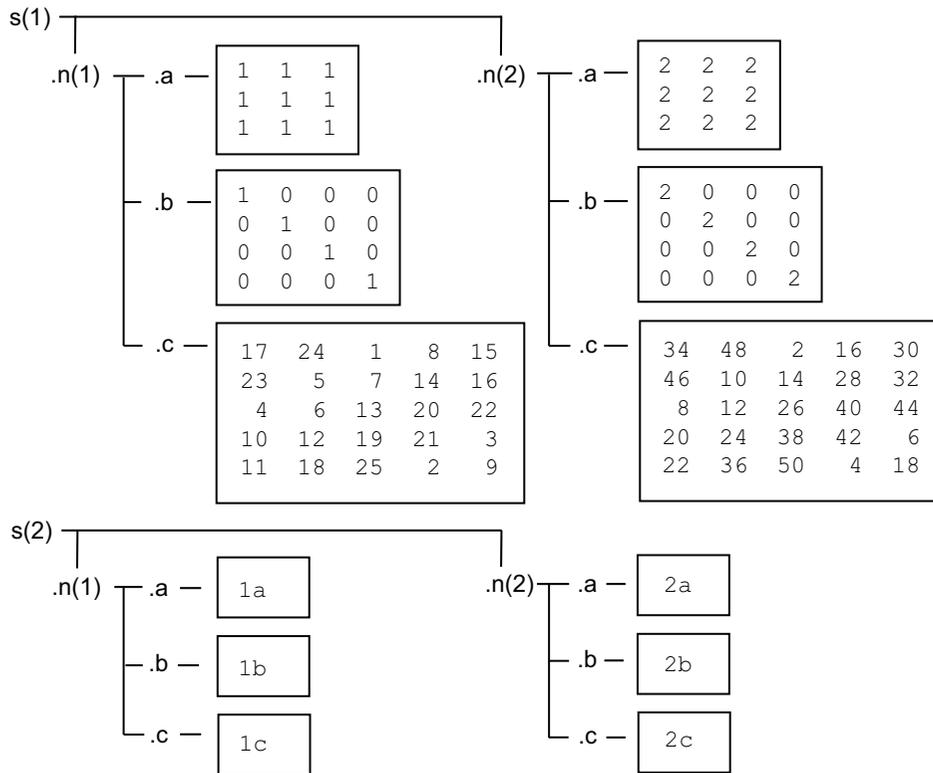
```
third_row_b =  
    0     0     1     0
```

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 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 2-74
- “Anonymous Functions” on page 20-24

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.

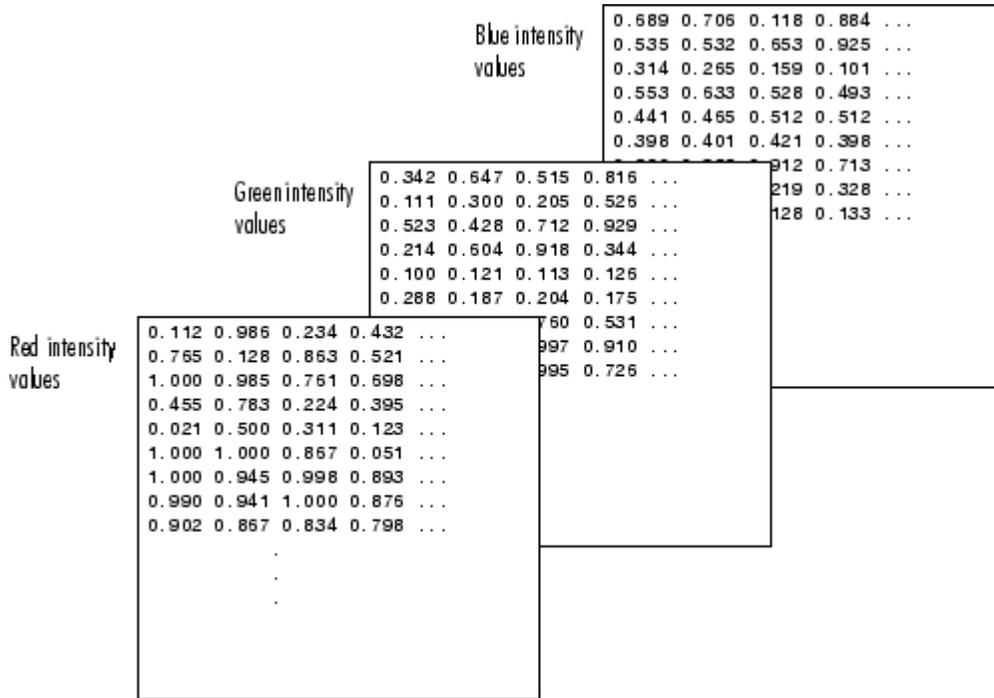
Plane organization allows easier access to all values within a field. Element-by-element organization allows easier access to all information related to a single element or record. The following sections include an example of each type of organization:

- “Plane Organization” on page 11-18
- “Element-by-Element Organization” on page 11-20

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 “How MATLAB Allocates Memory” on page 29-12.

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;
```

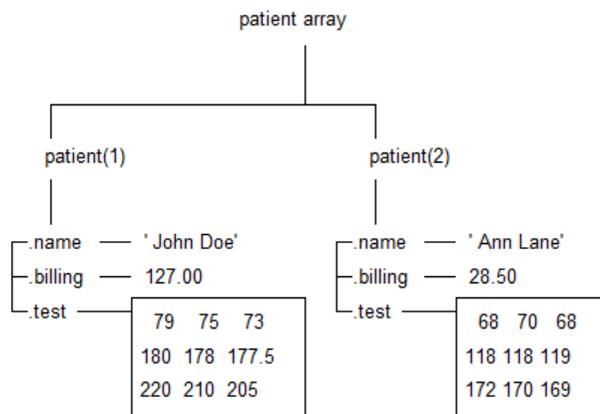
Plane organization allows you to easily extract entire image planes for display, filtering, or other processing. For example, multiply the red intensity values by 0.9:

```
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.



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 11-7.

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:

- “Reshaping and Rearranging Arrays”
- “How MATLAB Allocates Memory” on page 29-12

Cell Arrays

- “What Is a Cell Array?” on page 12-2
- “Create Cell Array” on page 12-3
- “Access Data in Cell Array” on page 12-5
- “Add Cells to Cell Array” on page 12-8
- “Delete Data from Cell Array” on page 12-10
- “Combine Cell Arrays” on page 12-12
- “Pass Contents of Cell Arrays to Functions” on page 12-13
- “Preallocate Memory for Cell Array” on page 12-18
- “Cell vs. Struct Arrays” on page 12-19
- “Multilevel Indexing to Access Parts of Cells” on page 12-25

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 pieces of text, 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 Cell Array” on page 12-3
- “Access Data in Cell Array” on page 12-5

Create 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}}
```

```
myCell = 2x3 cell array
    {[ 1]}    {[ 2]}    {[ 3]}
    {'text'}  {5x10x2 double}  {3x1 cell}
```

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.

You also can use the `{}` operator to create an empty 0-by-0 cell array.

```
C = {}
```

```
C =
```

```
0x0 empty cell array
```

To add values to a cell array over time or in a loop, create an empty N-dimensional array using the `cell` function.

```
emptyCell = cell(3,4,2)
```

```
emptyCell = 3x4x2 cell array
emptyCell(:,:,1) =
```

```
{0x0 double}    {0x0 double}    {0x0 double}    {0x0 double}
{0x0 double}    {0x0 double}    {0x0 double}    {0x0 double}
{0x0 double}    {0x0 double}    {0x0 double}    {0x0 double}
```

```
emptyCell(:,:,2) =
```

```
{0x0 double}    {0x0 double}    {0x0 double}    {0x0 double}
{0x0 double}    {0x0 double}    {0x0 double}    {0x0 double}
```

```
{0x0 double} {0x0 double} {0x0 double} {0x0 double}
```

`emptyCell` is a 3-by-4-by-2 cell array, where each cell contains an empty array, [].

See Also

`cell`

Related Examples

- “Access Data in Cell Array” on page 12-5
- “Create Structure Array” on page 11-2
- “Create and Work with Tables” on page 9-2

More About

- “Cell vs. Struct Arrays” on page 12-19
- “Advantages of Using Tables” on page 9-64

Access Data in Cell Array

This example shows how to read and write data to and from a cell array.

Create a 2-by-3 cell array of text and numeric data.

```
C = {'one', 'two', 'three';
     1, 2, 3}

C = 2x3 cell array
    {'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, to create a 2-by-2 cell array that is a subset of `C`, use smooth parentheses.

```
upperLeft = C(1:2,1:2)

upperLeft = 2x2 cell array
    {'one'}    {'two'}
    {[ 1]}    {[ 2]}
```

Update sets of cells by replacing them with the same number of cells. For example, replace cells in the first row of `C` with an equivalent-sized (1-by-3) cell array.

```
C(1,1:3) = {'first', 'second', 'third'}

C = 2x3 cell array
    {'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)
```

```
numericCells = 1x3 cell array
    {[1]}    {[2]}    {[3]}

numericVector = cell2mat(numericCells)

numericVector = 1x3
    1     2     3
```

`numericCells` is a 1-by-3 cell array, but `numericVector` is a 1-by-3 array of type `double`.

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, to access the contents of the last cell of `C`, use curly braces.

```
last = C{2,3}
```

```
last = 3
```

`last` is a numeric variable of type `double`, because the cell contains a `double` value.

Similarly, you can index with curly braces to replace the contents of a cell.

```
C{2,3} = 300
```

```
C = 2x3 cell array
    {'first'}    {'second'}    {'third'}
    {[ 1]}      {[ 2]}      {[ 300]}
```

You can access the contents of multiple cells by indexing with curly braces. MATLAB® returns the contents of the cells as 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.

Assign contents of four cells of `C` to four variables.

```
[r1c1, r2c1, r1c2, r2c2] = C{1:2,1:2}
```

```
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.

Concatenate the contents of the second row into a numeric array.

```
nums = [C{2,:}]  
nums = 1×3  
      1      2    300
```

See Also

`cell` | `cell2mat`

Related Examples

- “Create Cell Array” on page 12-3
- “Multilevel Indexing to Access Parts of Cells” on page 12-25
- “Comma-Separated Lists” on page 2-74

Add Cells to Cell Array

This example shows how to add cells to a cell array.

Create a 1-by-3 cell array.

```
C = {1, 2, 3}
```

```
C = 1x3 cell array
    {[1]}    {[2]}    {[3]}
```

Assign data to a cell outside the current dimensions. MATLAB® expands the cell array to a rectangle that includes the specified subscripts. Any intervening cells contain empty arrays.

```
C{4,4} = 44
```

```
C = 4x4 cell array
    {[      1]}    {[      2]}    {[      3]}    {0x0 double}
    {0x0 double}    {0x0 double}    {0x0 double}    {0x0 double}
    {0x0 double}    {0x0 double}    {0x0 double}    {0x0 double}
    {0x0 double}    {0x0 double}    {0x0 double}    {[      44]}
```

Add cells without specifying a value by assigning an empty array as the contents of a cell. C is now a 5-by-5 cell array.

```
C{5,5} = []
```

```
C = 5x5 cell array
Columns 1 through 4

    {[      1]}    {[      2]}    {[      3]}    {0x0 double}
    {0x0 double}    {0x0 double}    {0x0 double}    {0x0 double}
    {0x0 double}    {0x0 double}    {0x0 double}    {0x0 double}
    {0x0 double}    {0x0 double}    {0x0 double}    {[      44]}
    {0x0 double}    {0x0 double}    {0x0 double}    {0x0 double}

Column 5

    {0x0 double}
    {0x0 double}
    {0x0 double}
```

```
{0x0 double}  
{0x0 double}
```

See Also

Related Examples

- “Access Data in Cell Array” on page 12-5
- “Combine Cell Arrays” on page 12-12
- “Delete Data from Cell Array” on page 12-10

Delete Data from Cell Array

This example shows how to remove data from individual cells, and how to delete entire cells from a cell array.

Create a 3-by-3 cell array

```
C = {1, 2, 3; 4, 5, 6; 7, 8, 9}
```

```
C = 3x3 cell array
    {[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} = []
```

```
C = 3x3 cell array
    {[1]}    {[    2]}    {[3]}
    {[4]}    {0x0 double}    {[6]}
    {[7]}    {[    8]}    {[9]}
```

Delete sets of cells using standard array indexing with smooth parentheses, `()`. For example, remove the second row of `C`.

```
C(2,:) = []
```

```
C = 2x3 cell array
    {[1]}    {[2]}    {[3]}
    {[7]}    {[8]}    {[9]}
```

See Also

Related Examples

- “Add Cells to Cell Array” on page 12-8

- “Access Data in Cell Array” on page 12-5

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}
```

To combine cell arrays of character vectors into one character vector, use the `strjoin` function.

See Also

`strjoin`

Related Examples

- “Creating, Concatenating, and Expanding Matrices”

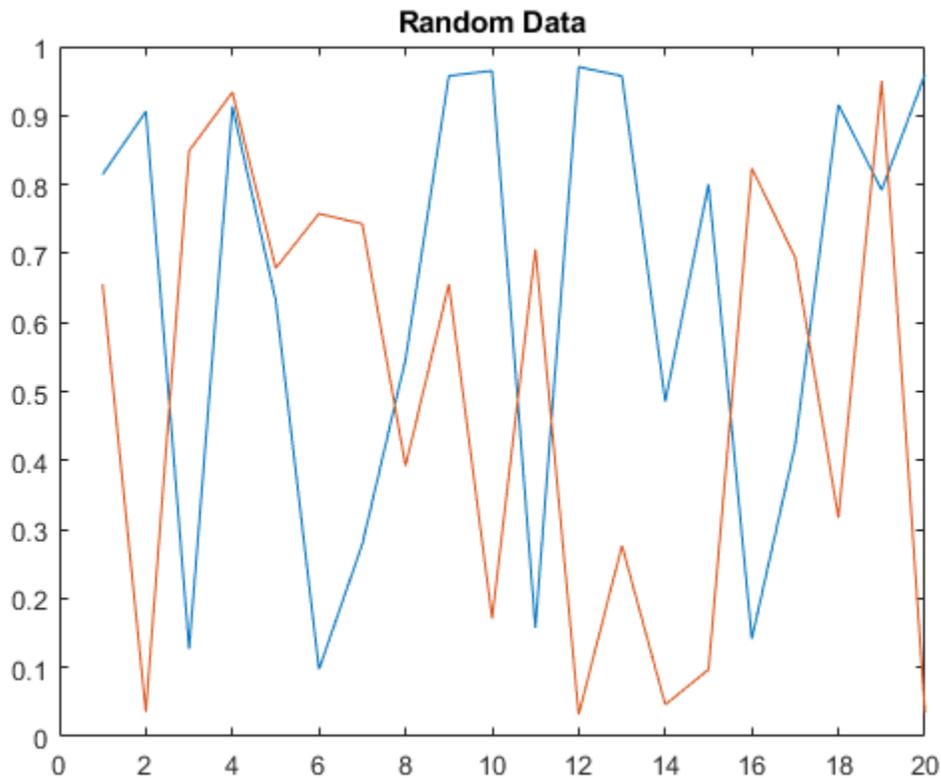
Pass Contents of Cell Arrays to Functions

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, {}.

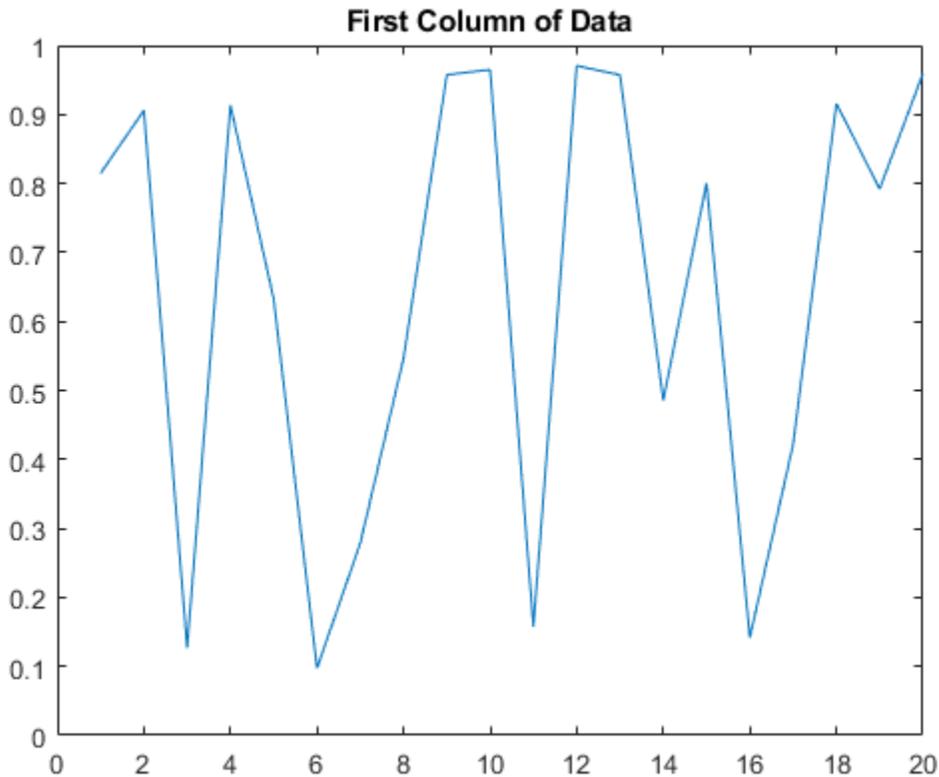
This example creates a cell array that contains text and a 20-by-2 array of random numbers.

```
randCell = {'Random Data', rand(20,2)};  
plot(randCell{1,2})  
title(randCell{1,1})
```



Plot only the first column of data by indexing further into the content (multilevel indexing).

```
figure  
plot(randCell{1,2}{:,1})  
title('First Column of Data')
```

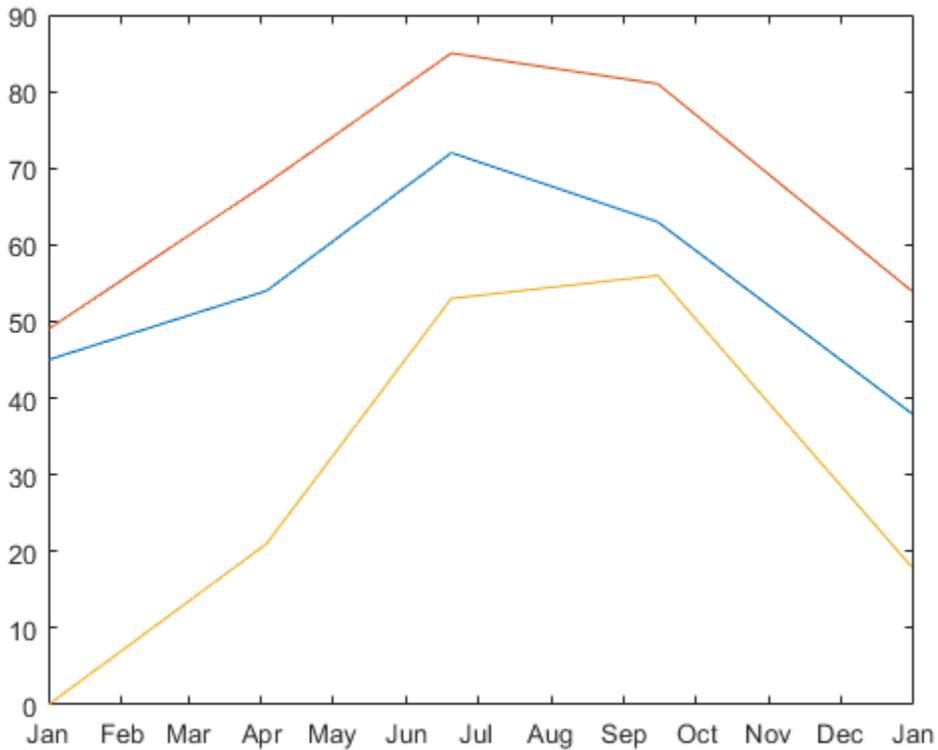


Combine numeric data from multiple cells using the cell2mat function.

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')
```



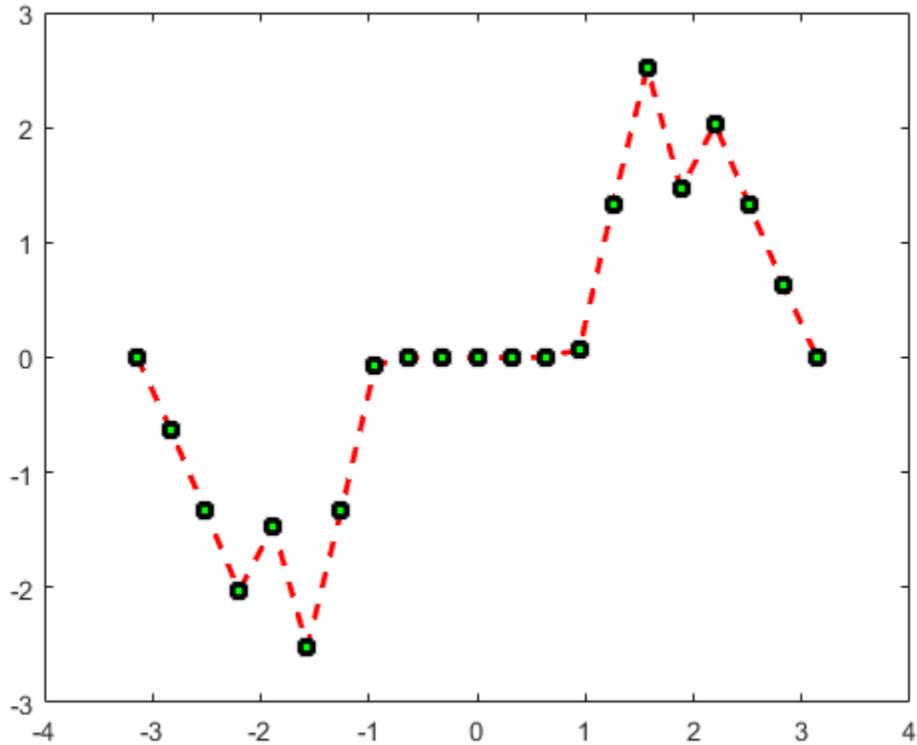
Pass the contents of multiple cells as a comma-separated list to functions that accept multiple inputs.

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{:})
```



See Also

More About

- “Access Data in Cell Array” on page 12-5
- “Multilevel Indexing to Access Parts of Cells” on page 12-25
- “Comma-Separated Lists” on page 2-74

Preallocate Memory for 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.

See Also

`cell`

Related Examples

- “Reshaping and Rearranging Arrays”
- “How MATLAB Allocates Memory” on page 29-12

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];
```

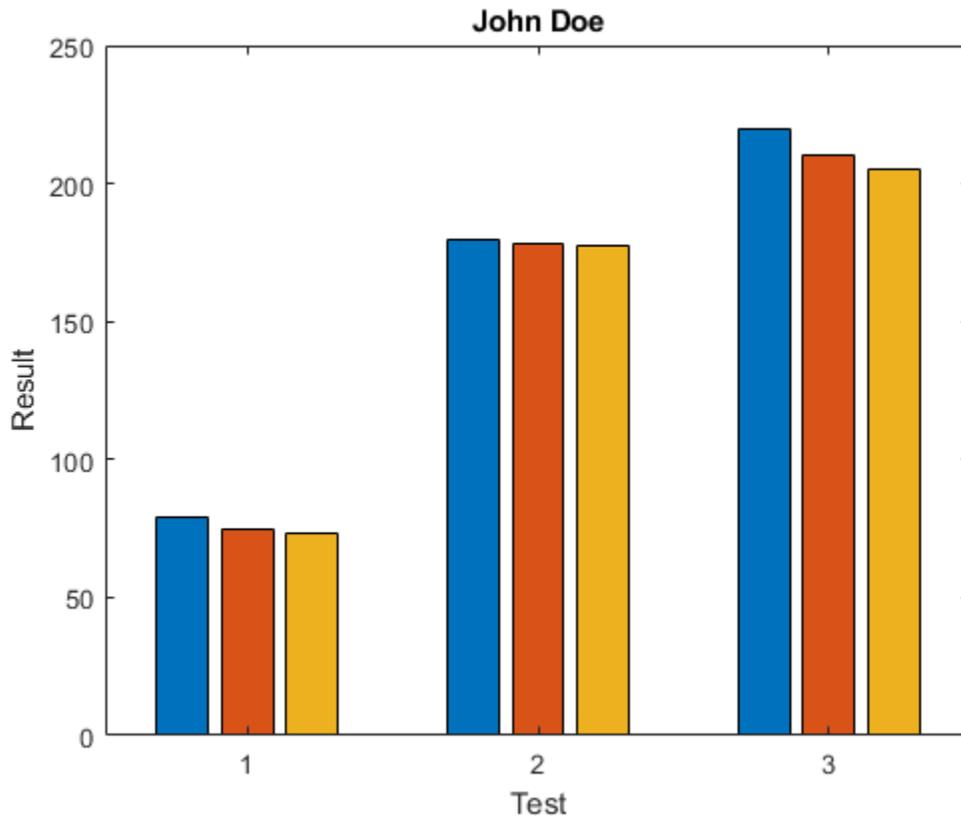
```
patient(2).name = 'Ann Lane';  
patient(2).billing = 28.50;  
patient(2).test = [68, 70, 68; 118, 118, 119; 172, 170, 169];
```

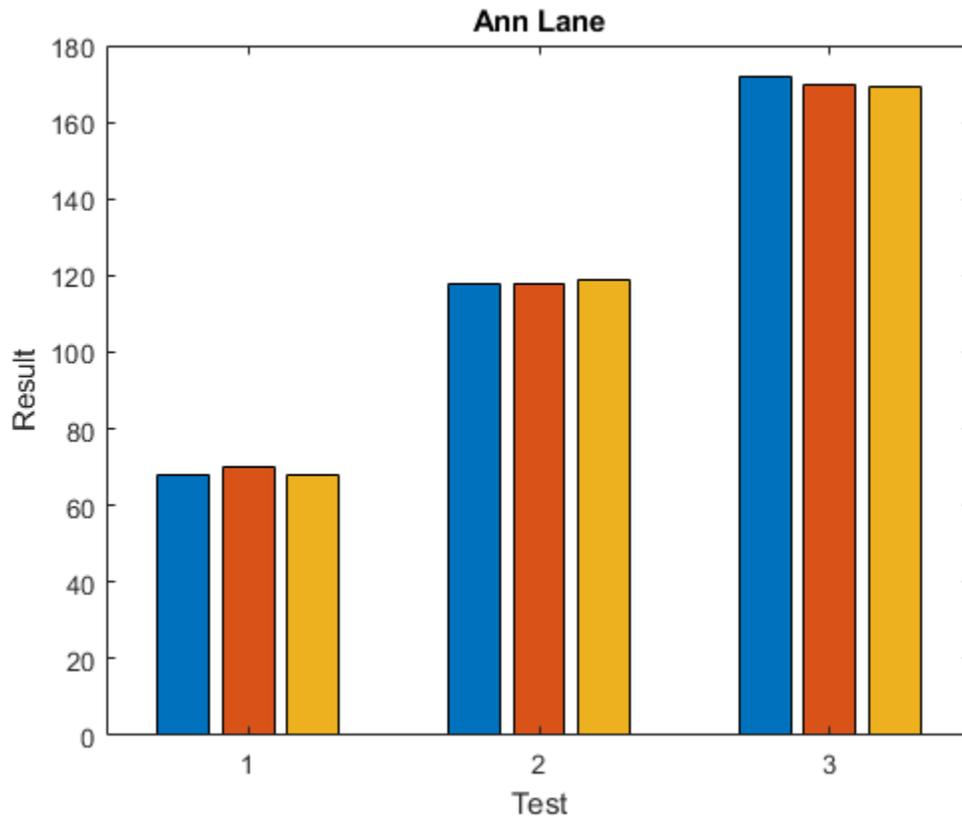
```
patient
```

```
patient = 1x2 struct array with fields:  
    name  
    billing  
    test
```

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)  
    xlabel('Test')  
    ylabel('Result')  
end
```





Cell Arrays

Cell arrays contain data in cells that you access by numeric indexing. Common applications of cell arrays include storing separate pieces of text and storing heterogeneous data from spreadsheets.

For example, store temperature data for three cities over time in a cell array.

```
temperature(1,:) = {'2009-12-31', [45, 49, 0]};  
temperature(2,:) = {'2010-04-03', [54, 68, 21]};  
temperature(3,:) = {'2010-06-20', [72, 85, 53]};  
temperature(4,:) = {'2010-09-15', [63, 81, 56]};  
temperature(5,:) = {'2010-12-09', [38, 54, 18]};
```

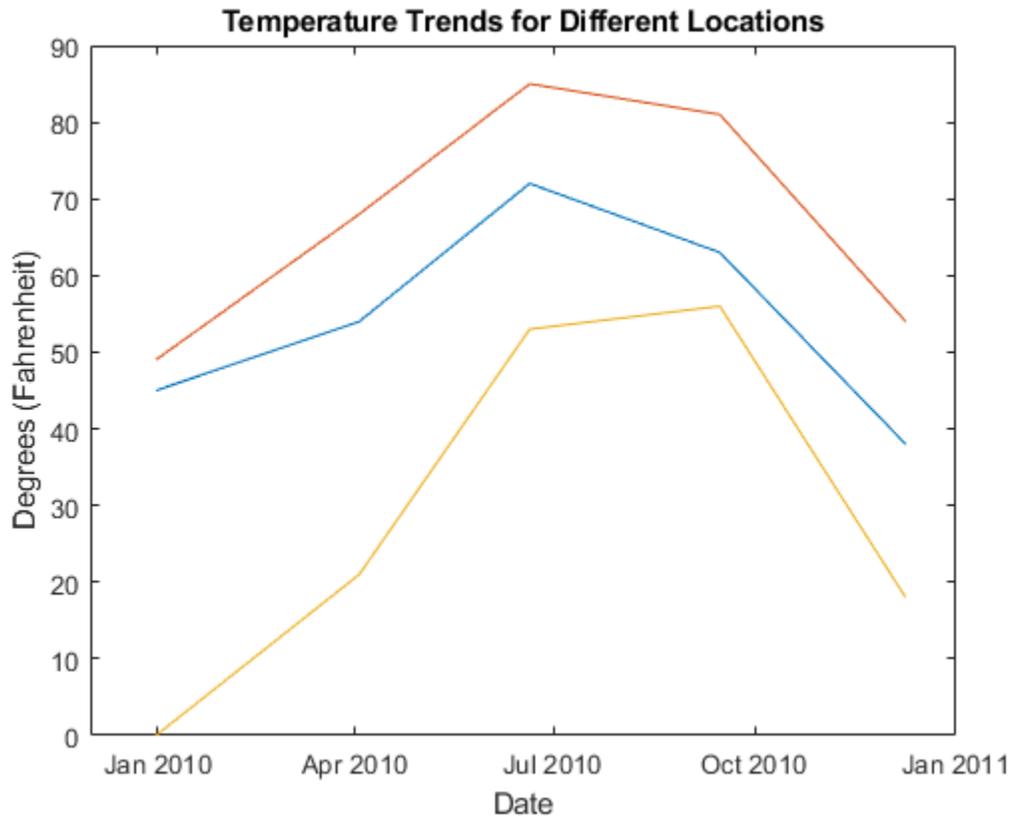
temperature

```
temperature = 5x2 cell array
    {'2009-12-31'}    {1x3 double}
    {'2010-04-03'}    {1x3 double}
    {'2010-06-20'}    {1x3 double}
    {'2010-09-15'}    {1x3 double}
    {'2010-12-09'}    {1x3 double}
```

Plot the temperatures for each city by date.

```
allTemps = cell2mat(temperature(:,2));
dates = datetime(temperature(:,1));

plot(dates,allTemps)
title('Temperature Trends for Different Locations')
xlabel('Date')
ylabel('Degrees (Fahrenheit)')
```



Other Container Arrays

Struct and cell arrays are the most commonly used containers for storing heterogeneous data. Tables are convenient for storing heterogeneous column-oriented or tabular data. Alternatively, use map containers, or create your own class.

See Also

`cell` | `cell2mat` | `containers.Map` | `datetime` | `plot` | `struct` | `table`

Related Examples

- “Access Data in Cell Array” on page 12-5
- “Access Data in a Structure Array” on page 11-7
- “Access Data in a Table” on page 9-42

More About

- “Advantages of Using Tables” on page 9-64

Multilevel Indexing to Access Parts of Cells

This example shows techniques for accessing data in arrays stored within cells of cell arrays.

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}

C = 2x2 cell array
     {1x3 double}      {1x3 double}
     {1x2 cell }      {1x1 struct}
```

Access the complete contents of a particular cell using curly braces, {}. For example, return a numeric vector from the cell that contains it.

```
C{1,2}

ans = 1x3

     100     200     300
```

Access part of the contents of a cell by appending indices, using syntax that matches the data type of the contents.

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 using smooth parentheses.

```
C{1,1}(1,2)

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 using curly braces.

```
C{2,1}{1,2}
```

```
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 using dot notation and smooth parentheses.

```
C{2,2}.Field2(5,1)
```

```
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'}) );
```

Access parts of the new data using curly braces, smooth parentheses, or dot notation.

```
copy_pi = C{2,1}{2,2}{1,1}
```

```
copy_pi = 3.1416
```

```
part_magic = C{2,2}.Field3.NestedField2(1:2,1:2)
```

```
part_magic = 2×2
```

```
    16     2  
     5    11
```

```
nested_cell = C{2,2}.Field3.NestedField3{2,1}
```

```
nested_cell =  
'more text'
```

See Also

Related Examples

- “Access Data in Cell Array” on page 12-5

Function Handles

- “Create Function Handle” on page 13-2
- “Pass Function to Another Function” on page 13-6
- “Call Local Functions Using Function Handles” on page 13-8
- “Compare Function Handles” on page 13-10

Create Function Handle

In this section...

“What Is a Function Handle?” on page 13-2

“Creating Function Handles” on page 13-2

“Anonymous Functions” on page 13-4

“Arrays of Function Handles” on page 13-4

“Saving and Loading Function Handles” on page 13-5

You can create function handles to named and anonymous functions. You can store multiple function handles in an array, and save and load them, as you would any other variable.

What Is a Function Handle?

A function handle is a MATLAB data type that stores an association to a function. Indirectly calling a function enables you to invoke the function regardless of where you call it from. Typical uses of function handles include:

- Pass a function to another function (often called *function functions*). For example, passing a function to integration and optimization functions, such as `integral` and `fzero`.
- Specify callback functions. For example, a callback that responds to a UI event or interacts with data acquisition hardware.
- Construct handles to functions defined inline instead of stored in a program file (anonymous functions).
- Call local functions from outside the main function.

You can see if a variable, `h`, is a function handle using `isa(h, 'function_handle')`.

Creating Function Handles

To create a handle for a function, precede the function name with an @ sign. For example, if you have a function called `myfunction`, create a handle named `f` as follows:

```
f = @myfunction;
```

You call a function using a handle the same way you call the function directly. For example, suppose that you have a function named `computeSquare`, defined as:

```
function y = computeSquare(x)
y = x.^2;
end
```

Create a handle and call the function to compute the square of four.

```
f = @computeSquare;
a = 4;
b = f(a)
```

```
b =
    16
```

If the function does not require any inputs, then you can call the function with empty parentheses, such as

```
h = @ones;
a = h()
```

```
a =
    1
```

Without the parentheses, the assignment creates another function handle.

```
a = h
a =
```

```
@ones
```

Function handles are variables that you can pass to other functions. For example, calculate the integral of x^2 on the range $[0,1]$.

```
q = integral(f,0,1);
```

Function handles store their absolute path, so when you have a valid handle, you can invoke the function from any location. You do not have to specify the path to the function when creating the handle, only the function name.

Keep the following in mind when creating handles to functions:

- Name length — Each part of the function name (including package and class names) must be less than the number specified by `namelengthmax`. Otherwise, MATLAB truncates the latter part of the name.
- Scope — The function must be in scope at the time you create the handle. Therefore, the function must be on the MATLAB path or in the current folder. Or, for handles to local or nested functions, the function must be in the current file.
- Precedence — When there are multiple functions with the same name, MATLAB uses the same precedence rules to define function handles as it does to call functions. For more information, see “Function Precedence Order” on page 20-43.
- Overloading — If the function you specify overloads a function in a class that is not a fundamental MATLAB class, the function is not associated with the function handle at the time it is constructed. Instead, MATLAB considers the input arguments and determines which implementation to call at the time of evaluation.

Anonymous Functions

You can create handles to anonymous functions. An anonymous function is a one-line expression-based MATLAB function that does not require a program file. Construct a handle to an anonymous function by defining the body of the function, `anonymous_function`, and a comma-separated list of input arguments to the anonymous function, `arglist`. The syntax is:

```
h = @(arglist)anonymous_function
```

For example, create a handle, `sqr`, to an anonymous function that computes the square of a number, and call the anonymous function using its handle.

```
sqr = @(n) n.^2;  
x = sqr(3)
```

```
x =
```

```
9
```

For more information, see “Anonymous Functions” on page 20-24.

Arrays of Function Handles

You can create an array of function handles by collecting them into a cell or structure array. For example, use a cell array:

```
C = {@sin, @cos, @tan};  
C{2}(pi)
```

```
ans =
```

```
-1
```

Or use a structure array:

```
S.a = @sin; S.b = @cos; S.c = @tan;  
S.a(pi/2)
```

```
ans =
```

```
1
```

Saving and Loading Function Handles

You can save and load function handles in MATLAB, as you would any other variable. In other words, use the `save` and `load` functions. If you save a function handle, MATLAB does not save the path information. If you load a function handle, and the function file no longer exists on the path, the handle is invalid. An invalid handle occurs if the file location or file name has changed since you created the handle. If a handle is invalid, MATLAB might display a warning when you load the file. When you invoke an invalid handle, MATLAB issues an error.

See Also

`func2str` | `functions` | `isa` | `str2func`

Related Examples

- “Pass Function to Another Function” on page 13-6

More About

- “Anonymous Functions” on page 20-24

Pass Function to Another Function

You can use function handles as input arguments to other functions, which are called *function functions*. These functions evaluate mathematical expressions over a range of values. Typical function functions include `integral`, `quad2d`, `fzero`, and `fminbnd`.

For example, to find the integral of the natural log from 0 through 5, pass a handle to the `log` function to `integral`.

```
a = 0;  
b = 5;  
q1 = integral(@log,a,b)
```

```
q1 = 3.0472
```

Similarly, to find the integral of the `sin` function and the `exp` function, pass handles to those functions to `integral`.

```
q2 = integral(@sin,a,b)
```

```
q2 = 0.7163
```

```
q3 = integral(@exp,a,b)
```

```
q3 = 147.4132
```

Also, you can pass a handle to an anonymous function to function functions. An anonymous function is a one-line expression-based MATLAB® function that does not require a program file. For example, evaluate the integral of $x/(e^x - 1)$ on the range `[0, Inf]`:

```
fun = @(x)x./(exp(x)-1);  
q4 = integral(fun,0,Inf)
```

```
q4 = 1.6449
```

Functions that take a function as an input (called *function functions*) expect that the function associated with the function handle has a certain number of input variables. For example, if you call `integral` or `fzero`, the function associated with the function handle must have exactly one input variable. If you call `integral3`, the function associated with the function handle must have three input variables. For information on calling function functions with more variables, see “Parameterizing Functions”.

See Also

Related Examples

- “Create Function Handle” on page 13-2
- “Parameterizing Functions”

More About

- “Anonymous Functions” on page 20-24

Call Local Functions Using Function Handles

This example shows how to create handles to local functions. If a function returns handles to local functions, you can call the local functions outside of the main function. This approach allows you to have multiple, callable functions in a single file.

Create the following function in a file, `ellipseVals.m`, in your working folder. The function returns a struct with handles to the local functions.

```
% Copyright 2015 The MathWorks, Inc.

function fh = ellipseVals
    fh.focus = @computeFocus;
    fh.eccentricity = @computeEccentricity;
    fh.area = @computeArea;
end

function f = computeFocus(a,b)
    f = sqrt(a^2-b^2);
end

function e = computeEccentricity(a,b)
    f = computeFocus(a,b);
    e = f/a;
end

function ae = computeArea(a,b)
    ae = pi*a*b;
end
```

Invoke the function to get a struct of handles to the local functions.

```
h = ellipseVals

h =

    struct with fields:
        focus: @computeFocus
        eccentricity: @computeEccentricity
```

```
area: @computeArea
```

Call a local function using its handle to compute the area of an ellipse.

```
h.area(3,1)
```

```
ans =
```

```
9.4248
```

Alternatively, you can use the `localfunctions` function to create a cell array of function handles from all local functions automatically. This approach is convenient if you expect to add, remove, or modify names of the local functions.

See Also

`localfunctions`

Related Examples

- “Create Function Handle” on page 13-2

More About

- “Local Functions” on page 20-30

Compare Function Handles

Compare Handles Constructed from Named Function

MATLAB® considers function handles that you construct from the same named function to be equal. The `isequal` function returns a value of `true` when comparing these types of handles.

```
fun1 = @sin;  
fun2 = @sin;  
isequal(fun1, fun2)
```

```
ans =  
  
    logical  
  
     1
```

If you save these handles to a MAT-file, and then load them back into the workspace, they are still equal.

Compare Handles to Anonymous Functions

Unlike handles to named functions, function handles that represent the same anonymous function are not equal. They are considered unequal because MATLAB cannot guarantee that the frozen values of nonargument variables are the same. For example, in this case, `A` is a nonargument variable.

```
A = 5;  
h1 = @(x)A * x.^2;  
h2 = @(x)A * x.^2;  
isequal(h1, h2)
```

```
ans =  
  
    logical  
  
     0
```

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 =
     logical
     1
```

Compare 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 function. This function 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 =
     logical
     1
```

Function handles constructed from different calls are not considered equal.

```
[q1,q2] = test_eq(4,19,-7);  
isequal(h1,q1)
```

```
ans =
```

```
logical
```

```
0
```

See Also

isequal

Related Examples

- “Create Function Handle” on page 13-2

Map Containers

- “Overview of Map Data Structure” on page 14-2
- “Description of Map Class” on page 14-4
- “Create Map Object” on page 14-6
- “Examine Contents of Map” on page 14-9
- “Read and Write Using Key Index” on page 14-11
- “Modify Keys and Values in Map” on page 14-16
- “Map to Different Value Types” on page 14-19

Overview of 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 vector.

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 character vector 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, character arrays, 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.

See Also

`containers.Map` | `keys` | `values`

Related Examples

- “Description of Map Class” on page 14-4
- “Create Map Object” on page 14-6
- “Examine Contents of Map” on page 14-9

Description of Map Class

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 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 <code>Map</code> object.	0
KeyType	Character vector that indicates the type of all keys contained in the <code>Map</code> object. <code>KeyType</code> 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	Character vector that indicates the type of values contained in the <code>Map</code> object. If the values in a <code>Map</code> are all scalar numbers of the same type, <code>ValueType</code> is set to that type. If the values are all character arrays, <code>ValueType</code> is <code>'char'</code> . Otherwise, <code>ValueType</code> 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 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

See Also

`containers.Map` | `isKey` | `keys` | `length` | `remove` | `size` | `values`

Related Examples

- “Overview of Map Data Structure” on page 14-2
- “Create Map Object” on page 14-6
- “Examine Contents of Map” on page 14-9

Create Map Object

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)
```

Construct 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 =
```

```
Map with properties:
```

```
    Count: 0  
    KeyType: char  
    ValueType: any
```

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 Map Class” on page 14-5

Construct 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 vectors, be sure that you specify them enclosed within single quotation marks. For example, when constructing a Map that has character vectors as keys, use

```
mapObj = containers.Map(...
    {'keyst1', 'keyst2', ...}, {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 =
```

Map with properties:

```
Count: 13  
KeyType: char  
ValueType: double
```

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`.

Combine 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 “Build Map with Concatenation” on page 14-13, below.

See Also

`containers.Map` | `keys` | `values`

Related Examples

- “Overview of Map Data Structure” on page 14-2
- “Description of Map Class” on page 14-4
- “Examine Contents of Map” on page 14-9

Examine Contents of 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 `ticketMap` 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'
```

See Also

`containers.Map` | `isKey` | `keys` | `length` | `remove` | `size` | `values`

Related Examples

- “Create Map Object” on page 14-6
- “Read and Write Using Key Index” on page 14-11
- “Modify Keys and Values in Map” on page 14-16
- “Map to Different Value Types” on page 14-19

Read and Write Using Key Index

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.

Read From 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 vector, enclose it in single quotation marks:

```
valueN = mapObj(keyN);
```

Start with the Map `ticketMap` :

```
ticketMap = containers.Map(...
    {'2R175', 'B7398', 'A479GY', 'NZ1452'}, ...
    {'James Enright', 'Carl Haynes', 'Sarah Latham', ...
     'Bradley Reid'});
```

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(:)
```

Add 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;
```

Start with the Map `ticketMap`:

```
ticketMap = containers.Map(...  
    {'2R175', 'B7398', 'A479GY', 'NZ1452'}, ...  
    {'James Enright', 'Carl Haynes', 'Sarah Latham', ...  
    'Bradley Reid'});
```

Add two more entries to the `ticketMap` Map. Verify that `ticketMap` now has six key/value pairs:

```
ticketMap('947F4') = 'Susan Spera';  
ticketMap('417R93') = 'Patricia Hughes';  
  
ticketMap.Count  
  
ans =
```

List all of the keys and values in `ticketMap`:

```
keys(ticketMap), values(ticketMap)

ans =

    '2R175'    '417R93'    '947F4'    'A479GY'    'B7398'    'NZ1452'

ans =

    'James Enright'    'Patricia Hughes'    'Susan Spera'    'Sarah Latham'    'Carl Haynes'    'Bradley Reid'
```

Build 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 Map objects, 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 Map objects. 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 Map objects:

```
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 =  
  'James Enright'  'Patricia Hughes'  'Susan Spera'  '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'
```

See Also

`containers.Map` | `isKey` | `keys` | `values`

Related Examples

- “Create Map Object” on page 14-6
- “Examine Contents of Map” on page 14-9

- “Modify Keys and Values in Map” on page 14-16
- “Map to Different Value Types” on page 14-19

Modify Keys and Values in Map

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 “Modify Copy of Map” on page 14-17, below.

Remove Keys and Values from 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');
```

Start with the Map `ticketMap` :

```
ticketMap = containers.Map(...  
    {'2R175', 'B7398', 'A479GY', 'NZ1452'}, ...  
    {'James Enright', 'Carl Haynes', 'Sarah Latham', ...  
    'Bradley Reid'});
```

Remove one entry (the specified key and its value) from the Map object:

```
remove(ticketMap, 'NZ1452');  
values(ticketMap)  
  
ans =  
  
    'James Enright'    'Sarah Latham'    'Carl Haynes'
```

Modify 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
```

Modify 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.
```

Modify Copy of 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 the `ticketMap` Map. 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;
```

See Also

`containers.Map` | `isKey` | `keys` | `length` | `remove` | `size` | `values`

Related Examples

- “Create Map Object” on page 14-6
- “Examine Contents of Map” on page 14-9
- “Read and Write Using Key Index” on page 14-11
- “Map to Different Value Types” on page 14-19

Map to Different Value Types

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.

Map to Structure Array

The following example maps airline seat numbers to structures that contain ticket numbers and destinations. Start with the Map `ticketMap`, which maps ticket numbers to passengers:

```
ticketMap = containers.Map(...
    {'2R175', 'B7398', 'A479GY', 'NZ1452'}, ...
    {'James Enright', 'Carl Haynes', 'Sarah Latham', ...
     'Bradley Reid'});
```

Then create the following structure array, containing ticket numbers and destinations:

```
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 seats to 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'
```

Using `ticketMap` and `seatingMap` together, you can find the name of the person who has reserved seat 15B:

```
ticket = seatingMap('15B').ticketNum;  
passenger = ticketMap(ticket)  
  
passenger =  
  
Sarah Latham
```

Map to 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.
```

See Also

`cell` | `containers.Map` | `isKey` | `keys` | `struct` | `values`

Related Examples

- “Create Map Object” on page 14-6
- “Create Structure Array” on page 11-2
- “Create Cell Array” on page 12-3
- “Examine Contents of Map” on page 14-9
- “Read and Write Using Key Index” on page 14-11
- “Modify Keys and Values in Map” on page 14-16

Combining Unlike Classes

- “Valid Combinations of Unlike Classes” on page 15-2
- “Combining Unlike Integer Types” on page 15-3
- “Combining Integer and Noninteger Data” on page 15-5
- “Combining Cell Arrays with Non-Cell Arrays” on page 15-6
- “Empty Matrices” on page 15-7
- “Concatenation Examples” on page 15-8

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.

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.

See Also

More About

- “Combining Unlike Integer Types” on page 15-3
- “Combining Integer and Noninteger Data” on page 15-5
- “Combining Cell Arrays with Non-Cell Arrays” on page 15-6
- “Concatenation Examples” on page 15-8

Combining Unlike Integer Types

In this section...

“Overview” on page 15-3

“Example of Combining Unlike Integer Sizes” on page 15-3

“Example of Combining Signed with Unsigned” on page 15-4

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)]
```

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 15-8
“Combining Integer and Double Types” on page 15-8
“Combining Character and Double Types” on page 15-9
“Combining Logical and Double Types” on page 15-9

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    8

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

Object Behavior

In this section...

“Two Copy Behaviors” on page 16-2

“Handle Object Copy” on page 16-2

“Value Object Copy Behavior” on page 16-2

“Handle Object Copy Behavior” on page 16-3

“Testing for Handle or Value Class” on page 16-6

Two Copy Behaviors

There are two fundamental kinds of MATLAB objects — handles and values.

Value objects behave like MATLAB fundamental types with respect to copy operations. Copies are independent values. Operations that you perform on one object do not affect copies of that object.

Handle objects are referenced by their handle variable. Copies of the handle variable refer to the same object. Operations that you perform on a handle object are visible from all handle variables that reference that object.

Handle Object Copy

If you are defining classes and want to support handle object copy, see “Implement Copy for Handle Classes”.

Value Object Copy Behavior

MATLAB numeric variables are 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`. `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 as numeric variables. For example, suppose `vobj1` is a value object with property `a`:

```
vobj1.a = 8;
```

If you copy `vobj1` to `vobj2`, and then change the value of `vobj1` property `a`, 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
```

```
end  
end
```

Create an object of this class:

```
hobj1 = HdClass(8)
```

Because this statement is not terminated with a semicolon, MATLAB displays information about the object:

```
hobj1 =  
    HdClass with properties:  
    Data: 8
```

The variable `hobj1` is a handle that references the object created. Copying `hobj1` to `hobj2` results in another handle referring to the same object:

```
hobj2 = hobj1  
hobj2 =  
    HdClass with properties:  
    Data: 8
```

Because handles reference the object, copying a handle copies the handle to a new variable name, but the handle still refers to the same object. 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
```

Reassigning Handle Variables

Reassigning a handle variable produces the same result as reassigning any MATLAB variable. When you create an 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 remove the object referenced by the other handle. However, if there are no references to an object, MATLAB destroys the object.

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 with properties:
```

```
Data: 256
```

If you clear both `hobj1` and `hobj2`, then there are no references to the object. MATLAB destroys 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 refer to the same object, delete either handle. MATLAB deletes the object:

```
hobj1 = HdClass(8);
hobj2 = hobj1;
delete(hobj1)
hobj2

hobj2 =

    handle to deleted HdClass
```

Use `clear` to remove the variable from the workspace.

Modifying Objects

When you pass an object to a function, MATLAB passes a copy of the object into the function workspace. If the function modifies the object, MATLAB modifies only the copy of the object that is in the function workspace. The differences in copy behavior between handle and value classes are important in such cases:

- Value object — The function must return the modified copy of the object. To modify the object in the caller's workspace, assign the function output to a variable of the same name
- Handle object — The copy in the function workspace refers to the same object. Therefore, the function does not have to return the modified copy.

Testing for Handle or Value Class

To determine if an object is a handle object, use the `isa` function. If `obj` is an object of some class, this statement determines if `obj` is a handle:

```
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
```

Querying the class of `hobj` shows that it is a `containers.Map` object:

```
class(hobj)
```

```
ans =
```

```
containers.Map
```

The `class` function returns the specific class of an object.

See Also

Related Examples

- “Implement Copy for Handle Classes”

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 “Role of Classes in MATLAB”.

Scripts and Functions

Scripts

- “Create Scripts” on page 18-2
- “Add Comments to Programs” on page 18-4
- “Run Code Sections” on page 18-7
- “Scripts vs. Functions” on page 18-16
- “Add Functions to Scripts” on page 18-18

Create 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 or series of commands you have to reference.

You can create a new script in the following ways:

- Highlight commands from the Command History, right-click, and select **Create Script**.
- Click the **New Script**  button on the **Home** tab.
- Use the `edit` function. For example, `edit new_file_name` creates (if the file does not exist) and opens the file `new_file_name`. If `new_file_name` is unspecified, MATLAB opens a new file called `Untitled`.

After you create a script, you can add code to the script and save it. For example, you can save this code that generates random numbers from 0 through 100 as a script called `numGenerator.m`.

```
columns = 10000;  
rows = 1;  
bins = columns/100;  
  
rng(now);  
list = 100*rand(rows,columns);  
histogram(list,bins)
```

Save your script and run the code using either of these methods:

- Type the script name on the command line and press **Enter**. For example, to run the `numGenerator.m` script, type `numGenerator`.
- Click the **Run**  button on the **Editor** tab

You also can run the code from a second program file. To do this, add a line of code with the script name to the second program file. For example, to run the `numGenerator.m` script from a second program file, add the line `numGenerator;` to the file. MATLAB runs the code in `numGenerator.m` when you run the second file.

When execution of the script completes, the variables remain in the MATLAB workspace. In the `numGenerator.m` example, the variables `columns`, `rows`, `bins`, and `list` remain

in the workspace. To see a list of variables, type `whos` at the command prompt. Scripts share the base workspace with your interactive MATLAB session and with other scripts.

See Also

More About

- “Run Code Sections” on page 18-7
- “Scripts vs. Functions” on page 18-16
- “Base and Function Workspaces” on page 20-10
- “Create Live Scripts in the Live Editor” on page 19-7

Add Comments to Programs

When you write code, it is a good practice to add comments that describe the code. Comments allow others to understand your code and can refresh your memory when you return to it later. During program development and testing, you also can use comments to comment out any code that does not need to run.

In the Live Editor, you can insert lines of text before and after code to describe a process or code. Text lines provide additional flexibility such as standard formatting options, and the insertion of images, hyperlinks, and equations. For more information, see “Create Live Scripts in the Live Editor” on page 19-7.

Note When you have a MATLAB code file (.m) containing text that has characters in a different encoding than that of your platform, when you save or publish your file, MATLAB displays those characters as garbled text. Live scripts and functions (.mlx) support storing and displaying characters across all locales.

To add comments to MATLAB code, use the percent (%) symbol. 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.
```

To comment out multiple lines of code, use the block comment operators, %{ and %}. 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.

For example:

```
a = magic(3);  
%{  
sum(a)  
diag(a)  
sum(diag(a))  
%}  
sum(diag(fliplr(a)))
```

To comment out a selection, select the lines of code, go to the **Editor** or **Live Editor** tab, and press the  button. You also can type **Ctrl+R**. To uncomment the selected lines code, press the  button or type **Ctrl+T**.

To comment out part of a statement that spans multiple lines, use an ellipsis (. . .) instead of a percent sign. For example:

```
header = ['Last Name, ',      ...
         'First Name, ',   ...
         ... 'Middle Initial, ', ...
         'Title']
```

The MATLAB Editor includes tools and context menu items to help you add, remove, or change the format of comments for MATLAB, Java, and C/C++ code. For example, suppose that you have this lengthy text into a commented line.

```
% This is a program that has a comment that is a little more than 75 columns wide.
disp('Hello, world')
```

With the cursor on the line, go to **Editor** tab and in the **Edit** section, press the  button. The Editor wraps the comment:

```
% This is a program that has a comment that is a little more than 75
% columns wide.
disp('Hello, world')
```

The  button is not available in the Live Editor or in MATLAB Online.

By default, as you type comments in the Editor, the text wraps when it reaches a column width of 75. To change the column where the comment text wraps or to disable automatic comment wrapping, go to the **Home** tab and in the **Environment** section, click  **Preferences**. Select **MATLAB > Editor/Debugger > Language**, and adjust the **Comment formatting** preferences.

The Editor does not wrap comments with:

- Code section titles (comments that begin with %%)
- Long contiguous text, such as URLs
- Bulleted list items (text that begins with * or #) onto the preceding line

Changing the comment formatting preferences is not supported in MATLAB Online.

See Also

Related Examples

- “Add Help for Your Program” on page 20-6
- “Create Scripts” on page 18-2
- “Create Live Scripts in the Live Editor” on page 19-7
- “Editor/Debugger Preferences”

Run Code Sections

In this section...

- “Divide Your File into Code Sections” on page 18-7
- “Evaluate Code Sections” on page 18-7
- “Navigate Among Code Sections in a File” on page 18-9
- “Example of Evaluating Code Sections” on page 18-9
- “Change the Appearance of Code Sections” on page 18-12
- “Use Code Sections with Control Statements and Functions” on page 18-13

Divide Your File into Code Sections

MATLAB files often consist of many commands. You typically focus efforts on a single part of your program at a time, working with the code in chunks. Similarly, when explaining your files to others, often you describe your program in chunks. To facilitate these processes, use *code sections*, also known as code cells or cell mode. A code section contains contiguous lines of code that you want to evaluate as a group in a MATLAB script, beginning with two comment characters (%%).

To define code section boundaries explicitly, insert section breaks using these methods:

- On the **Editor** tab, in the **Edit** section, in the **Insert** button group, click .
- Enter two percent signs (%%) at the start of the line where you want to begin the new code section.

The text on the same line as %% is called the *section title*. Including section titles is optional, however, it improves the readability of the file and appears as a heading if you publish your code.

Evaluate Code Sections

As you develop a MATLAB file, you can use the Editor section features to evaluate the file section-by-section. This method helps you to experiment with and fine-tune your program. You can navigate among sections, and evaluate each section individually. To evaluate a section, it must contain all the values it requires, or the values must exist in the MATLAB workspace.

The section evaluation features run the section code currently highlighted in yellow. MATLAB does not automatically save your file when evaluating individual code sections. The file does not have to be on your search path.

This table provides instructions on evaluating code sections.

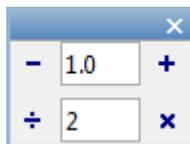
Operation	Instructions
Run the code in the current section.	<ul style="list-style-type: none"> Place the cursor in the code section. On the Editor tab, in the Run section, click  Run Section.
Run the code in the current section, and then move to the next section.	<ul style="list-style-type: none"> Place the cursor in the code section. On the Editor tab, in the Run section, click  Run and Advance.
Run all the code in the file.	<ul style="list-style-type: none"> Type the saved script name in the Command Window. On the Editor tab, in the Run section, click  Run.

Increment Values in Code Sections

You can increment numbers within a section, rerunning that section after every change. This helps you fine-tune and experiment with your code.

To increment or decrement a number in a section:

- 1 Highlight or place your cursor next to the number.
- 2 Right-click to open the context menu.
- 3 Select **Increment Value and Run Section**. A small dialog box appears.



- 4 Input appropriate values in the $- / +$ text box or \div / \times text box.
- 5 Click the $+$, $-$, \times , or \div button to add to, subtract from, multiply, or divide the selected number in your section.

MATLAB runs the section after every click.

Note MATLAB software does not automatically save changes you make to the numbers in your script.

Navigate Among Code Sections in a File

You can navigate among sections in a file without evaluating the code within those sections. This facilitates jumping quickly from section to section within a file. You might do this, for example, to find specific code in a large file.

Operation	Instructions
Move to the next section.	<ul style="list-style-type: none"> On the Editor tab, in the Run section, click  Advance.
Move to the previous section.	<ul style="list-style-type: none"> Press Ctrl + Up arrow.
Move to a specific section.	<ul style="list-style-type: none"> On the Editor tab, in the Navigate section, use the  Go To to move the cursor to a selected section.

Example of Evaluating Code Sections

This example defines two code sections in a file called `sine_wave.m` and then increments a parameter to adjust the created plot. To open this file in your Editor, run the following command, and then save the file to a local folder:

```
edit(fullfile(matlabroot, 'help', 'techdoc', 'matlab_env', ...
'examples', 'sine_wave.m'))
```

After the file is open in your Editor:

- 1 Insert a section break and the following title on the first line of the file.

```
%% Calculate and Plot Sine Wave
```
- 2 Insert a blank line and a second section break after `plot(x,y)`. Add a section title, **Modify Plot Properties**, so that the entire file contains this code:

```
%% Calculate and Plot Sine Wave
% Define the range for x.
```

```
% Calculate and plot y = sin(x).  
x = 0:1:6*pi;  
y = sin(x);  
plot(x,y)
```

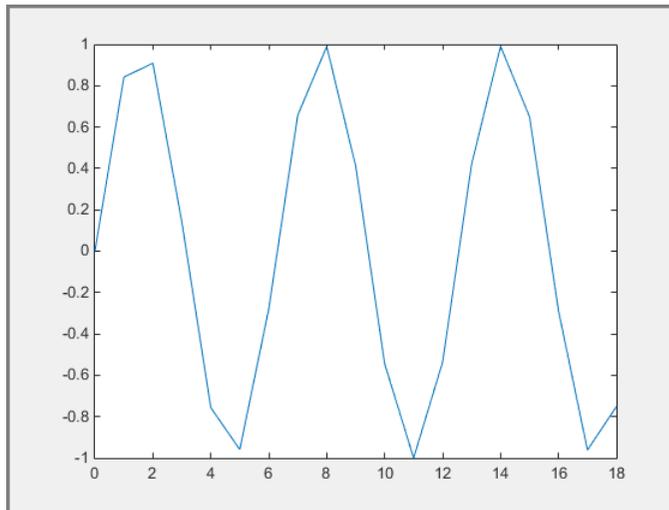
```
%% Modify Plot Properties  
title('Sine Wave')  
xlabel('x')  
ylabel('sin(x)')  
fig = gcf;  
fig.MenuBar = 'none';
```

3 Save the file.

4 Place your cursor in the section titled Calculate and Plot Sine Wave. On the

Editor tab, in the **Run** section, click  **Run Section**.

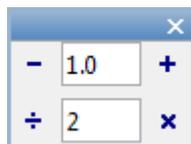
A figure displaying a course plot of $\sin(x)$ appears.



5 Smooth the sine plot.

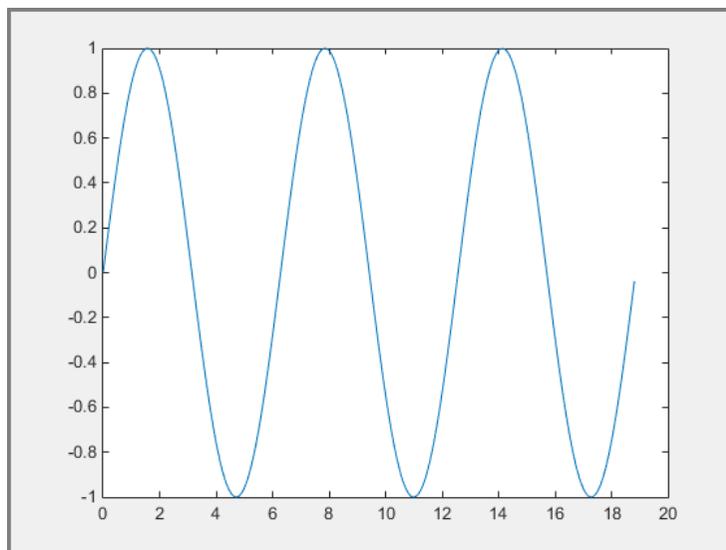
1 Highlight 1 in the statement: `x = 0:1:6*pi; .`

2 Right-click and select **Increment Value and Run Section**. A small dialog box appears.

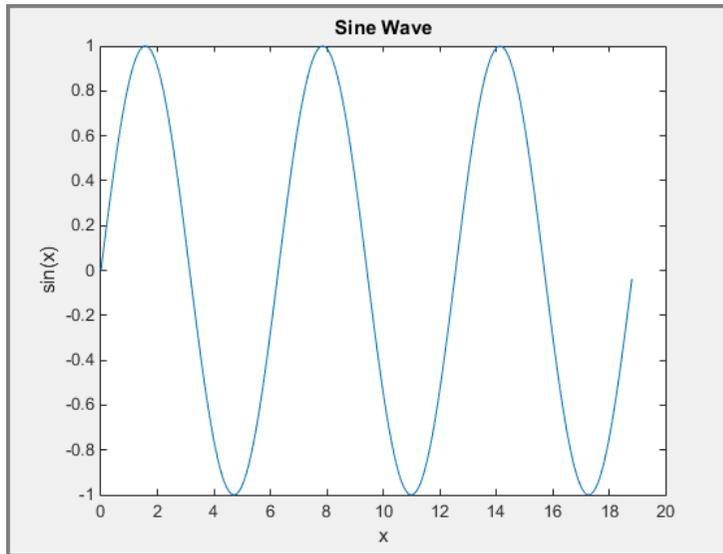


- 3 Type 2 in the \div / \times text box.
- 4 Click the \div button several times.

The sine plot becomes smoother after each subsequent click.



- 5 Close the Figure and save the file.
- 6 Run the entire `sine_wave.m` file. A smooth sine plot with titles appears in a new Figure.



Change the Appearance of Code Sections

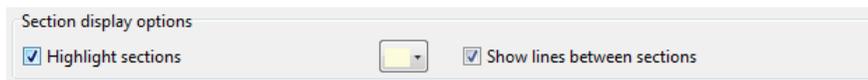
You can change how code sections appear within the MATLAB Editor. MATLAB highlights code sections in yellow, by default, and divides them with horizontal lines. When the cursor is positioned in any line within a section, the Editor highlights the entire section.

To change how code sections appear:

- 1 On the **Home** tab, in the **Environment** section, click  **Preferences**.

The Preference dialog box appears.

- 2 In the left pane, select **MATLAB > Colors > Programming Tools**.
- 3 Under **Section display options**, select the appearance of your code sections.



You can choose whether to highlight the sections, the color of the highlighting, and whether dividing lines appear between code sections.

Use Code Sections with Control Statements and Functions

Unexpected results can appear when using code sections within control statements and functions because MATLAB automatically inserts section breaks that do not appear in the Editor unless *you* insert section breaks explicitly. This is especially true when nested code is involved. Nested code occurs wherever you place a control statement or function within the scope of another control statement or function.

MATLAB automatically defines section boundaries in a code block, according to this criteria:

- MATLAB inserts a section break at the top and bottom of a file, creating a code section that encompasses the entire file. However, the Editor does not highlight the resulting section, which encloses the entire file, unless you add one or more explicit code sections to the file.
- If you define a section break within a control flow statement (such as an `if` or `while` statement), MATLAB automatically inserts section breaks at the lines containing the start and end of the statement.
- If you define a section break within a function, MATLAB inserts section breaks at the function declaration and at the function end statement. If you do not end the function with an `end` statement, MATLAB behaves as if the end of the function occurs immediately before the start of the next function.

If an automatic break occurs on the same line as a break you insert, they collapse into one section break.

Nested Code Section Breaks

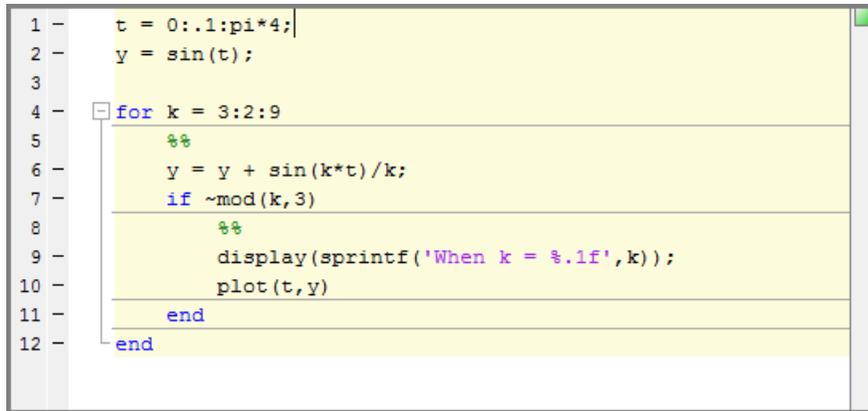
The following code illustrates the concept of nested code sections:

```
t = 0:.1:pi*4;
y = sin(t);

for k = 3:2:9
    %%
    y = y + sin(k*t)/k;
    if ~mod(k,3)
        %%
        display(sprintf('When k = %.1f',k));
        plot(t,y)
    end
end
```

If you copy and paste this code into a MATLAB Editor, you see that the two section breaks create three nested levels:

- **At the outermost level of nesting**, one section spans the entire file.



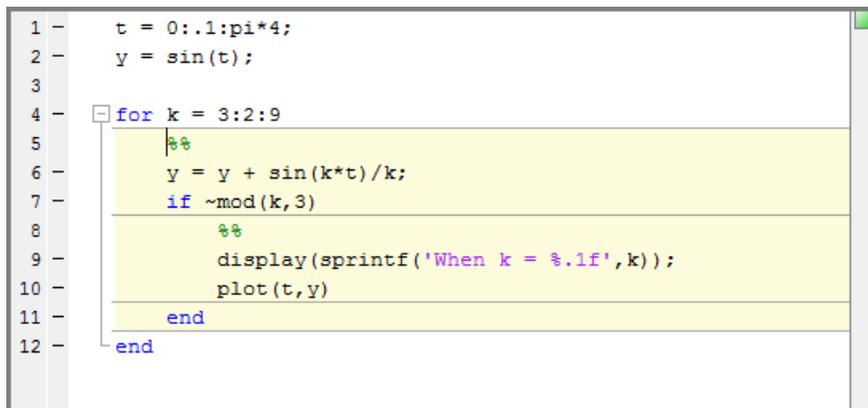
```

1 - t = 0:.1:pi*4;
2 - y = sin(t);
3
4 - □ for k = 3:2:9
5     %%
6     y = y + sin(k*t)/k;
7     if ~mod(k,3)
8         %%
9         display(sprintf('When k = %.1f',k));
10        plot(t,y)
11    end
12 - end

```

MATLAB only defines section in a code block if you specify section breaks *at the same level* within the code block. Therefore, MATLAB considers the cursor to be within the section that encompasses the entire file.

- **At the second level of nesting**, a section exists within the for loop.



```

1 - t = 0:.1:pi*4;
2 - y = sin(t);
3
4 - □ for k = 3:2:9
5     □ %%
6     y = y + sin(k*t)/k;
7     if ~mod(k,3)
8         □ %%
9         display(sprintf('When k = %.1f',k));
10        plot(t,y)
11    end
12 - end

```

- **At the third-level of nesting**, one section exists within the if statement.

```
1 - t = 0:.1:pi*4;  
2 - y = sin(t);  
3  
4 - for k = 3:2:9  
5     %%  
6     y = y + sin(k*t)/k;  
7     if ~mod(k,3)  
8         %%  
9         display(sprintf('When k = %.1f',k));  
10        plot(t,y)  
11    end  
12 end
```

See Also

More About

- “Create Scripts” on page 18-2
- “Create Live Scripts in the Live Editor” on page 19-7
- “Scripts vs. Functions” on page 18-16

Scripts vs. Functions

This topic discusses the differences between scripts and functions, and shows how to convert a script to a function.

Both scripts and functions allow you to reuse sequences of commands by storing them in program files. Scripts are the simplest type of program, since they store commands exactly as you would type them at the command line. However, functions are more flexible and more easily extensible.

Create a script in a file named `triarea.m` that computes the area of a triangle:

```
b = 5;  
h = 3;  
a = 0.5*(b.*h)
```

After you save the file, you can call the script from the command line:

```
triarea  
  
a =  
    7.5000
```

To calculate the area of another triangle using the same script, you could update the values of `b` and `h` in the script and rerun it. Each time you run it, the script stores the result in a variable named `a` that is in the base workspace.

However, instead of manually updating the script each time, you can make your program more flexible by converting it to a function. Replace the statements that assign values to `b` and `h` with a function declaration statement. The declaration includes the `function` keyword, the names of input and output arguments, and the name of the function.

```
function a = triarea(b,h)  
a = 0.5*(b.*h);  
end
```

After you save the file, you can call the function with different base and height values from the command line without modifying the script:

```
a1 = triarea(1,5)  
a2 = triarea(2,10)  
a3 = triarea(3,6)
```

```
a1 =  
    2.5000  
a2 =  
    10  
a3 =  
    9
```

Functions have their own workspace, separate from the base workspace. Therefore, none of the calls to the function `triaarea` overwrite the value of `a` in the base workspace. Instead, the function assigns the results to variables `a1`, `a2`, and `a3`.

See Also

More About

- “Create Scripts” on page 18-2
- “Create Functions in Files” on page 20-2
- “Add Functions to Scripts” on page 18-18
- “Base and Function Workspaces” on page 20-10

Add Functions to Scripts

MATLAB scripts, including live scripts, can contain code to define functions. These functions are called local functions. Local functions are useful if you want to reuse code within a script. By adding local functions, you can avoid creating and managing separate function files. They are also useful for experimenting with functions, which can be added, modified, and deleted easily as needed. Functions in scripts are supported in R2016b or later.

Add Local Functions

Local functions are only visible within the file where they are defined, both to the script code and other local functions within the file. They are not visible to functions in other files, and cannot be called from the command line. They are equivalent to subroutines in other programming languages, and are sometimes called subfunctions.

To add local functions to a script, first, create the script. Go to the **Home** tab and select **New > Script**. For more information about creating scripts, see “Create Scripts” on page 18-2. You also can “Create Live Scripts in the Live Editor” on page 19-7.

After you create the script, add code to the script and save it. For example, add this code and save it as a script called `mystats.m`. This code declares an array, determines the length of the array, and passes both values to the local functions `mymean` and `mymedian`. The local functions `mymean` and `mymedian` calculate the average and median of the input list and return the results.

Note Including functions in scripts requires MATLAB R2016b or later.

```
x = 1:10;
n = length(x);
avg = mymean(x,n);
med = mymedian(x,n);

function a = mymean(v,n)
% MYMEAN Example of a local function.

    a = sum(v)/n;
end
```

```
function m = mymedian(v,n)
% MYMEDIAN Another example of a local function.

    w = sort(v);
    if rem(n,2) == 1
        m = w((n + 1)/2);
    else
        m = (w(n/2) + w(n/2 + 1))/2;
    end
end
```

You can add local functions in any order, as long as they all appear after the rest of the script code. Each function begins with its own function definition statement, and ends with the `end` keyword. The definition statement is the first executable line of any function, for example, `function a = mymean(v,n)`. For more information about function definition statements, including how to create them, see “Create Functions in Files” on page 20-2.

Access Help

Although you cannot call a local function from the command line or from functions in other files, you can access its help using the `help` command. Specify the names of both the script and the local function, separating them with a `>` character:

```
help mystats>mymean

mymean Example of a local function.
```

Run Code

To run a script, including all local functions, click the  **Run** (for scripts) or  **Run All** (for live scripts) button, or type the saved script name in the Command Window. You can also run individual sections in a script by clicking the  **Run Section** button.

Local functions in the current file have precedence over functions in other files. That is, when you call a function within a program file, MATLAB checks whether the function is a local function before looking for other functions. This allows you to create an alternate version of a particular function while retaining the original in another file.

Scripts create and access variables in the base workspace. Local functions, similar to other functions, have their own workspaces that are separate from the base workspace.

Local functions cannot access variables in the workspace of other functions or in the base workspace, unless you pass them as arguments. For more information, see “Base and Function Workspaces” on page 20-10.

Add and Run Sections in Live Scripts

The Live Editor does not support running individual sections within local functions. For this reason, you cannot add section breaks within local functions in live scripts. When you add local functions to a live script, MATLAB automatically adds a section break before the first local function definition, and removes all section breaks after it. You can however run individual sections within the script code, even if they contain a call to one of the local functions in the file. For more information on section breaks and running live scripts, see “Run Sections in Live Scripts” on page 19-15.

See Also

More About

- “Create Functions in Files” on page 20-2
- “Function Precedence Order” on page 20-43

Live Scripts and Functions

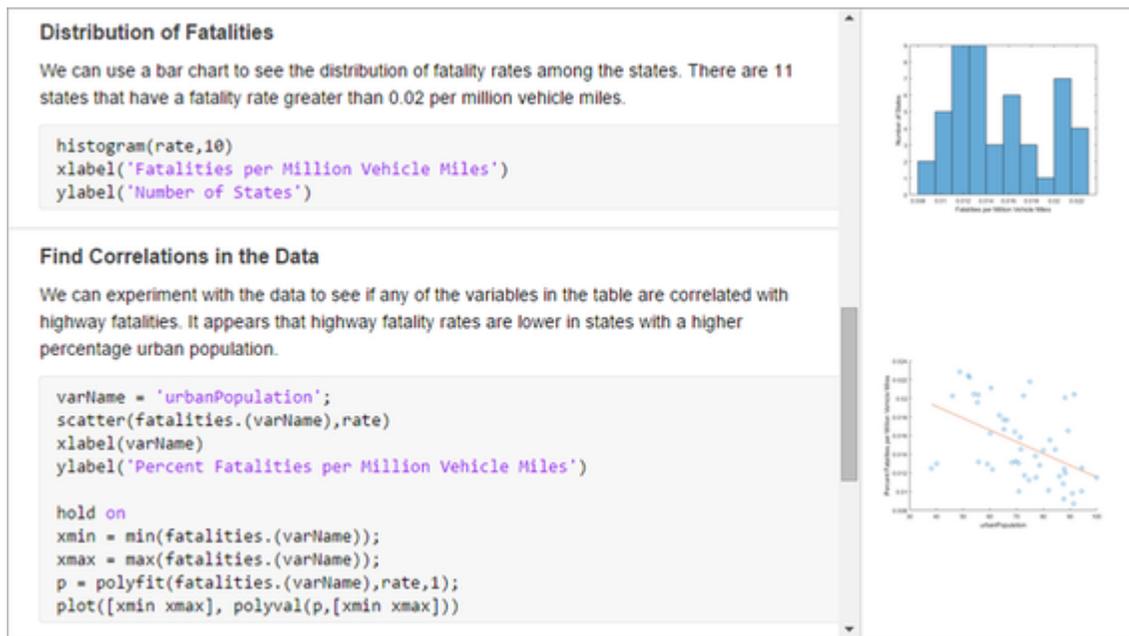
What Is a Live Script or Function?

MATLAB live scripts and live functions are interactive documents that combine MATLAB code with formatted text, equations, and images in a single environment called the Live Editor. In addition, live scripts store and display output alongside the code that creates it.

Use live scripts and functions to:

Visually explore and analyze problems

- Write, execute, and test code in a single interactive environment.
- Run blocks of code individually or as a whole file, and view the results and graphics with the code that produced them.

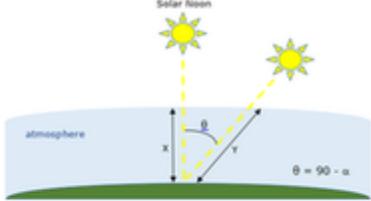


Share richly formatted, executable narratives

- Add titles, headings, and formatted text to describe a process and include equations, images, and hyperlinks as supporting material.

- Save your narratives as richly formatted, executable documents and share them with colleagues or the MATLAB community, or convert them to HTML, PDF, or LaTeX files for publication.

Air Mass and Solar Radiation



The larger the air mass, the less radiation reaches the ground. The air mass can be calculated from the equation

$$AM = \frac{1}{\cos(90 - \alpha) + 0.5057(6.0799 + \alpha)^{-1.6364}}$$

Then the solar radiation (in Kw/m²) reaching the ground can be calculated from the empirical equation

$$sRad = 1.353 * 0.7^{AM^{0.678}}$$

```
AM = 1/(cosd(90-alpha) + 0.50572*(6.07955+alpha)^-1.6354);
sRad = 1.353*0.7^(AM^0.678); % kW/m^2
disp(['Air Mass = ' num2str(AM) ' Solar Radiation = ' num2str(sRad) ' kW/m^2'])
```

Create interactive lectures for teaching

- Combine code and results with formatted text and mathematical equations.
- Create step-by-step lectures and evaluate them incrementally to illustrate a topic.
- Modify code on the fly to answer questions or explore related topics.
- Share lectures with students as interactive documents or in hard copy format, and distribute partially completed files as assignments.

Homework

Use the techniques described above to complete the following exercises:

Exercise 1: Write MATLAB code to calculate the 3 cube roots of i .

% Put your code here

Exercise 2: Write MATLAB code to calculate the 5 fifth roots of -1 .

% Put your code here

Exercise 3: Describe the mathematical approach you would use to calculate the n th roots of an arbitrary complex number. Include the equations you used in your approach.

(Describe your approach here)

Differences with Plain Code Scripts and Functions

Live scripts and live functions differ from plain code scripts and functions in several ways. This table summarizes the main differences.

	Live Scripts and Functions	Plain Code Scripts and Functions
File Format	Live Code file format. For more information, see “Live Code File Format (.mlx)” on page 19-69	Plain Text file format
File Extension	.mlx	.m
Output Display	With code in the Live Editor (live scripts only)	In Command Window
Internationalization	Interoperable across locales	Non-7-bit ASCII characters are not compatible across all locales
Text Formatting	Add and view formatted text in the Live Editor	Use publishing markup to add formatted text, publish to view

	Live Scripts and Functions	Plain Code Scripts and Functions
Visual Representation	<p>Viewing a Penny</p> <p>This example shows four techniques to visualize the surface data of a penny. The file PENNY.MAT contains measurements made at the National Institute of Standards and Technology of the depth of the mold used to mint a U. S. penny, sampled on a 128-by-128 grid.</p> <pre>1 % Copyright 1984-2014 The MathWorks, Inc.</pre> <p>Drawing a Contour Plot</p> <p>Draw a contour plot with 15 copper colored contour lines.</p> <pre>2 load penny.mat 3 contour(P,15) 4 colormap(copper) 5 axis ij square</pre> <p>Drawing a Pseudocolor Plot</p> <p>Draw a pseudocolor plot with brightness proportional to height.</p> <pre>6 pcolor(P) 7 axis ij square 8 shading flat</pre>	<pre>1 %% Viewing a Penny 2 % This example shows four techniques to visualize the su 3 % penny. The file PENNY.MAT contains measurements made a 4 % Institute of Standards and Technology of the depth of 5 % mint a U. S. penny, sampled on a 128-by-128 grid. 6 7 % Copyright 1984-2014 The MathWorks, Inc. 8 9 %% Drawing a Contour Plot 10 % Draw a contour plot with 15 copper colored contour lin 11 12 load penny.mat 13 contour(P,15) 14 colormap(copper) 15 axis ij square 16 17 18 %% Drawing a Pseudocolor Plot 19 % Draw a pseudocolor plot with brightness proportional t 20 21 pcolor(P) 22 axis ij square 23 shading flat 24 25</pre>

Requirements

- **MATLAB R2016a** — MATLAB supports live scripts in versions R2016a and above, and live functions in versions R2018a and above.
- **Operating System** — MATLAB supports the Live Editor in most of the operating systems supported by MATLAB. For more information, see System Requirements. There are several operating systems supported by MATLAB but the Live Editor does not support.

Unsupported versions include:

- Red Hat Enterprise Linux 6.
- Red Hat Enterprise Linux 7.
- SUSE Linux Enterprise Desktop versions 13.0 and earlier.
- Debian 7.6 and earlier.

Some operating systems require additional configuration to run the Live Editor. If you are unable to run the Live Editor on your system, Contact Technical Support for information on how to configure your system.

Unsupported Features

When deciding whether to create a live script or function, it is important to note several unsupported features in the Live Editor:

- **Classes** — Classes are not supported in the Live Editor. Create classes as plain code files (.m) instead. You then can use the classes in your live scripts or functions.
- **Editor preferences** — The Live Editor ignores most Editor preferences, including custom keyboard shortcuts and Emacs-style keyboard shortcuts.

Save Live Scripts and Functions as Plain Code

To save a live script or function as a plain code file (.m):

- 1** On the **Live Editor** tab, in the **File** section, select **Save > Save As...**
- 2** In the dialog box that appears, select **MATLAB Code files (*.m)** as the **Save as type**.
- 3** Click **Save**.

When saving, MATLAB converts all formatted content to publish markup.

See Also

Related Examples

- “Create Live Scripts in the Live Editor” on page 19-7
- “Live Code File Format (.mlx)” on page 19-69

Create Live Scripts in the Live Editor

Live scripts are program files that contain your code, output, and formatted text together in a single interactive environment called the Live Editor. In live scripts, you can write your code and view the generated output and graphics along with the code that produced it. Add formatted text, images, hyperlinks, and equations to create an interactive narrative that you can share with others.

Homework

Use the techniques described above to complete the following exercises:

Exercise 1: Write MATLAB code to calculate the 3 cube roots of i .

`% Put your code here`

Exercise 2: Write MATLAB code to calculate the 5 fifth roots of -1 .

`% Put your code here`

Exercise 3: Describe the mathematical approach you would use to calculate a complex number. Include the equations you used in your approach.
(Describe your approach here)

Distribution of Fatalities

You can include visualizations in your program. Live output, plots and figures appear together with the code that produced them.

We can use a bar chart to see the distribution of fatality rates among the states. There are 11 states that have a fatality rate greater than 0.02 per million vehicle miles.

```

histogram(rate,10)
xlabel('Fatalities per Million Vehicle Miles')
ylabel('Number of States')

```

Viewing a Penny

This example shows four techniques to visualize the surface data of a penny. The file PENNY.MAT contains measurements made at the National Institute of Standards and Technology of the depth of the most used to mint a U.S. penny, sampled on a 128-by-128 grid.

`% Copyright 1984-2014 The MathWorks, Inc.`

Drawing a Contour Plot

Draw a contour plot with 15 copper colored contour lines.

```

load penny.mat
contour(P,15)
colormap(copper)
axis('square')

```

Drawing a Pseudocolor Plot

Draw a pseudocolor plot with brightness proportional to height.

```

pcolor(P)
axis('square')
shading flat

```

Create Live Script

To create a live script in the Live Editor, go to the **Home** tab and click **New Live Script**



You also can use the `edit` function in the Command Window. For example, type `edit penny.mlx` to open or create the file `penny.mlx`. To ensure that a live script is created, specify a `.mlx` extension. If an extension is not specified, MATLAB defaults to a file with `.m` extension, which only supports plain code.

Open Existing Script as Live Script

If you have an existing script, you can open it as a live script in the Live Editor. Opening a script as a live script creates a copy of the file and leaves the original file untouched. MATLAB converts publishing markup from the original script to formatted content in the new live script.

To open an existing script (`.m`) as a live script (`.mlx`) from the Editor, right-click the document tab, and select **Open *scriptName* as Live Script** from the context menu.

Alternatively, go to the **Editor** tab, click **Save**, and select **Save As**. Then, set the **Save as type:** to MATLAB Live Code Files (*.mlx) and click **Save**.

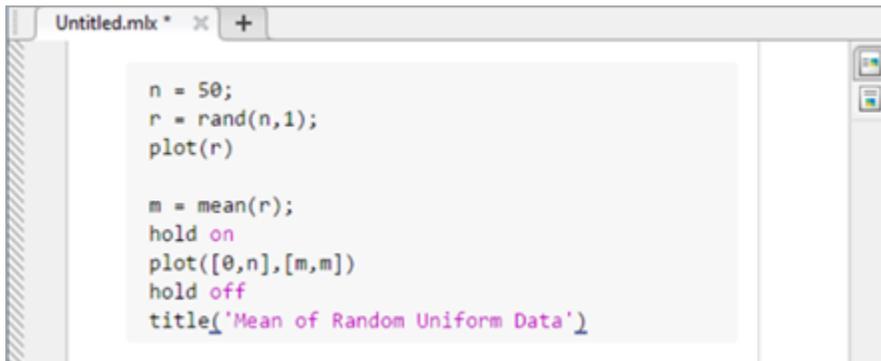
Note You must use one of the described conversion methods to convert your script to a live script. Simply renaming the script with a .mlx extension does not work and can corrupt the file.

Add Code

After you create a live script, you can add code and run it. For example, add this code that plots a vector of random data and draws a horizontal line on the plot at the mean.

```
n = 50;
r = rand(n,1);
plot(r)

m = mean(r);
hold on
plot([0,n],[m,m])
hold off
title('Mean of Random Uniform Data')
```



By default, MATLAB autocompletes block endings, parentheses, and quotes when entering code in the Live Editor. For example, type `if` and then press **Enter**. MATLAB automatically adds the end statement.

```
if
end
```

MATLAB also autocompletes comments, character vectors, and strings when split across two lines. To escape out of an autocompletion, press **Ctrl+Z** or the **Undo**  button. Autocompletions are enabled by default. To disable them, see “Editor/Debugger Autocoding Preferences”.

When adding or editing code, you can select and edit a rectangular area of code (also known as column selection or block edit). This is useful if you want to copy or delete several columns of data (as opposed to rows), or if you want to edit multiple lines at one time. To select a rectangular area, press the **Alt** key while making a selection.

For example, select the second column of data in *A*.

```
A = [ 10 20 30 40 50; ...
      60 70 80 90 100; ...
      110 120 130 140 150];
```

Type **0** to set all the selected values to 0.

```
A = [ 10 0 30 40 50; ...
      60 0 80 90 100; ...
      110 0 130 140 150];
```

Run Code

To run the code, click the vertical striped bar to the left of the code. Alternatively, go to the **Live Editor** tab and click **Run**. While your program is running, a status indicator  appears at the top left of the Editor window. A gray blinking bar to the left of a line of code indicates the line that MATLAB is evaluating. To navigate to the line MATLAB is evaluating, click the status indicator.

If an error occurs while MATLAB is running your program, the status indicator becomes an error icon . To navigate to the error, click the icon. An error icon  to the right of the line of code indicates the error. The corresponding error message is displayed as an output.

You do not need to save your live script to run it. When you do save your live script, MATLAB automatically saves it with a `.mlx` extension. For example, go the **Live Editor** tab, click  **Save**, and enter the name `plotRand`. MATLAB saves the live script as `plotRand.mlx`.

Display Output

By default, MATLAB displays output to the right of the code. Each output is displayed with the line that creates it, like in the Command Window.



You can change the size of the output display panel by dragging left or right on the resizer bar between the code and output.

To clear all output, right-click anywhere in the script and select **Clear All Output**. Alternatively, go to the **View** tab and in the **Output** section, click the **Clear all Output** button.

When scrolling, MATLAB aligns the output to the code that generates it. To disable the alignment of output to code, right-click the output section and select **Disable Synchronous Scrolling**.

To move the output in line with the code, click the  icon to the right of the live script. You also can go to the **View** tab and in the **Layout** section, click the **Output Inline** button.



To modify figures in the output, use the tools in the upper-right corner of the figure axes or in the **Figure** toolstrip. You can use the tools to explore the data in a figure and add formatting and annotations. For more information, see “Modify Figures in Live Scripts” on page 19-26.

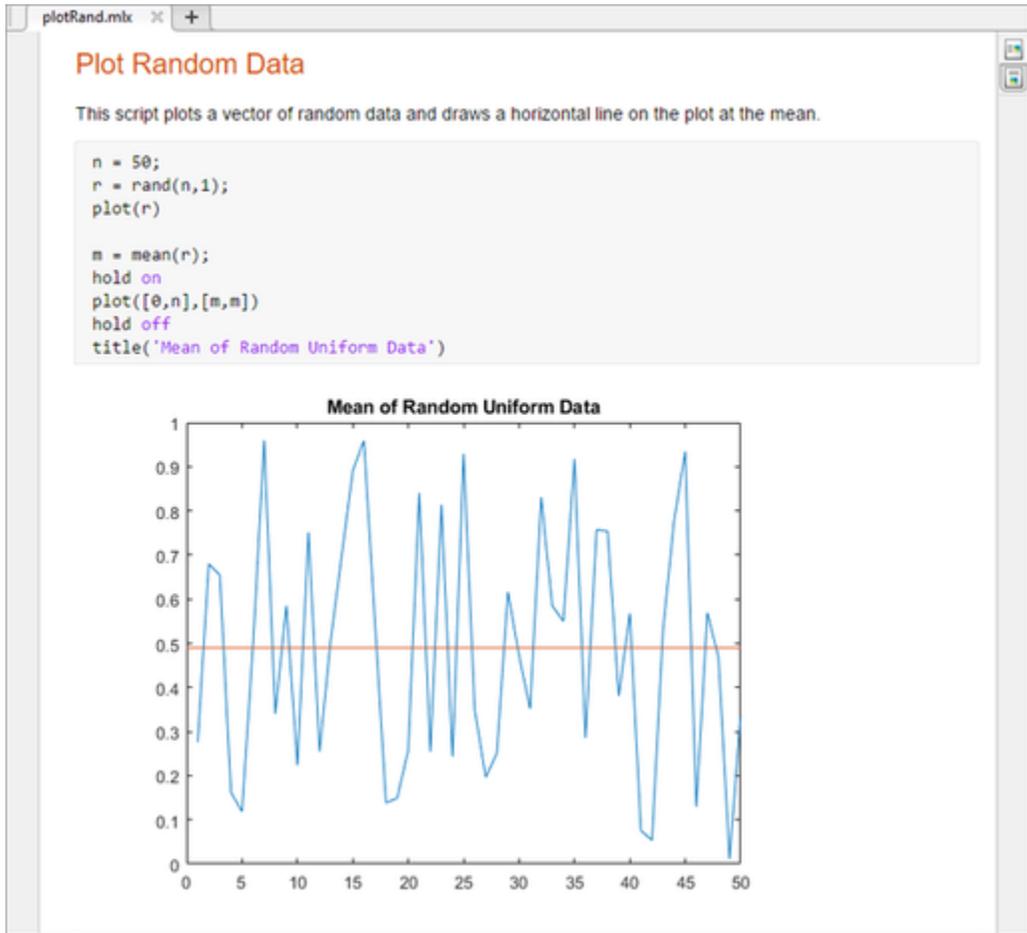
To open individual outputs, such as variables and figures, in a separate window, click the  icon in the upper right corner of the output. Variables open in the Variables editor, and figures open in a new figure window. Changes made to variables or figures outside of a live script do not apply to the output displayed in the live script.

Format Text

You can add formatted text, hyperlinks, images, and equations to your live scripts to create a presentable document to share with others. For example, add a title and some introductory text to `plotRand.mlx`:

- 1 Place your cursor at the top of the live script and, in the **Live Editor** tab, select  **Text**. A new text line appears above the code.
- 2 Click  and select **Title**.
- 3 Add the text `Plot Random Data`.
- 4 With your cursor still in the line, click the  button to center the text.
- 5 Press **Enter** to move to the next line.
- 6 Type the text `This script plots a vector of random data and draws a horizontal line on the plot at the mean.`

For more information including a list of all available formatting options, see “Format Files in the Live Editor” on page 19-38.



To adjust the displayed font size in the Live Editor, use the **Ctrl + Mouse Scroll** keyboard shortcut. The change in the displayed font size is not honored when exporting the live script to PDF, HTML, or LaTeX.

See Also

More About

- “Format Files in the Live Editor” on page 19-38
- “Run Sections in Live Scripts” on page 19-15
- “Modify Figures in Live Scripts” on page 19-26

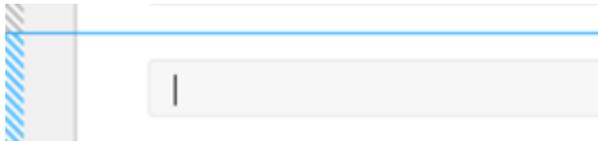
Run Sections in Live Scripts

Divide Your File Into Sections

Live scripts often contain many commands and lines of text. You typically focus efforts on a single part of your program at a time, working with the code and related text in pieces. For easier document management and navigation, divide your file into sections. Code, output, and related text can all appear together, in a single section.

To insert a section break into your live script, go to the **Live Editor** tab and in the **Section** section, click the **Section Break** button. The new section is highlighted in blue, indicating that it is selected. A vertical striped bar to the left of the section indicates that the section is stale. A stale section is a section that has not yet been run, or that has been modified since it was last run.

This image shows a new blank section in a live script.

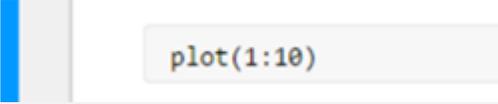


To delete a section break, click the beginning of the line directly after the section break and press **Backspace**. You also can click the end of the line directly before the section break and press **Delete**.

Evaluate Sections

Run your live script either by evaluating each section individually or by running all the code at once. To evaluate a section individually, it must contain all the values it requires, or the values must exist in the MATLAB workspace. Section evaluation runs the currently selected section, highlighted in blue. If there is only one section in your program file, the section is not highlighted, as it is always selected.

This table describes different ways to run your code.

Operation	Instructions
<p>Run the code in the selected section.</p>	<p>Click the blue bar to the left of the section.</p>  <p>OR</p> <p>On the Live Editor tab, in the Section section, click  Run Section.</p>
<p>Run the code in the selected section, and then move to the next section.</p>	<p>On the Live Editor tab, in the Section section, select  Run and Advance.</p>
<p>Run the code in the selected section, and then run all the code after the selected section.</p>	<p>On the Live Editor tab, in the Section section, select  Run to End.</p>
<p>Run all the code in the file.</p>	<p>On the Live Editor tab, in the Run section, click  Run.</p>

See Also

Related Examples

- “Modify Figures in Live Scripts” on page 19-26
- “Debug Code in the Live Editor” on page 19-17

Debug Code in the Live Editor

To diagnose problems in your live scripts or functions, debug your code in the Live Editor. There are several ways to debug in the Live Editor:

- Show output by removing semicolons.
- Run to a specific line of code and pause using the  button.
- Step into functions and scripts while paused by using the  button.
- Add breakpoints to your file to enable pausing at specific lines when you run.

Show Output

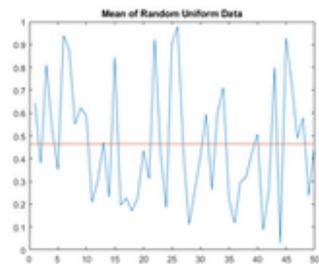
A simple way to determine where a problem occurs in your live script or function is to show output. To show the output for a line, remove the semi-colon from the end of that line. The Live Editor displays each output with the line of code that creates it, making it easy to determine where a problem occurs.

For example, suppose that you have a script called `randomdata.mlx` that plots a vector of random data and draws a horizontal line on the plot at the mean.

Plot Random Data

This script plots a vector of random data and draws a horizontal line on the plot at the mean.

```
1 n = 50;  
2 r = rand(n,1);  
3 plot(r)  
4  
5 m = mean(r);  
6 hold on  
7 plot([0,n],[m,m])  
8 hold off  
9 title('Mean of Random Uniform Data')
```



To display the output of the `rand` function at line 2, remove the semi-colon at the end of the line. To display line numbers in the Live Editor if they are not visible, go to the **View** tab and press the  **Line Numbers** button. MATLAB displays the value of `r`.

Plot Random Data

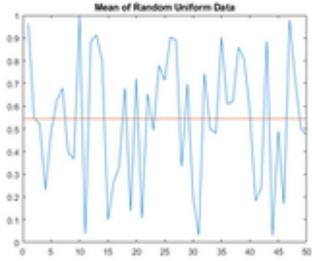
This script plots a vector of random data and draws a horizontal line on the plot at the mean.

```

1  n = 50;
2  r = rand(n,1)
3  plot(r)
4
5  m = mean(r);
6  hold on
7  plot([0,n],[m,m])
8  hold off
9  title('Mean of Random Uniform Data')
```

```

r = 50x1
0.9631
0.5468
0.5211
0.2316
0.4889
0.6241
0.6791
0.3955
0.3674
0.9880
:
:
```



Debug Using Run to Here

Showing output is useful if you want to display the state of a single variable. To explore the state of all variables in the workspace, run your live script or function and then pause before running the specified line of code.

To run to a specified line of code and then pause, click the  button to the left of the line. If the selected line cannot be reached, MATLAB continues running until the end of the file is reached or a breakpoint is encountered.

For example, click the  button to the left of line 2 in `randomdata.mlx`. MATLAB runs `randomdata.mlx` starting at line 1 and pauses before running line 2.

Plot Random Data

This script plots a vector of random data and draws a horizontal line on the plot at the mean.

```
1 n = 50;
2 r = rand(n,1);
3 plot(r)
4
5 m = mean(r);
6 hold on
7 plot([0,n],[m,m])
8 hold off
9 title('Mean of Random Uniform Data')
```

When MATLAB pauses, the  **Run** button in the **Live Editor** tab changes to a  **Continue** button. The Live Editor indicates the line at which MATLAB is paused by highlighting the line in green. The highlighted line does not run until after MATLAB resumes running.

Plot Random Data

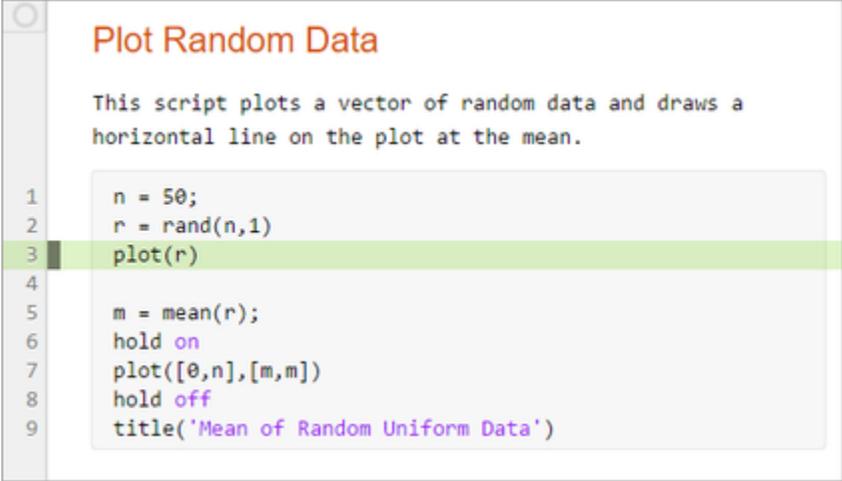
This script plots a vector of random data and draws a horizontal line on the plot at the mean.

```
1 n = 50;
2 r = rand(n,2);
3 plot(r)
4
5 m = mean(r);
6 hold on
7 plot([0,n],[m,m])
8 hold off
9 title('Mean of Random Uniform Data')
```

Tip It is a good practice to avoid modifying a file while MATLAB is paused. Changes that are made while MATLAB is paused do not run until after MATLAB finished running the code and the code is rerun.

To continue running the code, click the  **Continue** button. MATLAB continues running the file until it reaches the end of the file or a breakpoint. You also can click the  button to the left of the line of code you want to continue running to.

To continue running the code line-by-line, on the **Live Editor** tab, click  **Step**. MATLAB executes the current line at which it is paused and the pauses at the next line.



```
Plot Random Data

This script plots a vector of random data and draws a
horizontal line on the plot at the mean.

1  n = 50;
2  r = rand(n,1)
3  plot(r)
4
5  m = mean(r);
6  hold on
7  plot([0,n],[m,m])
8  hold off
9  title('Mean of Random Uniform Data')
```

View Variable Value While Debugging

To view the value of a variable while MATLAB is paused, hover the mouse pointer over the variable. The current value of the variable appears in a data tip. The data tip stays in view until you move the pointer. To disable data tips, go to the **View** tab and press the  **Datatips** button.

Plot Random Data

This script plots a vector of random data and draws a horizontal line on the plot at the mean.

```

1 n = 50;
2 r = rand(n,1);
3 plot(r)
4
5 m = mean(r);
6 hold on
7 plot([0,n],[m,m])
8 hold off
9 title('Mean of Random Uniform Data')

```

You also can view the value of a variable by typing the variable name in the Command Window. For example, to see the value of the variable `n`, type `n` and press **Enter**. The Command Window displays the variable name and its value. To view all the variables in the current workspace, use the Workspace browser.

Pause a Running File

To pause a program while it is running, go to the **Live Editor** tab and click the  **Pause** button. MATLAB pauses at the next executable line, and the  **Pause** button changes to a  **Continue** button. To continue running, press the  **Continue** button.

Pausing is useful if you want to check on the progress of a long running program to ensure that it is running as expected.

Note Clicking the pause button can cause MATLAB to pause in a file outside your own program file. Pressing the  **Continue** button resumes running without changing the results of the file.

End Debugging Session

After you identify a problem, end the debugging session by going to the **Live Editor** tab and clicking  **Stop**. To avoid confusion, make sure to end your debugging session

every time you are done debugging. The Live Editor automatically ends the debugging session when you save.

Step Into Functions

While debugging, you can step into called files, pausing at points where you want to examine values. To step into a file, click the  button directly to the left of the function you want to step into. The Live Editor only displays the button if the line contains a call to another function.

By default, the Live Editor only steps into user-defined functions and scripts. To enable stepping into MathWorks functions as well, on the **Home** tab, in the **Environment** section, click **Preferences**. Then, select **MATLAB > Editor/Debugger**, and in the **Debugging in the Live Editor** section, clear **Only show Step in button for user-defined functions**.

After stepping in, clicking the  button at the top of the file runs the rest of the called function, leaves the called function, and then pauses.

Examine Variables in the Workspace

When you step into a called function or file, the Live Editor displays the list of the functions MATLAB executed before pausing at the current line. The list is shown at the top of the file and displays the functions in order, starting on the left with the first called script or function, and ending on the right with the current script or function in which MATLAB is paused. This list is called the function call stack.



For each function in the function call stack, there is a corresponding workspace. Workspaces contain variables that you create within MATLAB or import from data files or other programs. Variables that you assign through the Command Window or create using scripts belong to the base workspace. Variables that you create in a function belong to their own function workspace.

To examine a variable during debugging, you must first select its workspace. The selected function in the function call stack indicates the current workspace. To select or change the workspace, click the function in the function call stack. MATLAB opens the function in the Live Editor and changes the current workspace to the workspace of the function.

Once the workspace is selected, you can view the values of the variables in it using the Workspace browser or as a data tip in the Live Editor.

Add Breakpoints and Run

If there are lines of code in your file that you want to pause at every time you run, add breakpoints at those lines. To add a breakpoint in the Live Editor, click the gray area to the left on an executable line where you want to set the breakpoint. For example, click the area next to line 3 in this code to add a breakpoint at that line.



The screenshot shows a MATLAB Live Editor window titled "Plot Random Data". The window contains a description: "This script plots a vector of random data and draws a horizontal line on the plot at the mean." Below the description is a code editor with the following code:

```
1 n = 50;
2 r = rand(n,1);
3 plot(r)
4
5 m = mean(r);
6 hold on
7 plot([0,n],[m,m])
8 hold off
9 title('Mean of Random Uniform Data')
```

A green arrow points to line 3, indicating a breakpoint has been set at that line.

When you run the file, MATLAB pauses at the line of code indicated by the breakpoint.

Clear Breakpoints

When you close and reopen a file, breakpoints are saved.

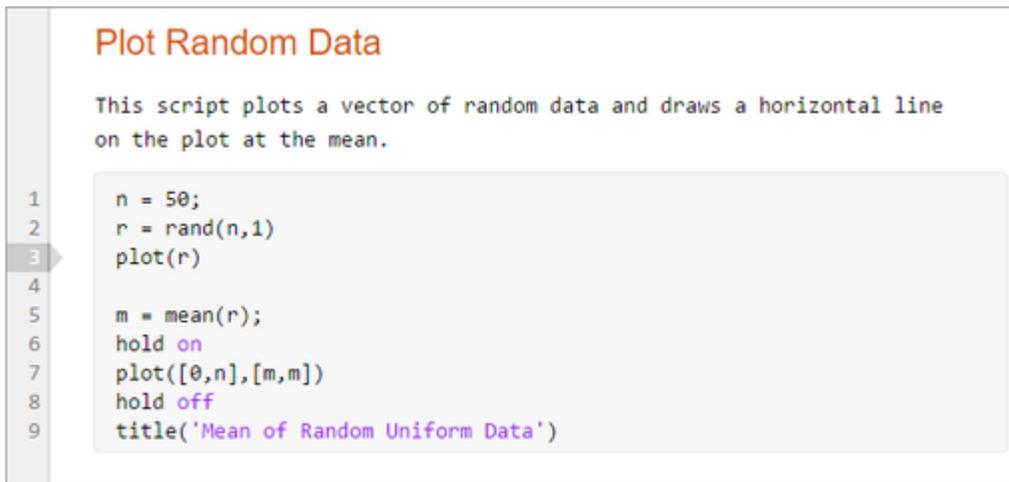
To clear a breakpoint, right-click the breakpoint and select **Clear Breakpoint** from the context menu. To clear all breakpoints in the file, select **Clear All in File**. To clear all breakpoints in *all* files, select **Clear All**.

Breakpoints clear automatically when you end a MATLAB session.

Disable Breakpoints

You can disable selected breakpoints so that your program temporarily ignores them and runs uninterrupted. For example, you might disable a breakpoint after you identify and correct a problem.

To disable a breakpoint, right-click it and select **Disable Breakpoint** from the context menu. The breakpoint becomes gray to indicate that it is disabled.



```
Plot Random Data

This script plots a vector of random data and draws a horizontal line
on the plot at the mean.

1  n = 50;
2  r = rand(n,1)
3  plot(r)
4
5  m = mean(r);
6  hold on
7  plot([0,n],[m,m])
8  hold off
9  title('Mean of Random Uniform Data')
```

To reenable a breakpoint, right-click it and select **Enable Breakpoint** from the context menu. To enable or disable all breakpoints in the file, select **Enable All Breakpoints in File** or **Disable All Breakpoints in File**. These options are only available if there is at least one breakpoint to enable or disable.

Add Conditional Breakpoints

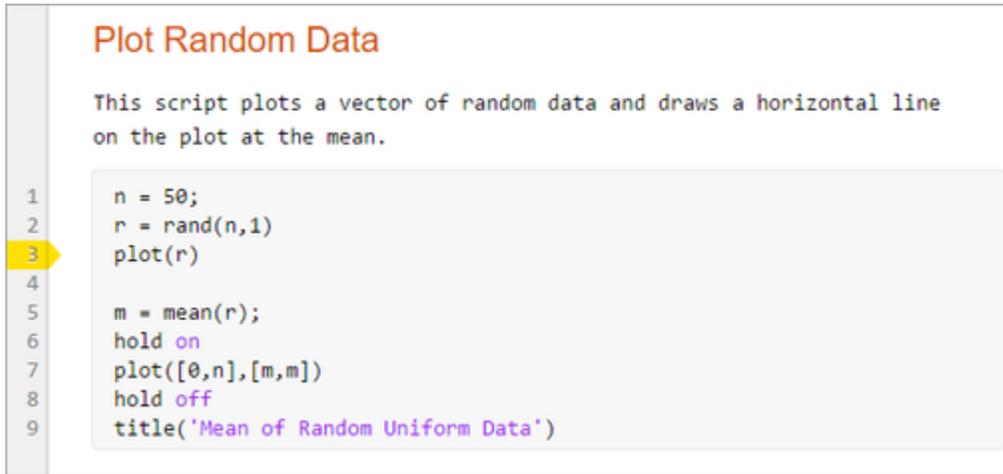
You can add a condition to a breakpoint that tells MATLAB when to pause at the specified line. To add a condition, right-click the breakpoint and select **Set/Modify Condition**. When the Editor dialog box opens, enter a condition and click **OK**. A condition is any valid MATLAB expression that returns a logical scalar value. MATLAB evaluates the condition before running the line.

For example, suppose that you only want to pause in `randomdata.mlx` if the random generated data contains a 0.

Add a breakpoint with the following condition at line 3:

```
any(r == 0)
```

A yellow conditional breakpoint appears at that line.



Plot Random Data

This script plots a vector of random data and draws a horizontal line on the plot at the mean.

```
1 n = 50;
2 r = rand(n,1)
3 plot(r)
4
5 m = mean(r);
6 hold on
7 plot([0,n],[m,m])
8 hold off
9 title('Mean of Random Uniform Data')
```

When you run the file, MATLAB pauses at the specified line when the condition is met. For example, in the `randomdata` example, MATLAB pauses before running line 3 if any of the values in `r` are equal to 0.

See Also

More About

- “Run Sections in Live Scripts” on page 19-15

Modify Figures in Live Scripts

You can modify figures interactively in the Live Editor. Use the provided tools to explore data and add formatting, annotations, or additional axes to your figures. Then, update your code to reflect changes using the generated code.

Explore Data

You can pan, zoom, and rotate a figure in your script using the tools that appear in the upper-right corner of the figure axes when you hover over the figure.

-  — Add data tips to display data values.
-  — Rotate the plot (3-D plots only).
-  — Pan the plot.
-  — Zoom in and out of the plot.
-  — Undo all pan, zoom, and rotate actions and restore the original view of the plot.

To undo or redo an action, click  or  at the upper right corner of the toolstrip.

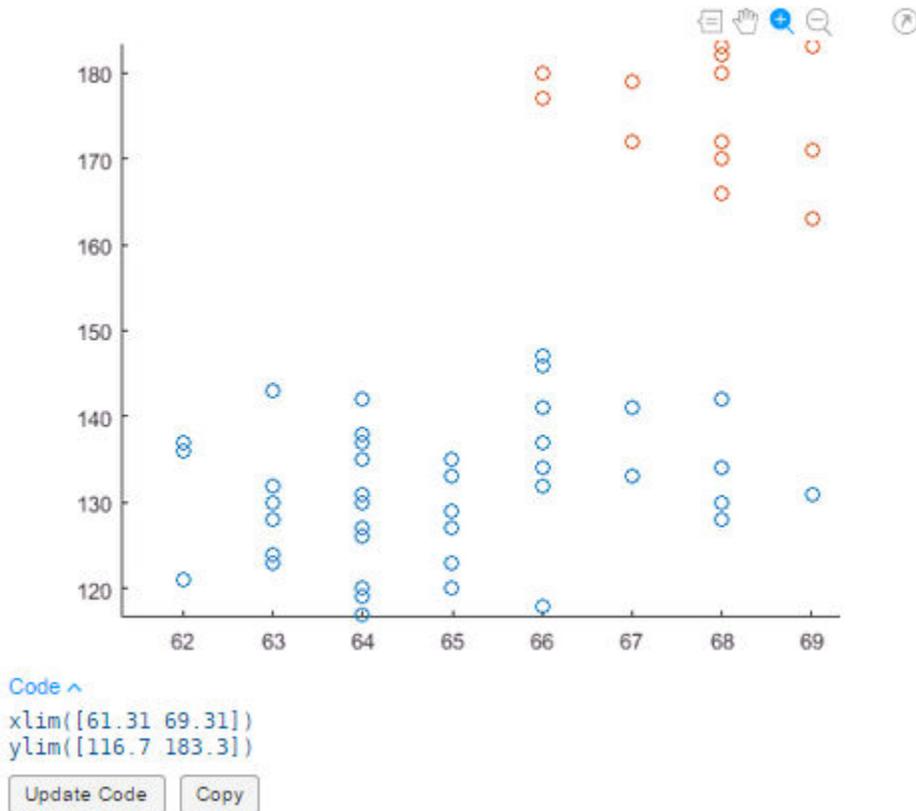
Note

- When you open a saved live script,  appears next to each output figure, indicating that the interactive tools are not available yet. To make these tools available, run the live script.
- The interactive tools are not available for invisible axes.

Suppose that you want to explore the health information for 100 different patients. Create a live script called `patients.mlx` and add code that loads the data and adds a scatter plot that shows the height versus weight of two groups of patients, female and male. Run the code by going to the **Live Editor** tab and clicking  **Run**.

```
load patients
```

```
figure
```

Update Code with Figure Changes

When modifying output figures in live scripts, changes to the figure are not automatically added to the script. With each interaction, MATLAB generates the code needed to reproduce the interactions and displays this code either underneath or to the right of the figure. Use the **Update Code** button to add the generated code to your script. This ensures that the interactions are reproduced the next time you run the live script.

For example, in the live script `patients.mlx`, after zooming in on patients with a height of 64, click the **Update Code** button. MATLAB adds the generated code after the line containing the code for creating the plot.

```
xlim([61.31 69.31])
ylim([116.7 183.3])
```

If MATLAB is unable to determine where to place the generated code, the **Update Code** button is disabled. This occurs, for example, if you modify the code without running the script again. In this case, use the **Copy** button to copy the generated code into the clipboard. You then can paste the code into your script at the appropriate location.

Add Formatting and Annotations

In addition to exploring the data, you can format and annotate your figures interactively by adding titles, labels, legends, grid lines, arrows, and lines. To add an item, first select the desired figure. Then, go to the **Figure** tab and, in the **Annotations** section, select one of the available options. Use the down arrow on the right side of the section to display all available annotations. To add a formatting or annotation option to your favorites, click the star at the top right of the desired annotation icon. To undo or redo a formatting or annotation action, click  or  at the upper right corner of the toolbar.

Annotation options include:

-  **Title** — Add a title to the axes. To modify an existing title, click the existing title and enter the modified text.
-  **X-Label**,  **Y-Label** — Add a label to the axes. To modify an existing label, click the existing label and enter the modified text.
-  **Legend** — Add a legend to the figure. To modify the existing legend descriptions, click the existing descriptions and enter the modified text. Select **Remove Legend** from the **Annotations** section to remove the legend from the axes.
-  **Colorbar** — Add a color bar legend to the figure. Select **Remove Colorbar** from the **Annotations** section to remove the color bar legend from the axes.
-  **Grid**,  **X-Grid**,  **Y-Grid** — Add grid lines to the figure. Select **Remove Grid** from the **Annotations** section to remove all the grid lines from the axes.

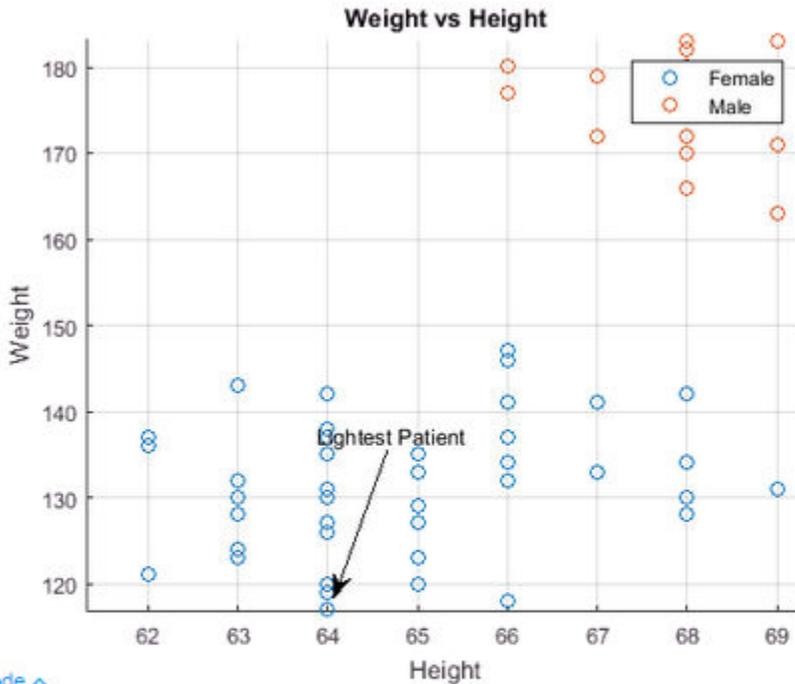
-  **Line**,  **Arrow**,  **Text Arrow**,  **Double Arrow** — Add a line or arrow annotation to the figure. Draw the arrow from tail to head. To move an existing annotation, click the annotation to select it and drag it to the desired location. Press the **Delete** key to delete the selected annotation.

Note Adding formatting and annotations using the **Figure** tab is not supported for invisible axes.

For example, suppose that you want to add formatting and annotations to the figure in `patients.mlx`.

- 1 Add a title** — In the **Annotations** section, select  **Title**. A blue rectangle appears prompting you to enter text. Type the text `Weight vs. Height` and press **Enter**.
- 2 Add X and Y Labels** — In the **Annotations** section, select  **X-Label**. A blue rectangle appears prompting you to enter text. Type the text `Height` and press **Enter**. Select  **Y-Label**. A blue rectangle appears prompting you to enter text. Type the text `Weight` and press **Enter**.
- 3 Add a legend** — In the **Annotations** section, select  **Legend**. A legend appears at the top right corner of the axes. Click the `data1` description in the legend and replace the text with `Female`. Click the `data2` description in the legend and replace the text with `Male`. Press **Enter**.
- 4 Add grid lines** — In the **Annotations** section, select  **Grid**. Grid lines appear in the axes.
- 5 Add an arrow annotation** — In the **Annotations** section, select  **Text Arrow**. Drawing the arrow from tail to head, position the arrow on the scatter plot pointing to the lightest patient. Enter the text `Lightest Patient` and press **Enter**.
- 6 Update the code** — In the selected figure, click the **Update Code** button. The live script now contains the code needed to reproduce the figure changes.

```
grid on
legend({'Female','Male'})
title('Weight vs Height')
xlabel('Height')
ylabel('Weight')
annotation('textarrow',[0.455 0.3979],[0.3393 0.13],'String','Lightest Patient');
```



Code ^

```
grid on
legend({'Female','Male'})
title('Weight vs Height')
xlabel('Height')
ylabel('Weight')
annotation('textarrow',[0.455 0.3979],[0.3393 0.13],'String','Lightest Pat:
```

Update Code

Copy

Add and Modify Multiple Subplots

You can combine multiple plots by creating subplots in a figure. To add multiple subplots to your figure, use the **Subplot** button to divide the figure into a grid of subplots. First, select the desired figure. Then, go to the **Figure** tab and choose a subplot layout using

the **Subplot** button. You only can add additional subplots to a figure if the figure contains one subplot. If a figure contains multiple subplots, the **Subplot** button is disabled.

For example, suppose that you want to compare the blood pressure of smoking and non-smoking patients. Create a live script called `patients_smoking.mlx` and add code that loads the health information for 100 different patients.

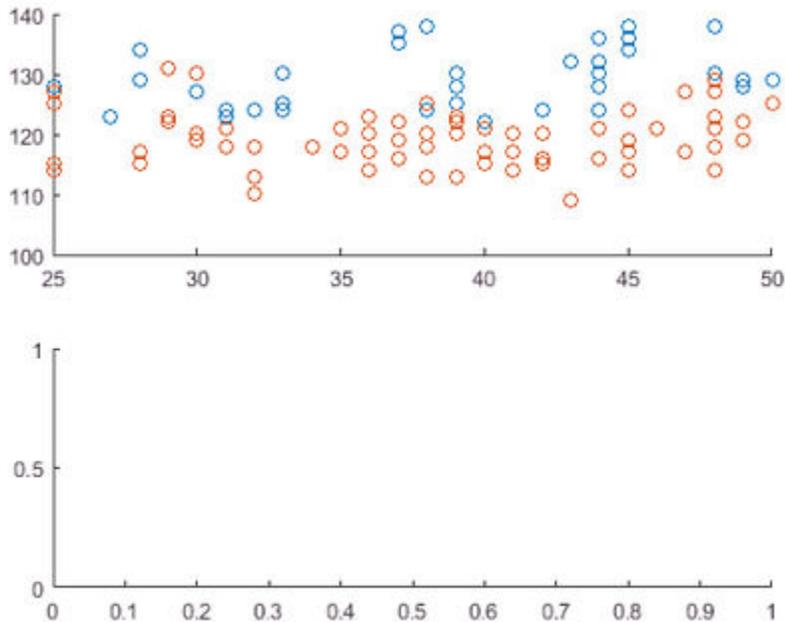
```
load patients
```

Run the code by going to the **Live Editor** tab and clicking  **Run**.

Add a scatter plot that shows the systolic blood pressure of patients that smoke versus the systolic blood pressure of patients that do not smoke. Run the code.

```
figure
scatter(Age(Smoker==1),Systolic(Smoker==1));
hold on
scatter(Age(Smoker==0),Systolic(Smoker==0));
hold off
```

In the **Figure** tab, select **Subplot** and choose the layout for two horizontal graphs.



Code ^

```
subplot(2,1,1,gca)
subplot(2,1,2)
```

Update Code

Copy

In the newly created figure, click the **Update Code** button. The live script now contains the code needed to reproduce the two subplots.

```
subplot(2,1,1,gca)
subplot(2,1,2)
```

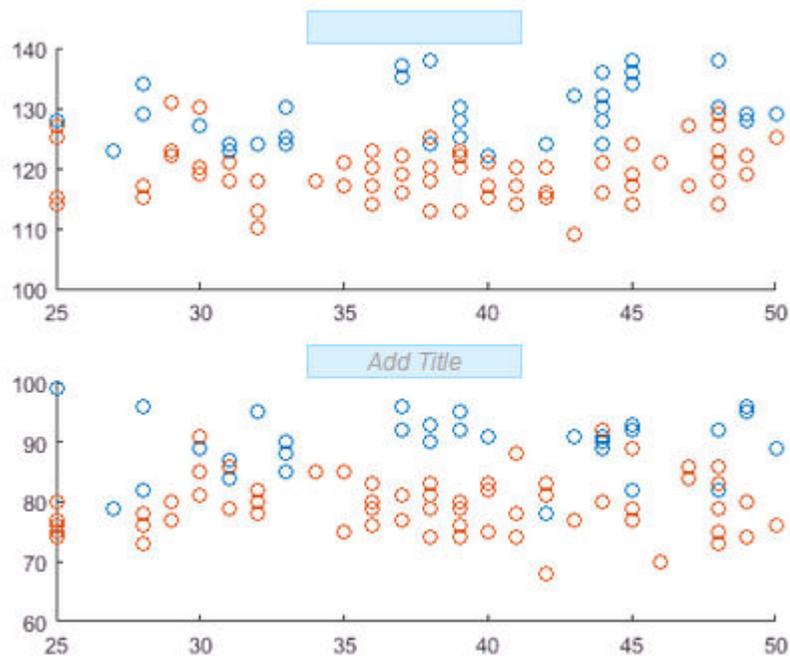
Add a scatter plot that shows the diastolic blood pressure of patients that smoke versus the diastolic blood pressure of patients that do not smoke. Run the code.

```
scatter(Age(Smoker==1),Diastolic(Smoker==1));
hold on
scatter(Age(Smoker==0),Diastolic(Smoker==0));
hold off
```

Add formatting:

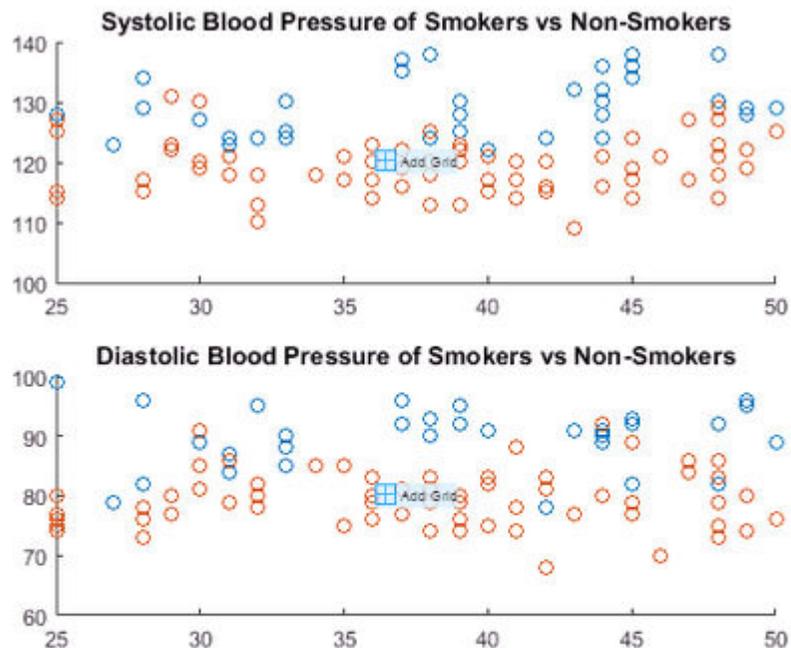
1

Add titles to each subplot — In the **Annotations** section, select  **Title**. A blue rectangle appears in each subplot prompting you to enter text. Type the text **Systolic Blood Pressure of Smokers vs Non-Smokers** in the first subplot and **Diastolic Blood Pressure of Smokers vs Non-Smokers** in the second subplot and press **Enter**.



2

Add grid lines to each subplot — In the **Annotations** section, select  **Grid**. An **Add Grid** icon appears on each subplot. Click the **Add Grid** icon on each subplot. Grid lines appear in both subplots.



Code ^

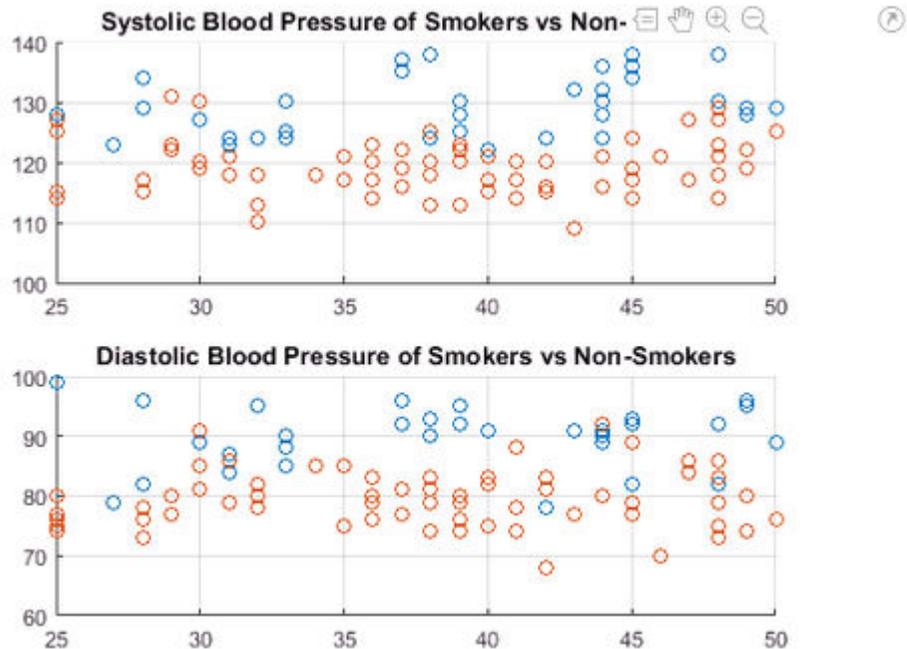
```
subplot(2,1,1)
title('Systolic Blood Pressure of Smokers vs Non-Smokers')
subplot(2,1,2)
title('Diastolic Blood Pressure of Smokers vs Non-Smokers')
```

Update Code

Copy

- 3 Update the code** — In the selected figure, click the **Update Code** button. The live script now contains the code needed to reproduce the figure changes.

```
subplot(2,1,1)
grid on
title('Systolic Blood Pressure of Smokers vs Non-Smokers')
subplot(2,1,2)
grid on
title('Diastolic Blood Pressure of Smokers vs Non-Smokers')
```



Save and Print Figure

At any point during figure modification, you can choose to save or print the figure for future use.

- 1 Click the  icon in the upper-right corner of the output. This opens the figure in a separate figure window.
- 2
 - a **To save the figure** — Select **File > Save As**. For more information on saving figures, see “Save Figure to Open in Another Application” or “Save Figure to Reopen in MATLAB Later”.
 - b **To print the figure** — Select **File > Print**. For more information on printing figures, see “Print Figure from File Menu”.

Note Any changes made to the figure in the separate figure window are not reflected in the live script. Similarly, any changes made to the figure in the live script are not reflected in the open figure window.

See Also

Related Examples

- “Format Files in the Live Editor” on page 19-38
- “Run Sections in Live Scripts” on page 19-15

Format Files in the Live Editor

You can add formatted text, hyperlinks, images, and equations to your live scripts and functions to create a presentable document to share with others.

To insert a new item, go to the **Insert** tab and select from the available options:

Option	Description	Additional Details
 Code	Insert a blank line of code.	You can insert a code line before, after, or between text lines.
 Section Break	Insert a section break.	You can insert a section break to divide your live script or function into manageable sections to evaluate individually. A section can consist of code, text, and output. For more information, see “Run Sections in Live Scripts” on page 19-15.
 Text	Insert a blank line of text.	A text line can contain formatted text, hyperlinks, images, or equations. You can insert a text line before, after, or between code lines.
 Table of Contents	Insert a table of contents.	The table of contents contains a list of all the titles and headings in the document. Only the title of the table of contents is editable. You can add a table of contents only in text lines. If you insert a table of contents into a code line, MATLAB places it directly above the current code section.
 Code Example	Insert a formatted code example.	A code example is sample code that appears as indented and monospaced text. Select Plain to insert sample code as unhighlighted text. Select MATLAB to insert sample code as text highlighted according to MATLAB syntax.
 Image	Insert an image.	You can add images only in text lines. If you insert an image into a code line, MATLAB places the image in a new text line directly under the selected code line.

Option	Description	Additional Details
 Hyperlink	Insert a hyperlink.	<p>Select External URL to insert an external URL.</p> <p>Select Internal Hyperlink to insert a hyperlink that points to an existing location within the document. When prompted, click the desired location within the document to select it as the target. Location can be a code section, text paragraph, title, or heading. Linking to individual lines of text or code is not supported.</p> <p>You can add hyperlinks only in text lines. If you insert a hyperlink into a code line, MATLAB places the hyperlink in a new text line directly under the selected code line.</p>
 Equation	Insert an equation.	You can add equations only in text lines. If you insert an equation into a code line, MATLAB places the equation in a new text line directly under the selected code line. For more information, see “Insert Equations into the Live Editor” on page 19-44.

To format existing text, use any of the options included in the **Live Editor** tab **Text** section:

Format Type	Options
Text Style	 Normal  Heading 1  Heading 2  Heading 3  Title

Format Type	Options
Text Alignment	 Left  Center  Right
Lists	 Numbered list  Bulleted list
Standard Formatting	B Bold <i>I</i> Italic <u>U</u> Underline M Monospace

To change the case of selected text or code from all uppercase to lowercase, or vice versa, select the text, right-click, and select **Change Case**. You also can press **Ctrl+Shift+A**. If the text contains both uppercase and lowercase text, MATLAB changes the case to all uppercase.

To adjust the displayed font size in the Live Editor, use the **Ctrl + Mouse Scroll** keyboard shortcut. The change in the displayed font size is not honored when exporting the live script to PDF, HTML, or LaTeX.

Autoformatting

For quick formatting in live scripts and functions, you can use a combination of keyboard shortcuts and character sequences. Formatting appears after you enter the final character in a sequence.

This table shows a list of formatting styles and their available keyboard shortcuts and autoformatting sequences.

Formatting Style	Autoformatting Sequence	Keyboard Shortcut
Title	# <i>text</i> + Enter	Ctrl + Alt + L
Heading 1	## <i>text</i> + Enter	Ctrl + Shift + 1
Heading 2	### <i>text</i> + Enter	Ctrl + Shift + 2
Heading 3	#### <i>text</i> + Enter	Ctrl + Shift + 3
Section break with heading 1	%% <i>text</i> + Enter	With cursor at beginning of line with <i>text</i> : Ctrl + Shift + 1, then Ctrl + Alt + Enter
Section break	%% + Enter - - - + Enter *** + Enter	Ctrl + Alt + Enter
Bulleted list	* <i>text</i> - <i>text</i> + <i>text</i>	Ctrl + Alt + U
Numbered list	<i>number.</i> <i>text</i>	Ctrl + Alt + O
Italic	* <i>text</i> * _ <i>text</i> _	Ctrl + I
Bold	** <i>text</i> ** __ <i>text</i> __	Ctrl + B
Bold and italic	*** <i>text</i> *** ___ <i>text</i> ___	Ctrl + B, then Ctrl + I

Formatting Style	Autoformatting Sequence	Keyboard Shortcut
Monospace	<code>`text`</code> <code> text </code>	Ctrl + M
Underline	None	Ctrl + U
LaTeX equation	<code>\$LaTeX\$</code>	Ctrl + Alt + G
Hyperlink	URL + Space or Enter <code><URL></code> <code>[Label]</code> <code>(URL)</code>	Ctrl + K
Trademark, service mark, and copyright symbols ([™] , SM , ®, and ©)	(TM) (SM) (R) (C)	None

Note Title, heading, section break, and list sequences must be entered at the beginning of a line.

Sometimes you want an autoformatting sequence such as `***` to appear literally. To display the characters in the sequence, escape out of the autoformatting by pressing the

Backspace key or by clicking **Undo** . For example, if you type `## text + Enter`, a heading in the Heading 1 style with the word `text` appears. To undo the formatting style and simply display `## text`, press the **Backspace** key. You only can escape out of a sequence directly after completing it. After you enter another character or move the cursor, escaping is no longer possible.

To revert the autoformatting for LaTeX equations and hyperlinks, use the **Backspace** key at any point.

To force formatting to reappear after escaping out of a sequence, click the **Redo**  button. You only can redo an action directly after escaping it. After you enter another character or move the cursor, the redo action is no longer possible. In this case, to force the formatting to reappear, delete the last character in the sequence and type it again.

To disable all or certain autoformatting sequences, you can adjust the “Editor/Debugger Autoformatting Preferences”. Adjusting the autoformatting preferences is not available in MATLAB Online.

See Also

More About

- “Insert Equations into the Live Editor” on page 19-44
- “Share Live Scripts and Functions” on page 19-67

Insert Equations into the Live Editor

To describe a mathematical process or method used in your code, insert equations into your live script or function. Only text lines can contain equations. If you insert an equation into a code line, MATLAB places the equation into a new text line directly under the selected code line.

Solar Elevation

The sun's declination (δ) is the angle of the sun relative to the earth's equatorial plane. The solar declination is 0° at the vernal and autumnal equinox and rises to a maximum of 23.45° at the summer solstice. On any given day of the year (d), declination can be calculated from the following formula

$$\delta = \sin^{-1} \left(\sin(23.45) \sin \left(\frac{360}{365} (d - 81) \right) \right)$$

From the declination (δ) and the latitude (ϕ) we can calculate the sun's elevation (α) at the current time.

$$\alpha = \sin^{-1} (\sin \delta \sin \phi + \cos \delta \cos \phi \cos \omega)$$

Here ω is the hour angle which is the degrees of rotation of the earth between the current solar time and solar noon.

```

delta = asind(sind(23.45)*sind(360*(d - 81)/365));           % Declination
omega = 15*(solarTime.Hour + solarTime.Minute/60 - 12);   % Hour angle
alpha = asind(sind(delta)*sind(phi) + ...                 % Elevation
    cosd(delta)*cosd(phi)*cosd(omega));
fprintf('Solar Declination = %6.2f\nSolar Elevation = %6.2f\n', delta, alpha)

```

There are two ways to insert an equation into a live script or function.

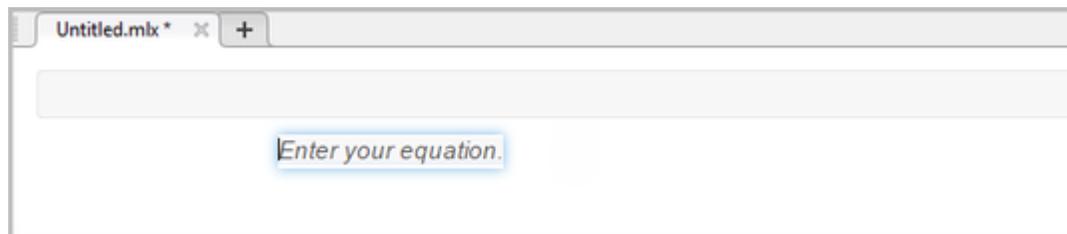
- Insert an equation interactively — You can build an equation interactively by selecting from a graphical display of symbols and structures.
- Insert a LaTeX equation — You can enter LaTeX commands and the Live Editor inserts the corresponding equation.

Insert Equation Interactively

To insert an equation interactively:

- 1 Go to the **Insert** tab and click  **Equation**.

A blank equation appears.



- 2 Build your equation by selecting symbols, structures, and matrices from the options displayed in the **Equation** tab. View additional options by clicking the \equiv to the right of each section.

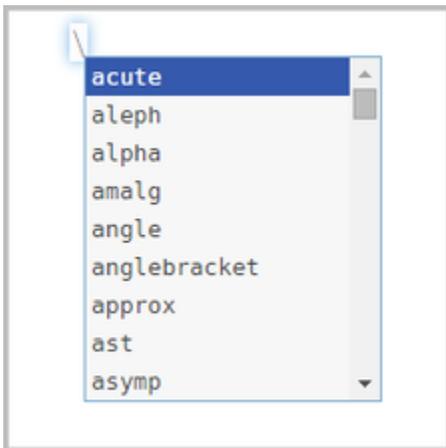
When adding or editing a matrix, a context menu appears, which you can use to delete and insert rows and columns.

- 3 Format your equation using the options available in the **Text** section. Formatting is only available for text within the equation. Numbers and symbols cannot be formatted. The formatting option is disabled unless the cursor is placed within text that can be formatted.

Keyboard Shortcuts for Equation Editing

The equation editor provides a few shortcuts for adding elements to your equation:

- To insert symbols, structures, and matrices, type a backslash followed by the name of the symbol. For example, type `\pi` to insert a π symbol into the equation. To discover the name of a symbol or structure, hover over the corresponding icon in the **Equation** tab. You can also type backslash in the equation editor to bring up a completion menu of all supported names.



Note Although the `\name` syntax closely resembles LaTeX command syntax, entering full LaTeX expressions is not supported when inserting equations interactively.

- To insert subscripts, superscripts, and fractions, use the symbols `'_'`, `'^'` or `'/'`. For example:
 - Type `x_2` to insert x_2 into the equation.
 - Type `x^2` to insert x^2 into the equation.
 - Type `x/2` to insert $\frac{x}{2}$ into the equation.
- To insert a new column into a matrix, type a `'` at the end of the last cell in a matrix row. To insert a new row, type a semicolon `';` at the end of the last cell in a matrix column.
- To insert the common symbols listed in this table, type a combination of other symbols.

Keyboard Input	Symbol	Keyboard Input	Symbol	Keyboard Input	Symbol
<code> </code>	\parallel	<code>=></code>	\Rightarrow	<code>!=</code>	\neq
<code> =</code>	\vDash	<code><--></code>	\leftrightarrow	<code>!<</code>	\nless
<code> -</code>	\vdash	<code><-></code>	\Leftrightarrow	<code>!></code>	\ngtr

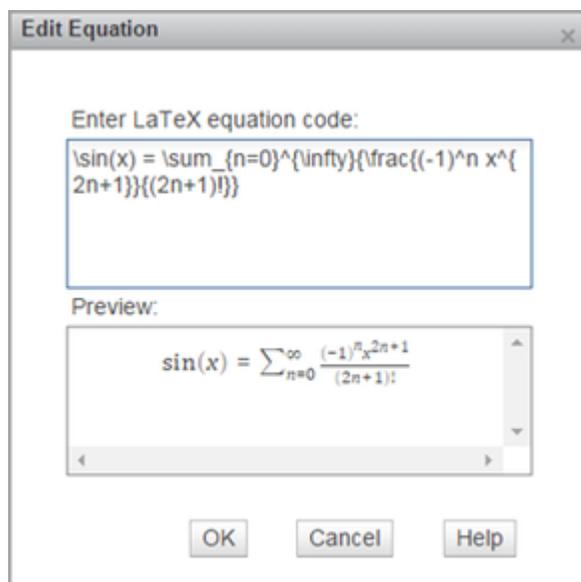
Keyboard Input	Symbol	Keyboard Input	Symbol	Keyboard Input	Symbol
-	⊥	<=	≤	!<=	≠
->	→	>=	≥	!>=	≠
<-	←	<>	≠		
<--	←	~ =	≠		

Insert LaTeX Equation

To insert a LaTeX equation:

- 1 Go to the **Insert** tab, click **Equation** and select **LaTeX Equation**.
- 2 Enter a LaTeX expression in the dialog box that appears. For example, you can enter `\sin(x) = \sum_{n=0}^{\infty} \{\frac{(-1)^n x^{2n+1}}{(2n+1)!}\}`.

The preview pane shows a preview of equation as it would appear in the live script.



- 3 Press **OK** to insert the equation into your live script.

LaTeX expressions describe a wide range of equations. This table shows several examples of LaTeX expressions and their appearance when inserted into a live script.

LaTeX Expression	Equation in Live Script
<code>a^2 + b^2 = c^2</code>	$a^2 + b^2 = c^2$
<code>\int_{0}^{2} x^2 \sin(x) dx</code>	$\int_0^2 x^2 \sin(x) dx$
<code>\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}</code>	$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$
<code>{a,b,c} \neq \{a,b,c\}</code>	$a, b, c \neq \{a, b, c\}$
<code>x^{2} \geq 0 \quad \text{for all } x \in \mathbf{R}</code>	$x^2 \geq 0 \quad \text{for all } x \in \mathbf{R}$
<code>\matrix{a & b \cr c & d}</code>	$\begin{matrix} a & b \\ c & d \end{matrix}$

Supported LaTeX Commands

MATLAB supports most standard LaTeX math mode commands. These tables show a list of supported LaTeX commands.

Greek/Hebrew Letters

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol
α	<code>alpha</code>	ν	<code>nu</code>	ξ
β	<code>beta</code>	ω	<code>omega</code>	ζ
χ	<code>chi</code>	\omicron	<code>omicron</code>	ϵ
δ	<code>delta</code>	ϕ	<code>phi</code>	φ
ϵ	<code>epsilon</code>	π	<code>pi</code>	ϖ
η	<code>eta</code>	ψ	<code>psi</code>	ϱ
γ	<code>gamma</code>	ρ	<code>rho</code>	ς
ι	<code>iota</code>	σ	<code>sigma</code>	ϑ

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol
κ	kappa	τ	tau	\varkappa
λ	lambda	θ	theta	
μ	mu	υ	upsilon	
Δ	Delta	Φ	Phi	Θ
Γ	Gamma	Π	Pi	Υ
Λ	Lambda	Ψ	Psi	Ξ
Ω	Omega	Σ	Sigma	

Operator Symbols

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol
*	ast	\pm	pm	\cap
☆	star	\mp	mp	\cup
·	cdot	\amalg	amalg	\uplus
○	circ	\odot	odot	\sqcap
•	bullet	\ominus	ominus	\sqcup
◇	diamond	\oplus	oplus	\wedge
\	setminus	\oslash	oslash	\vee
×	times	\otimes	otimes	\triangleleft
÷	div	\dagger	dagger	\triangleright
⊥	bot, perp	\ddagger	ddagger	\triangle
⊤	top	\wr	wr	\perp
Σ	sum	\amalg	prod	\int
	biguplus	\oplus	bigoplus	\sphericalangle
\cap	bigcap	\otimes	bigotimes	\wedge
\cup	bigcup	\odot	bigodot	

Relation Symbols

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol
\equiv	<code>equiv</code>	$<$	<code>lt</code>	$>$
\cong	<code>cong</code>	\leq	<code>le, leq</code>	\geq
\neq	<code>neq, ne, not=</code>	\nless	<code>not<</code>	\ngtr
\sim	<code>sim</code>	\prec	<code>prec</code>	\succ
\simeq	<code>simeq</code>	\preceq	<code>preceq</code>	\succeq
\approx	<code>approx</code>	\ll	<code>ll</code>	\gg
\asymp	<code>asymp</code>	\subset	<code>subset</code>	\supset
\doteq	<code>doteq</code>	\subseteq	<code>subseteq</code>	\supseteq
\propto	<code>propto</code>	\sqsubset	<code>sqsubseteq</code>	\sqsupset
\models	<code>models</code>	$ $	<code>mid</code>	\in
\bowtie	<code>bowtie</code>	\parallel	<code>parallel</code>	\notin
\vDash	<code>vdash</code>	\Leftrightarrow	<code>iff</code>	\ni
\dashv	<code>dashv</code>			

Note The `leq`, `geq`, `equiv`, `approx`, `cong`, `sim`, `simeq`, `models`, `ni`, `succ`, `succeq`, `prec`, `preceq`, `parallel`, `subset`, `supset`, `subseteq`, and `supseteq` commands can be combined with the `not` command to create the negated version of the symbol. For example, `\not\leq` creates the symbol \nless .

Arrows

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol
\leftarrow	<code>leftarrow</code>	\rightarrow	<code>rightarrow</code>	\uparrow
\Leftarrow	<code>Leftarrow</code>	\Rightarrow	<code>Rightarrow</code>	\Uparrow
\longleftarrow	<code>longleftarrow</code>	\longrightarrow	<code>longrightarrow</code>	\downarrow
\Lleftarrow	<code>Longleftarrow</code>	\Rrightarrow	<code>Longrightarrow</code>	\Downarrow
\hookleftarrow	<code>hookleftarrow</code>	\hookrightarrow	<code>hookrightarrow</code>	\Updownarrow

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol
\leftarrow	leftharpoondown	\rightarrow	rightharpoondown	\updownarrow
\leftarrow	leftharpoonup	\rightarrow	rightharpoonup	\leftrightarrow
\swarrow	swarrow	\nearrow	nearrow	\Leftrightarrow
\searrow	nwarrow	\swarrow	searrow	\Leftrightarrow
\mapsto	mapsto	\mapsto	longmapsto	\Leftrightarrow

Brackets

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol
{	lbrace	}	rbrace	
[lbrack]	rbrack	
<	langle	>	rangle	\
⌈	lceil	⌋	rceil	
⌊	lfloor	⌋	rfloor	

Sample	LaTeX Command	Sample	LaTeX Command	
{	big, bigl, bigr, bigm	{abc}	brace	
{	Big, Bigl, Bigr, Bigm	[abc]	brack	
{	bigg, biggl, biggr, biggm	(abc)	choose	
{	Bigg, Biggl, Biggr, Biggm			

Misc Symbols

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol
∞	<code>infty</code>	\forall	<code>forall</code>	\wp
∇	<code>nabla</code>	\exists	<code>exists</code>	\angle
∂	<code>partial</code>	\emptyset	<code>emptyset</code>	Δ
\Im	<code>Im</code>	\mathbf{i}	<code>i</code>	\hbar
\Re	<code>Re</code>	\mathbf{j}	<code>j</code>	\prime
ℓ	<code>ell</code>	\mathbf{l}	<code>lmath</code>	\neg
\dots	<code>dots, ldots</code>	\mathbf{J}	<code>jmath</code>	$\sqrt{\quad}$
\cdots	<code>cdots</code>	$:$	<code>colon</code>	\leftarrow
\ddots	<code>ddots</code>	\cdot	<code>cdotp</code>	\rightarrow
\vdots	<code>vdots</code>	\cdot	<code>ldotp</code>	

Note The `exists` command can be combined with the `not` command to create the negated version of the symbol. For example, `\not\exists` creates the symbol \nexists .

Accents

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol
\acute{a}	<code>acute</code>	\ddot{a}	<code>ddot</code>	\tilde{a}
\bar{a}	<code>bar</code>	\dot{a}	<code>dot</code>	\bar{a}
\breve{a}	<code>breve</code>	\grave{a}	<code>grave</code>	
\check{a}	<code>check</code>	\hat{a}	<code>hat</code>	

Functions

Sample	LaTeX Command	Sample	LaTeX Command	Sample
\arccos	<code>\arccos</code>	\det	<code>\det</code>	\ln
\arcsin	<code>\arcsin</code>	\dim	<code>\dim</code>	\log
\arctan	<code>\arctan</code>	\exp	<code>\exp</code>	\max
\arg	<code>\arg</code>	\gcd	<code>\gcd</code>	\min
\cos	<code>\cos</code>	\hom	<code>\hom</code>	\Pr
\cosh	<code>\cosh</code>	\ker	<code>\ker</code>	\sec
\cot	<code>\cot</code>	\lg	<code>\lg</code>	\sin
\coth	<code>\coth</code>	\lim	<code>\lim</code>	\sinh
\csc	<code>\csc</code>	\liminf	<code>\liminf</code>	\sup
\deg	<code>\deg</code>	\limsup	<code>\limsup</code>	\tan

Math Constructs

Sample	LaTeX Command	Sample	LaTeX Command	Sample
$\frac{abc}{xyz}$	<code>\frac</code>	$\frac{a}{b}$	<code>\over</code>	$\frac{a}{b}$
\sqrt{abc}	<code>\sqrt</code>	$\left[\frac{a}{b} \right]$	<code>\overwithdelims</code>	$\frac{b}{a}$
$\bmod a$	<code>\bmod</code>	\overline{abc}	<code>\overleftarrow</code>	$\begin{pmatrix} a & b \\ e & d \end{pmatrix}$
$(\bmod a)$	<code>\pmod</code>	\overline{abc}	<code>\overrightarrow</code>	$a \ b$ $e \ d$

Sample	LaTeX Command	Sample	LaTeX Command	Sample
\widehat{abc}	widehat	\overline{abc}	overleftar row	$a \ b$ $e \ d$
\widetilde{abc}	widetilde	\int_a^b	limits	$\left\{ \begin{array}{l} a \ b \\ c \ d \end{array} \right.$
	left		right	\overline{ab} \underline{ed}

Note To create a matrix using the `matrix` and `pmatrix` commands, use the `&` symbol to separate columns, and `\cr` to separate rows. For example, to create a 2-by-2 matrix, use the expression `\matrix{a & b \cr c & d}`.

White Space

Sample	LaTeX Command	Sample	LaTeX Command	Sample
$a\!b$	negthinspace	abc	mathord	$a\! b$
ab	thinspace	$a\sum b$	mathop	$a\! b$
$a \ b$	enspace	$a + b$	mathbin	$a b$
$a \ b$	quad	$a = b$	mathrel	$a \ b$
$a \quad b$	qqquad	a, b	mathpunct	

Text Styling

Sample	LaTeX Command	Sample	LaTeX Command	Sample
Σ	displaystyle	ABCDE	text, textnormal	ABCDE

Sample	LaTeX Command	Sample	LaTeX Command	Sample
Σ	<code>textstyle</code>	ABCDE	<code>bf, textbf, mathbf</code>	ABCDE
Σ	<code>scriptstyle</code>	<i>ABCDE</i>	<code>it, textit, mathit</code>	A B C D E
Σ	<code>scriptscriptstyle</code>	ABCDE	<code>rm, textrm, mathrm</code>	ABCDE

See Also

Related Examples

- “Share Live Scripts and Functions” on page 19-67

External Websites

- <https://www.latex-project.org/>

Add Interactive Controls to a Live Script

You can add sliders and drop-downs to your live scripts to control variable values interactively. Adding interactive controls to a script is useful when you want to share the script with others. Use interactive controls to set and change the values of variables in your live script using familiar user interface components such as numeric sliders and drop-down lists.

Insert Numeric Slider

A numeric slider allows you to interactively change the value of a variable by moving the slider thumb to the desired numeric value. To insert a numeric slider, use these steps:

- 1 Select an existing numeric value (optional).
- 2 Go to the **Insert** tab, click **Control**, and select **Numeric Slider**.
- 3 Specify a minimum value, a maximum value, and a step size, and then press **Enter**.



The value to the left of the slider is the current value of the slider.

Insert Drop-Down List

A drop-down list allows you to interactively change the value of a variable by selecting from a list of text values. To insert a drop-down list, use these steps:

- 1 Select an existing text value (optional).
- 2 Go to the **Insert** tab, click **Control**, and select **Drop Down**.
- 3 Enter additional lines of text and press **Tab**. Make sure to enclose text values in quotes, because the Live Editor interprets each item in the list as code.



The text displayed in the drop-down list is its current value.

Create Live Script with Multiple Interactive Controls

This example shows how you can use interactive controls to visualize and investigate patient data in MATLAB®. The example plots the height versus the weight of either male or female patients, and highlights the patients over a specified height and weight.

Use the interactive controls to specify the gender of the patients to plot, as well as the threshold height and weight. To view and interact with the controls, open this example in your browser or in MATLAB.

```
load patients

thresholdHeight = 68; % Slider with min=60, max=70, step=1
thresholdWeight = 132; % Slider with min=111, max=202, step=1
selectedGender = "Female"; % Drop down with options "Male", "Female"

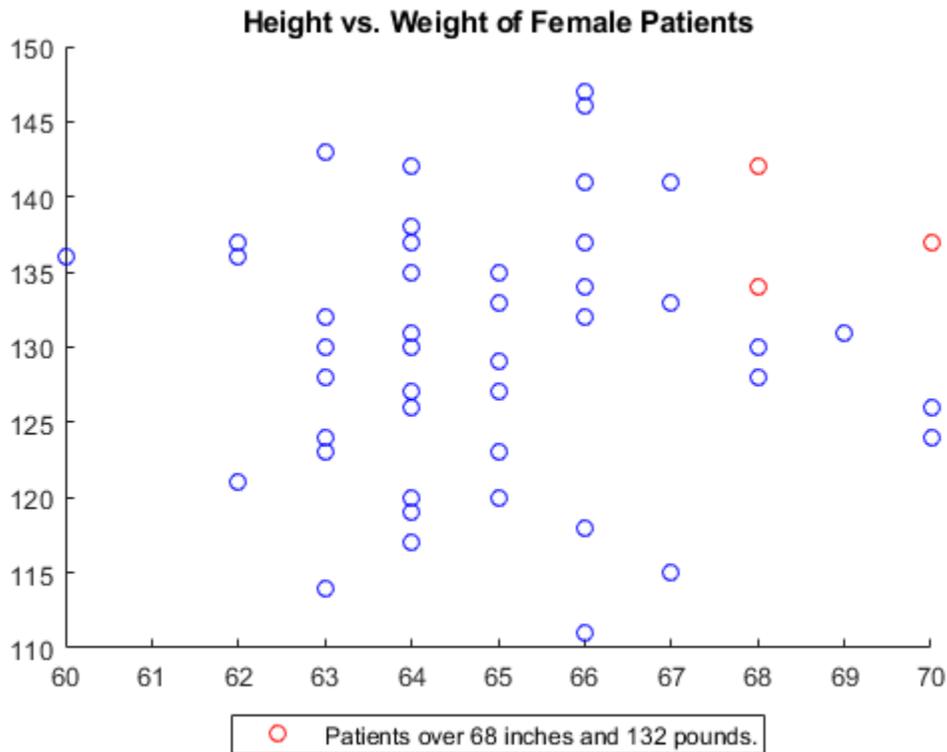
overThresholdWeights = Weight(Gender==selectedGender & Weight>=thresholdWeight & Height>=thresholdHeight);
overThresholdHeights = Height(Gender==selectedGender & Weight>=thresholdWeight & Height>=thresholdHeight);

sp1 = scatter(Height(Gender==selectedGender),Weight(Gender==selectedGender),'blue');
hold on

sp2 = scatter(overThresholdHeights, overThresholdWeights,'red');
hold off

title('Height vs. Weight of ' + selectedGender + ' Patients')

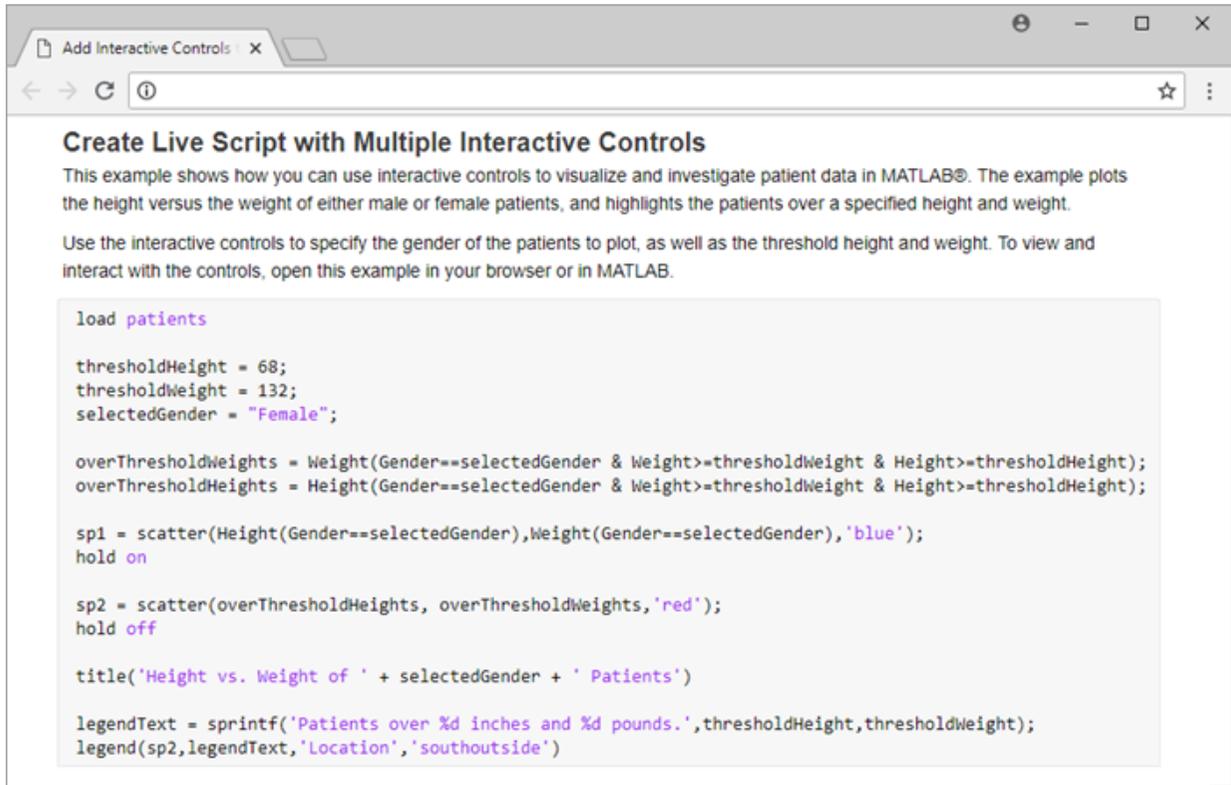
legendText = sprintf('Patients over %d inches and %d pounds.',thresholdHeight,thresholdWeight);
legend(sp2,legendText,'Location','southoutside')
```



Share Live Script

When the live script is ready, share it with others. Users can open the live script in MATLAB and experiment with using the controls interactively. Run the live script to see the updated results.

If you share the live script as a static PDF, HTML, or LaTeX document, the Live Editor saves the control as code. For example, in the live script shown here, the Live Editor replaces the slider controls with their current value (68 and 132) and replaces the drop-down control with the current value of the drop-down ("Female").



Create Live Script with Multiple Interactive Controls

This example shows how you can use interactive controls to visualize and investigate patient data in MATLAB®. The example plots the height versus the weight of either male or female patients, and highlights the patients over a specified height and weight.

Use the interactive controls to specify the gender of the patients to plot, as well as the threshold height and weight. To view and interact with the controls, open this example in your browser or in MATLAB.

```
load patients

thresholdHeight = 68;
thresholdWeight = 132;
selectedGender = "Female";

overThresholdWeights = Weight(Gender==selectedGender & Weight>=thresholdWeight & Height>=thresholdHeight);
overThresholdHeights = Height(Gender==selectedGender & Weight>=thresholdWeight & Height>=thresholdHeight);

sp1 = scatter(Height(Gender==selectedGender),Weight(Gender==selectedGender),'blue');
hold on

sp2 = scatter(overThresholdHeights, overThresholdWeights,'red');
hold off

title('Height vs. Weight of ' + selectedGender + ' Patients')

legendText = sprintf('Patients over %d inches and %d pounds.',thresholdHeight,thresholdWeight);
legend(sp2,legendText,'Location','southoutside')
```

See Also

More About

- “Share Live Scripts and Functions” on page 19-67

Create Live Functions

Live functions are program files that contain code and formatted text together in a single interactive environment called the Live Editor. Similar to live scripts, live functions allow you to reuse sequences of commands by storing them in program files. Live functions provide more flexibility, though, primarily because you can pass them input values and receive output values.

Create Live Function

To create a live function, go to the **Home** tab and select **New > Live Function**.

To open an existing function (.m) as a live function (.mlx) from the Editor, right-click the document tab and select **Open *functionName* as Live Function** from the context menu. Alternatively, go to the **Editor** tab, click **Save**, and select **Save As**. Then, set the **Save as type**: to MATLAB Live Code Files (*.mlx) and click **Save**. Opening a function as a live function creates a copy of the file and leaves the original file untouched. MATLAB converts publishing markup from the original function to formatted content in the new live function.

Note You must use one of the described conversion methods to convert your function to a live function. Simply renaming the function with a .mlx extension does not work and can corrupt the file.

Add Code

After you create the live function, add code to the function and save it. For example, add this code and save it as a function called `mymean.mlx`. The `mymean` function calculates the average of the input list and returns the results.

```
function a = mymean(v,n)
    a = sum(v)/n;
end
```

Add Help

To document the function, add formatted help text above the function definition. For example, add a title and some text to describe the functionality. For more information about adding help text to functions, see “Add Help for Live Functions” on page 19-63.

Mean value for a set of values

Get the mean value for a set of values by calculating the sum of all the values and dividing by the total number of values.

```
1 function a = mymean(v,n)
2     a = sum(v)/n;
3 end
```

Run Code

To run the live function, enter the name of the function in the Command Window. For example, use `mymean.mlx` to calculate the mean of 10 sequential numbers from 1 to 10.

```
mymean(1:10, 10)
```

```
ans =
    5.5000
```

You also can call the live function from a live script. For example, create a live script called `mystats.mlx`. Add this code that declares an array, determines the length of the array, and passes both values to the function `mymean`.

```
x = 1:10;
n = length(x);
avg = mymean(x,n);
disp(['Average = ', num2str(avg)])
```

Run the live script. The Live Editor displays the output.

<pre>1 x = 1:10; 2 n = length(x); 3 avg = mymean(x,n); 4 disp(['Average = ', num2str(avg)])</pre>	<p>Average = 5.5</p>
---	----------------------

If a live function displays text or returns values, the Live Editor displays the output in the calling live script, in line with the call to the live function. For example, add a line to `mymean` that displays the calculated mean before returning the value:

```
function a = mymean(v,n)
    a = sum(v)/n;
    disp(['a = ', num2str(a)])
end
```

When you run `mystats`, the Live Editor displays the output for `mymean` with the output from `mystats`.

<pre>1 2 3 4</pre>	<pre>x = 1:10; n = length(x); avg = mymean(x,n); disp(['Average = ', num2str(avg)])</pre>	<pre>a = 5.5 Average = 5.5</pre>
--------------------	---	----------------------------------

See Also

More About

- “Add Help for Live Functions” on page 19-63

Add Help for Live Functions

You can provide help for the live functions you write. Help text appears in the Command Window when you use the `help` command. You also can use the `doc` command to display the help text in a separate browser.

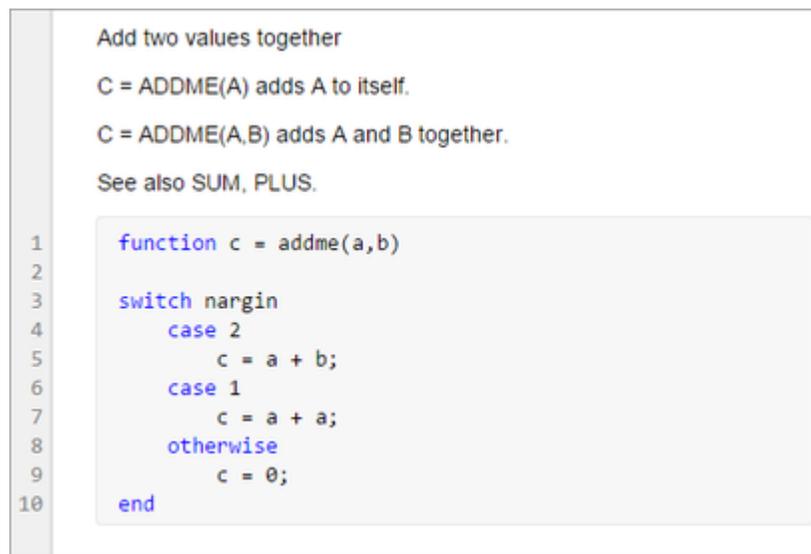
Create help text by inserting text at the beginning of the file, immediately before the function definition line (the line with the `function` keyword).

For example, create a live function called `addme.mlx` with this code:

```
function c = addme(a,b)

switch nargin
    case 2
        c = a + b;
    case 1
        c = a + a;
    otherwise
        c = 0;
end
```

Add help text to describe the function.



The screenshot shows the help text for the `addme` function. The text is displayed in a light gray box with a vertical line on the left side. The help text includes a title, two examples of function calls, and a reference to other functions. Below the help text is a code block showing the function definition.

```
Add two values together

C = ADDME(A) adds A to itself.
C = ADDME(A,B) adds A and B together.

See also SUM, PLUS.

1  function c = addme(a,b)
2
3  switch nargin
4      case 2
5          c = a + b;
6      case 1
7          c = a + a;
8      otherwise
9          c = 0;
10 end
```

When you type `help addme` in the Command Window, the help text displays.

```
>> help addme
addme    Add two values together
        c = addme(a,b)

        C = addme(A) adds A to itself.

        C = addme(A,B) adds A and B together.

See also sum, plus.

Open documentation in Help browser
```

The first line of help text, often called the H1 line, typically contains a brief description of the function. When displaying help for a function, MATLAB first displays the name of the function followed by the H1 line. Then, MATLAB displays the syntax of the function. Finally, MATLAB displays any remaining help text.

To add "See also" links, add a text line at the end of the help text that begins with the words `See also` followed by a list of function names. If the functions exist 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, `help` prints the function names as they appear in the help text.

Note When multiple programs have the same name, the `help` command determines which help text to display by applying the rules described in "Function Precedence Order" on page 20-43. However, if a program has the same name as a built-in function, the **Help on Selection** option in context menus always displays documentation for the built-in function.

To enhance the documentation displayed in the Help browser further, you can format the text and add hyperlinks, images, equations, and example code. For example, in the `addme` function, select the H1 line and in the **Live Editor** tab, change the **Normal** text style to **Title**. Then, position your cursor at the end of the second syntax description, go to the **Insert** tab and select **Σ Equation**. Enter the equation `c = a + b` and press **Esc**. Finally, in the **Insert** tab, select **Code Example > MATLAB** and add two examples. For more information about formatting files in the Live Editor, see "Format Files in the Live Editor" on page 19-38.

Add two values together

C = ADDME(A) adds A to itself.

C = ADDME(A,B) adds A and B together using the equation $c = a + b$.

Examples

Add a value to itself:

```
c = addme(10);
```

Add two values:

```
c = addme(10, 90);
```

See also SUM, PLUS.

```
1 function c = addme(a,b)
2
3
4 switch nargin
5     case 2
6         c = a + b;
7     case 1
8         c = a + a;
9     otherwise
10        c = 0;
end
```

Use the `doc` command to display the help text in a separate browser.

Documentation

addme

Add two values together

Syntax

```
c = addme(a,b)
```

Description

C = ADDME(A) adds A to itself.

C = ADDME(A,B) adds A and B together using the equation $c = a + b$.

Examples

Add a value to itself:

```
c = addme(10);
```

Add two values:

```
c = addme(10, 90);
```

See also SUM, PLUS.

See Also

More About

- “Create Live Functions” on page 19-60
- “Format Files in the Live Editor” on page 19-38
- “Share Live Scripts and Functions” on page 19-67

Share Live Scripts and Functions

You can share live scripts and functions with others for teaching or demonstration purposes, or to provide readable, external documentation of your code. You can share the files with other MATLAB users, or you can share static PDF, HTML, and LaTeX versions of the files for viewing outside of MATLAB.

This table shows the different ways to share live scripts and functions.

Ways to Share	Instructions
As an interactive document	<p>Distribute the live code file (.mlx). Recipients of the file can open and view the file in MATLAB in the same state that you last saved it in. This includes generated output.</p> <p>MATLAB supports live scripts in Versions R2016a and above, and live functions in Versions R2018a and above.</p>
As a full screen presentation	<p>Open the live script or function, go to the View tab and click the Full Screen button. MATLAB opens the file in full screen mode.</p> <p>To exit out of full screen mode, move the mouse to the top of the screen to display the View tab and click the Full Screen button again.</p>
With users of previous MATLAB versions	<p>Save the live script or function as a plain code file (.m) and distribute it. Recipients of the file can open and view the file in MATLAB. MATLAB converts formatted content from the live script or function to publish markup in the new script or function.</p> <p>For more information, see “Save Live Scripts and Functions as Plain Code” on page 19-6.</p>

Ways to Share	Instructions
As a static document capable of being viewed outside of MATLAB	<p>Export the script or function to a standard format. Available formats include PDF, HTML, and LaTeX.</p> <p>To export your live script or function to one of these formats, on the Live Editor tab, select Save > Export to PDF, Save > Export to HTML, or Save > Export to LaTeX.</p> <p>The saved file closely resembles the appearance of your live script or function when viewed in the Live Editor with output inline.</p> <p>When exporting to LaTeX, MATLAB creates a separate Extensible Stylesheet Language (XSL) file in the same folder as the output document, if one does not exist already. XSL files give you more control over the appearance of the output document. For more details, see http://docbook.sourceforge.net/release/xsl/current/doc/.</p>

See Also

Related Examples

- “Create Live Scripts in the Live Editor” on page 19-7
- “Format Files in the Live Editor” on page 19-38

Live Code File Format (.mlx)

MATLAB stores live scripts and functions using the Live Code file format in a file with a `.mlx` extension. The Live Code file format uses Open Packaging Conventions technology, which is an extension of the zip file format. Code and formatted content are stored in an XML document separate from the output using the Office Open XML (ECMA-376) format.

Benefits of Live Code File Format

- **Interoperable Across Locales** — Live code files support storing and displaying characters across all locales, facilitating sharing files internationally. For example, if you create a live script with a Japanese locale setting, and open the live script with a Russian locale setting, the characters in the live script display correctly.
- **Extensible** — Live code files can be extended through the ECMA-376 format, which supports the range of formatting options offered by Microsoft Word. The ECMA-376 format also accommodates arbitrary name-value pairs, should there be a need to extend the format beyond what the standard offers.
- **Forward Compatible** — Future versions of live code files are compatible with previous versions of MATLAB by implementing the ECMA-376 standard's forward compatibility strategy.
- **Backward Compatible** — Future versions of MATLAB can support live code files created by a previous version of MATLAB.

Source Control

To determine and display code differences between live scripts or functions, use the MATLAB Comparison Tool.

If you use source control, register the `.mlx` extension as binary. For more information, see “Register Binary Files with SVN” on page 31-18 or “Register Binary Files with Git” on page 31-32.

See Also

More About

- “What Is a Live Script or Function?” on page 19-2
- “Create Live Scripts in the Live Editor” on page 19-7

External Websites

- Open Packaging Conventions Fundamentals
- Office Open XML File Formats (ECMA-376)

Introduction to the Live Editor

This example is an introduction to the Live Editor. In the Live Editor, you can create live scripts that show output together with the code that produced it. Add formatted text, equations, images, and hyperlinks to enhance your narrative, and share the live script with others as an interactive document.

*Create a live script in the Live Editor. To create a live script, on the **Home** tab, in the **New** drop-down menu, select **Live Script**.*

Add the Census Data

*Divide your live script into sections. Sections can contain text, code, and output. MATLAB code appears with a gray background and output appears with a white background. To create a new section, go to the **Live Editor** tab and click the **Section Break** button.*

Add the US Census data for 1900 to 2000.

```
years = (1900:10:2000); % Time interval
pop = [75.995 91.972 105.711 123.203 131.669 ... % Population Data
      150.697 179.323 213.212 228.505 250.633 265.422]

pop = 1×11

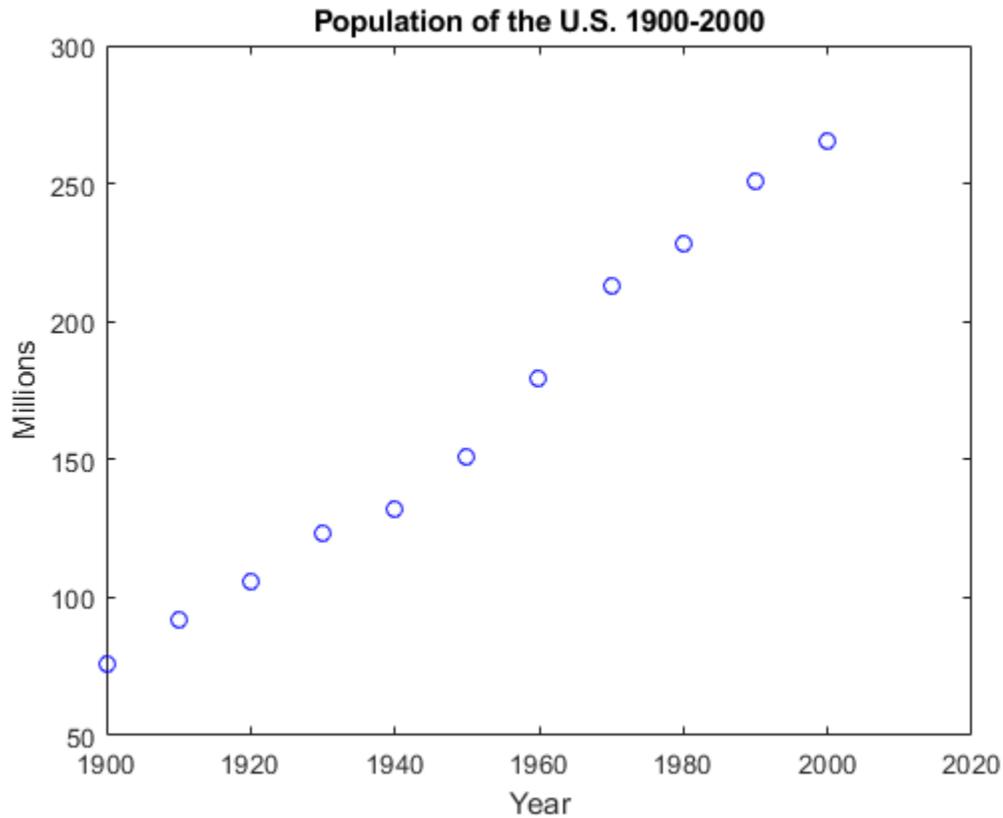
      75.9950      91.9720      105.7110      123.2030      131.6690      150.6970      179.3230      213.2120      228.5050      250.6330      265.4220
```

Visualize the Population Change Over Time

*Sections can be run independently. To run the code in a section, go to the **Live Editor** tab and click the **Run Section** button. You can also click the blue bar that appears when you move the mouse to the left of the section. When you run a section, output and figures appear together with the code that produced them.*

Plot the population data against the year.

```
plot(years,pop,'bo'); % Plot the population data
axis([1900 2020 0 400]);
title('Population of the U.S. 1900-2000');
ylabel('Millions');
xlabel('Year')
ylim([50 300])
```



Can we predict the US population in the year 2010?

Fitting the Data

Add supporting information to the text, including equations, images, and hyperlinks.

Let's try fitting the data with polynomials. We'll use the MATLAB `polyfit` function to get the coefficients.

The fit equations are:

$$y = ax + b \quad \text{linear}$$

$$y = ax^2 + bx + c \quad \text{quadratic}$$

$$y = ax^3 + bx^2 + cx + d. \quad \text{cubic}$$

```
x = (years-1900)/50;
coef1 = polyfit(x,pop,1)
```

```
coef1 = 1×2
```

```
98.9924 66.1296
```

```
coef2 = polyfit(x,pop,2)
```

```
coef2 = 1×3
```

```
15.1014 68.7896 75.1904
```

```
coef3 = polyfit(x,pop,3)
```

```
coef3 = 1×4
```

```
-17.1908 66.6739 29.4569 80.1414
```

Plotting the Curves

Create sections with any number of text and code lines.

We can plot the linear, quadratic, and cubic curves fitted to the data. We'll use the `polyval` function to evaluate the fitted polynomials at the points in `x`.

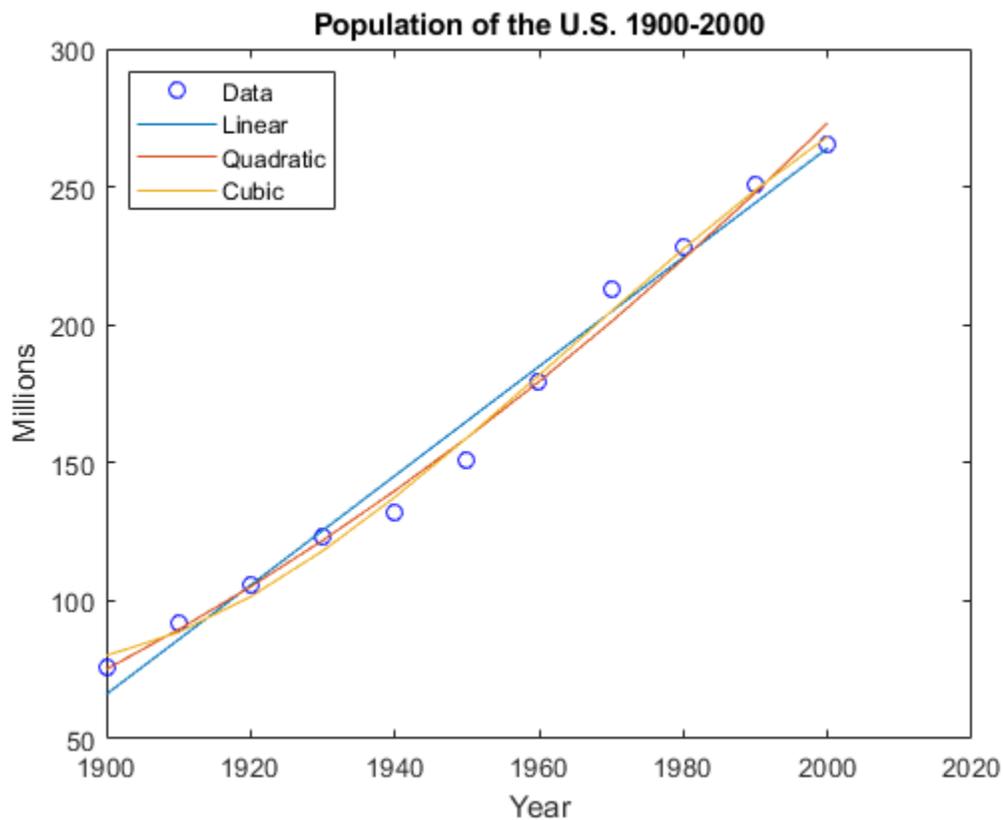
```
pred1 = polyval(coef1,x);
pred2 = polyval(coef2,x);
pred3 = polyval(coef3,x);
[pred1; pred2; pred3]
```

```
ans = 3×11
```

```
66.1296 85.9281 105.7266 125.5250 145.3235 165.1220 184.9205 204.7190 224.5175
75.1904 89.5524 105.1225 121.9007 139.8870 159.0814 179.4840 201.0946 223.9071
80.1414 88.5622 101.4918 118.1050 137.5766 159.0814 181.7944 204.8904 227.5968
```

Now let's plot the predicted values for each polynomial.

```
hold on
plot(years,pred1)
plot(years,pred2)
plot(years,pred3)
ylim([50 300])
legend({'Data' 'Linear' 'Quadratic' 'Cubic'},'Location', 'NorthWest')
hold off
```



Predicting the Population

You can share your live script with other MATLAB users so they can reproduce your results. You can also publish your results as PDF or HTML.

We can now calculate the predicted population in 2010 using our three equations.

```
x2010 = (2010-1900)/50;  
pred1 = polyval(coef1,x2010);  
pred2 = polyval(coef2,x2010);  
pred3 = polyval(coef3,x2010);  
[pred1 pred2 pred3]  
  
ans = 1×3  
  
    283.9129    299.6184    284.6005
```

The linear and cubic fits predict similar values of about 284 million people, while the quadratic fit predicts a much higher value of about 300 million people.

Use the Live Editor to Accelerate Exploratory Programming

The following is an example of how to use the Live Editor to accelerate exploratory programming. This example demonstrates how you can use the Live Editor to:

- See output together with the code that produced it.
- Divide your program into sections to evaluate blocks of code individually.
- Include visualizations.
- Experiment with parameter values.
- Summarize and share your findings.

Load Highway Fatality Data

The Live Editor displays output together with the code that produced it. To run a section, go to the **Live Editor** tab and select the **Run Section** button. You can also click the blue bar that appears when you move your mouse to the left edge of a section.

In this example, we explore some highway fatality data. Start by loading the data. The variables are shown as the column headers of the table.

```
load fatalities
fatalities(1:10,:)
```

ans=10×8 table

	longitude	latitude	deaths	drivers	vehicles
Wyoming	-107.56	43.033	164	380.18	671.53
District_of_Columbia	-77.027	38.892	43	349.12	240.4
Vermont	-72.556	44.043	98	550.46	551.52
North_Dakota	-99.5	47.469	100	461.78	721.84
South_Dakota	-99.679	44.272	197	563.3	882.77
Delaware	-75.494	39.107	134	533.94	728.52
Montana	-110.58	46.867	229	712.88	1056.7
Rhode_Island	-71.434	41.589	83	741.84	834.5
New_Hampshire	-71.559	43.908	171	985.77	1244.6
Maine	-69.081	44.886	194	984.83	1106.8

Calculate Fatality Rates

The Live Editor allows you to divide your program into sections containing text, code, and output. To create a new section, go to the **Live Editor** tab and click the **Section Break** button. The code in a section can be run independently, which makes it easy to explore ideas as you write your program.

Calculate the fatality rate per one million vehicle miles. From these values we can find the states with the lowest and highest fatality rates.

```
states = fatalities.Properties.RowNames;
rate = fatalities.deaths./fatalities.vehicleMiles;
[~, minIdx] = min(rate);           % Minimum accident rate
[~, maxIdx] = max(rate);          % Maximum accident rate
disp([states{minIdx} ' has the lowest fatality rate at ' num2str(rate(minIdx))])
```

```
Massachusetts has the lowest fatality rate at 0.0086907
```

```
disp([states{maxIdx} ' has the highest fatality rate at ' num2str(rate(maxIdx))])
```

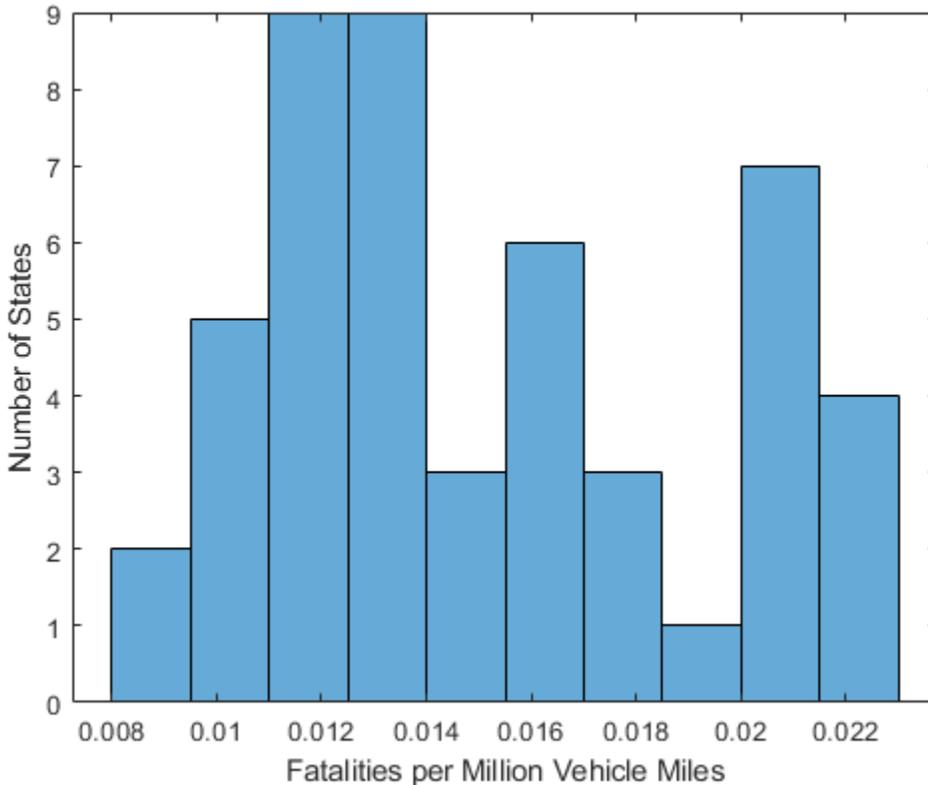
```
Mississippi has the highest fatality rate at 0.022825
```

Distribution of Fatalities

You can include visualizations in your program. Like output, plots and figures appear together with the code that produced them.

We can use a bar chart to see the distribution of fatality rates among the states. There are 11 states that have a fatality rate greater than 0.02 per million vehicle miles.

```
histogram(rate,10)
xlabel('Fatalities per Million Vehicle Miles')
ylabel('Number of States')
```



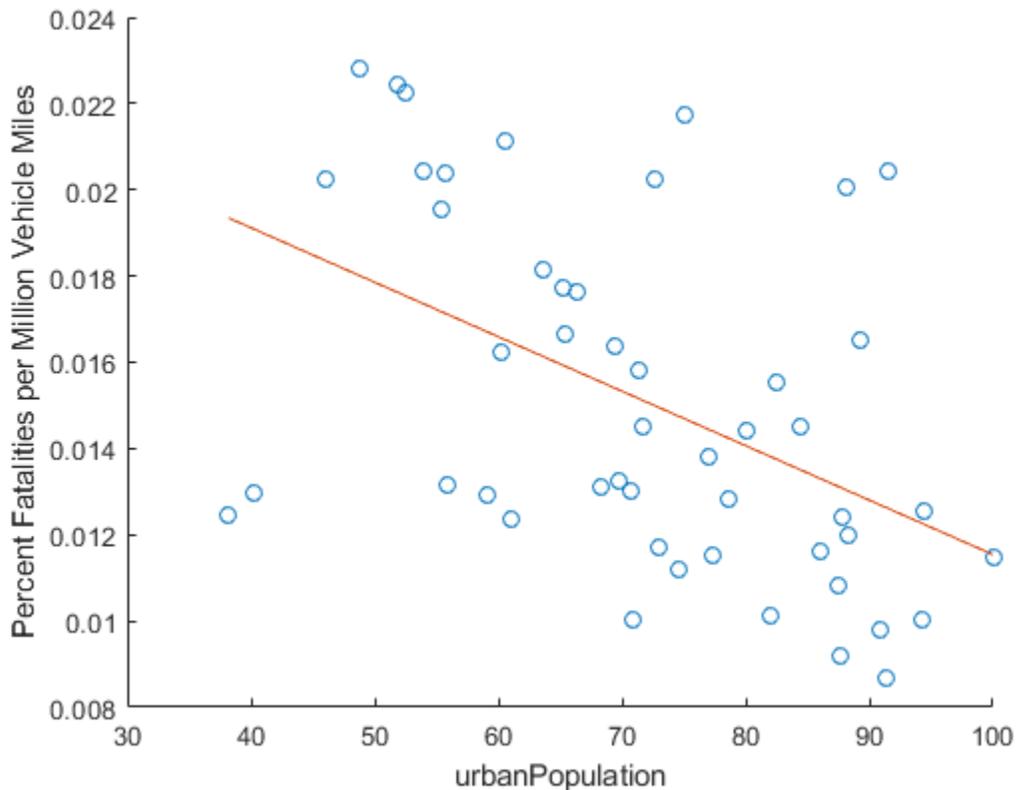
Find Correlations in the Data

You can explore your data quickly in the Live Editor by experimenting with parameter values to see how your results will change.

We can experiment with the data to see if any of the variables in the table are correlated with highway fatalities. It appears that highway fatality rates are lower in states with a higher percentage urban population.

```
varName = 'urbanPopulation';  
scatter(fatalities.(varName),rate)           % Plot fatalities vs. selected variable  
xlabel(varName)  
ylabel('Percent Fatalities per Million Vehicle Miles')
```

```
hold on
xmin = min(fatalities.(varName));
xmax = max(fatalities.(varName));
p = polyfit(fatalities.(varName),rate,1); % Calculate & plot least squares line
plot([xmin xmax], polyval(p,[xmin xmax]))
```

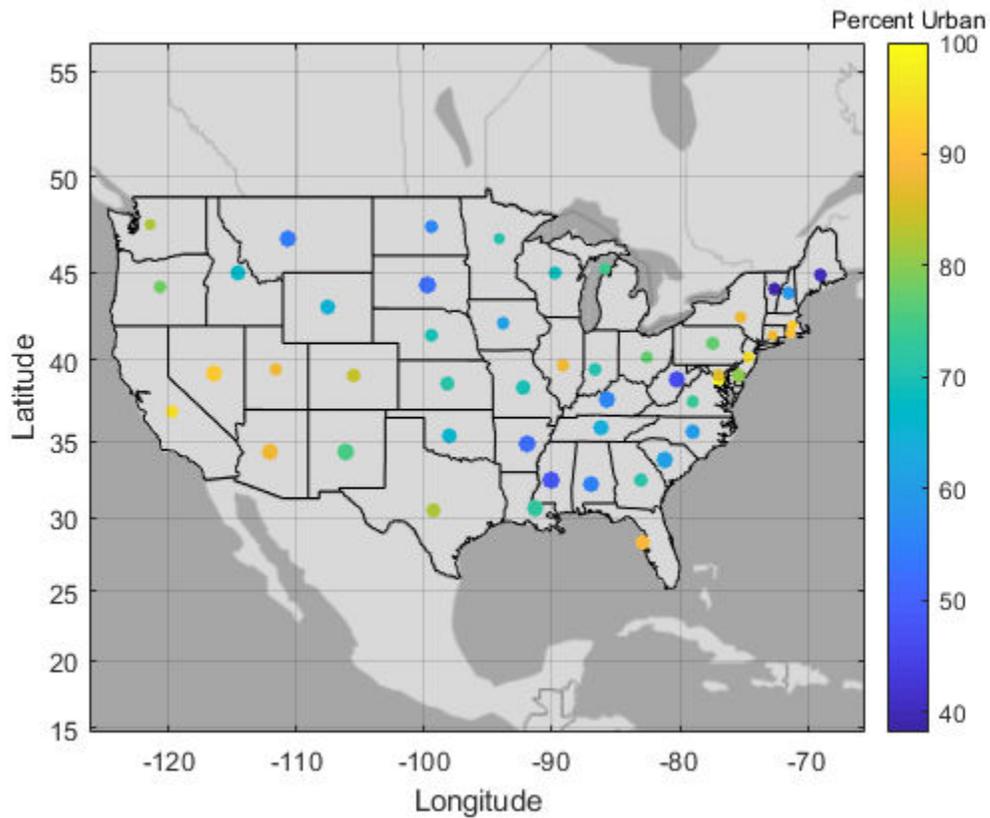


Plot Fatalities and Urbanization on a US Map

Summarize your results and share your live script with your colleagues. Using your live script, they can recreate and extend your analysis. You can also save your analysis as HTML or PDF for publication.

Based on this analysis, we can summarize our findings using a plot of fatality rates and urban population on a map of the continental United States.

```
load usastates.mat
figure
geoplot([usastates.Lat], [usastates.Lon], 'black')
hold on
geoscatter(fatalities.latitude, fatalities.longitude, 2000*rate, fatalities.urbanPopulation)
c = colorbar;
title(c, 'Percent Urban')
```



Create an Interactive Narrative with the Live Editor

The following is an example of how to use the Live Editor to create an interactive narrative. Create an interactive narrative to tell a story with computations that you have used to solve a problem. This example shows how to:

- *Use formatted text to describe your approach.*
- *Show output together with your MATLAB code.*
- *Use equations to describe the underlying mathematics.*
- *Include images to illustrate important points.*
- *Provide hyperlinks to background material.*
- *Let the reader modify parameters and re-run the analysis.*
- *Include plots for visualization.*
- *Invite colleagues to extend your analysis.*

The Overall Approach

In the Live Editor, you can include formatted text as part of the interactive narrative. Use bold, italic, and underlined text to highlight important words or ideas. Use bullets or numbers to format lists.

In this example we will estimate the **power output** from a typical solar panel installation. We will use 12 noon on June 1st in Boston to illustrate how to calculate the following:

- Solar time
- Solar declination and solar elevation
- Air mass and the solar radiation reaching the earth's surface
- Radiation on a solar panel given its position, tilt, and efficiency
- Power generated in a day and over the entire year

We will use these formulas to plot solar and panel radiation for our example day, and then plot the expected panel power generation over the course of a year. We'll use two MATLAB functions created for this analysis, `solarCorrection` and `hourlyPanelRadiation`, to streamline the analysis.

Solar Time

Show output together with the code that produced it. To run a section of code, go to the **Live Editor** tab and click the **Run Section** button.

Power generation in a solar panel depends on how much solar radiation reaches the panel which in turn depends on the sun's position relative to the panel as the sun moves across the sky.

```
lambda = -71.06;           % Boston longitude
phi = 42.36;              % Boston latitude
UTCoff = -5;              % Boston UTC offset
TZ = ['UTC' num2str(UTCoff)];
january1 = datetime(2016,1,1,'TimeZone',TZ); % January 1st
localTime = datetime(2016,6,1,12,0,0,'TimeZone',TZ) % Noon on June 1

localTime = datetime
    01-Jun-2016 12:00:00
```

To calculate the sun's position for a given date and time we need to use *solar time*. Twelve noon solar time is defined to be the time when the sun is highest in the sky. To calculate solar time, we apply a correction to local time. That correction has two parts:

- A term which corrects for the difference between the observer's location and the local meridian
- An orbital term related to the eccentricity of the earth's orbit and its axial tilt

We'll use a MATLAB function created for this analysis called `solarCorrection`. For example, at noon on June 1, the solar time would be:

```
d = caldays(between(january1,localTime,'Day')); % Day of year
solarCorr = solarCorrection(d,lambda,UTCoff); % Correction to local time
solarTime = localTime + minutes(solarCorr)

solarTime = datetime
    01-Jun-2016 12:18:06
```

Solar Declination and Elevation

Include equations to describe the underlying mathematics. Create equations using LaTeX commands. To add a new equation, go to the **Live Editor** tab and click the **Equation** button. Double-click an equation to edit it in the Equation Editor.

The sun's declination (δ) is the angle of the sun relative to the earth's equatorial plane. The solar declination is 0° at the vernal and autumnal equinoxes and rises to a maximum of 23.45° at the summer solstice. On any given day of the year (d), declination can be calculated from the following formula:

$$\delta = \sin^{-1} \left(\sin(23.45) \sin \left(\frac{360}{365} (d - 81) \right) \right)$$

From the declination (δ) and the latitude (ϕ) we can calculate the sun's elevation (α) at the current time.

$$\alpha = \sin^{-1} (\sin \delta \sin \phi + \cos \delta \cos \phi \cos \omega)$$

Here ω is the *hour angle*, which is the number of degrees of rotation of the earth between the current solar time and solar noon.

```
delta = asind(sind(23.45)*sind(360*(d - 81)/365));           % Declination
omega = 15*(solarTime.Hour + solarTime.Minute/60 - 12);    % Hour angle
alpha = asind(sind(delta)*sind(phi) + ...                  % Elevation
            cosd(delta)*cosd(phi)*cosd(omega));
disp(['Solar Declination = ' num2str(delta) '   Solar Elevation = ' num2str(alpha)])

Solar Declination = 21.9634   Solar Elevation = 69.2589
```

Knowing the sun's declination and the local latitude allows us to calculate the time of sunrise and sunset. Sunrise and sunset are calculated in Standard Time.

$$sunrise = 12 - \frac{\cos^{-1}(-\tan \phi \tan \delta)}{15^\circ} - \frac{TC}{60} \qquad sunset = 12 + \frac{\cos^{-1}(-\tan \phi \tan \delta)}{15^\circ} - \frac{TC}{60}$$

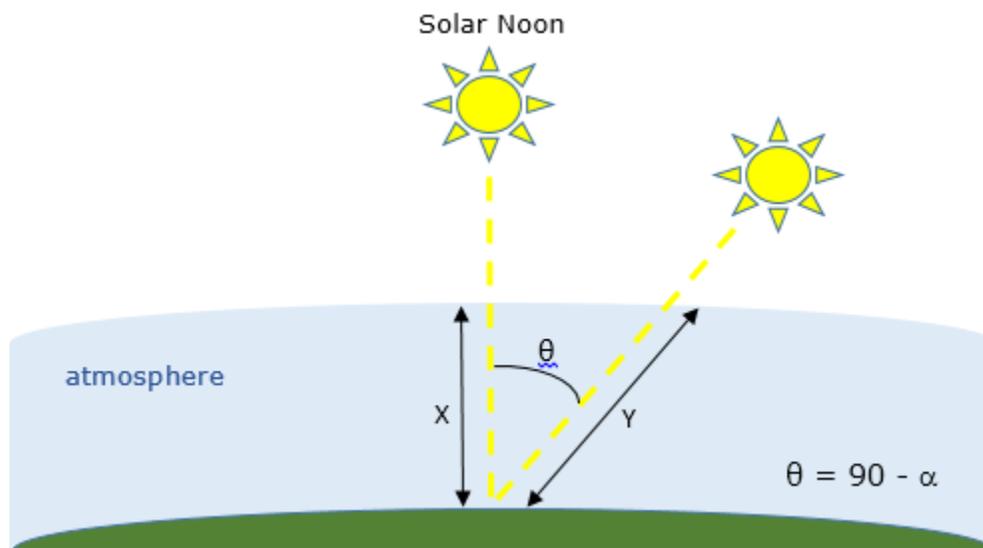
```
midnight = dateshift(localTime, 'start', 'day');
sr = 12 - acosd(-tand(phi)*tand(delta))/15 - solarCorr/60;
sunrise = timeofday(midnight + hours(sr));
ss = 12 + acosd(-tand(phi)*tand(delta))/15 - solarCorr/60;
sunset = timeofday(midnight + hours(ss));
disp(['Sunrise = ' datestr(sunrise, 'HH:MM:SS') '   Sunset = ' datestr(sunset, 'HH:MM:ss')])

Sunrise = 04:15:34   Sunset = 19:08:11
```

Air Mass and Solar Radiation

Include images to illustrate important points in your story. To include an image from a file, copy and paste an image from another source or go to the **Live Editor** tab and click the **Image** button.

As light from the sun passes through the earth's atmosphere, some of the solar radiation will be absorbed. The air mass is a function of solar elevation (α). As shown in the diagram below, it is a measure of the length of the path of light through the atmosphere (Y) relative to the shortest possible path (X) when the sun's elevation is 90° .



The larger the air mass, the less radiation reaches the ground. The air mass can be calculated from the equation

$$AM = \frac{1}{\cos(90 - \alpha) + 0.5057(6.0799 + \alpha)^{-1.6364}}$$

Then the solar radiation (in Kw/m²) reaching the ground can be calculated from the empirical equation

$$sRad = 1.353 * 0.7^{AM^{0.678}}$$

```
AM = 1/(cosd(90-alpha) + 0.50572*(6.07955+alpha)^-1.6354);
sRad = 1.353*0.7^(AM^0.678); % kW/m^2
disp(['Air Mass = ' num2str(AM) ' Solar Radiation = ' num2str(sRad) ' kW/m^2'])

Air Mass = 1.0688 Solar Radiation = 0.93164 kW/m^2
```

Solar Radiation on Fixed Panels

Use hyperlinks to reference supporting information from other sources. To add a hyperlink, go to the **Live Editor** tab and click the **Hyperlink** button.

Panels installed with a solar tracker can move with the sun and receive 100% of the sun's radiation as the sun moves across the sky. However, most solar cell installations have panels set at a fixed azimuth and tilt. Therefore the actual radiation reaching the panel will also depend on the sun's azimuth. The solar azimuth (γ) is the compass direction of the sun's position in the sky. At solar noon in the Northern hemisphere the sun's azimuth will be 180° (south). The solar azimuth is calculated from this equation:

$$\gamma = \begin{cases} \cos^{-1} \left(\frac{\sin \delta \cos \phi - \cos \delta \sin \phi \cos \omega}{\cos \alpha} \right) & \text{for solar time} \leq 12 \\ 360^\circ - \cos^{-1} \left(\frac{\sin \delta \cos \phi - \cos \delta \sin \phi \cos \omega}{\cos \alpha} \right) & \text{for solar time} > 12 \end{cases}$$

```
gamma = acosd((sind(delta)*cosd(phi) - cosd(delta)*sind(phi)*cosd(omega))/cosd(alpha))
if (hour(solarTime) >= 12) && (omega >= 0)
    gamma = 360 - gamma;
end
disp(['Solar Azimuth = ' num2str(gamma)])

Solar Azimuth = 191.8568
```

In the northern hemisphere, a typical solar panel installation would have panels oriented toward the south with a panel azimuth (β) of 180° . At northern latitudes a typical tilt angle (τ) would be 35° . For fixed panels, the panel radiation is calculated from the total solar radiation using this equation:

$$pRad = sRad[\cos(\alpha) \sin(\tau) \cos(\beta - \gamma) + \sin(\alpha) \cos(\tau)]$$

```
beta = 180; % Panel azimuth
tau = 35; % Panel tilt
```

```
pRad = sRad*max(0,(cosd(alpha)*sind(tau)*cosd(beta-gamma) + sind(alpha)*cosd(tau)));  
disp(['Panel Radiation = ' num2str(pRad) ' kW/m^2'])
```

```
Panel Radiation = 0.8989 kW/m^2
```

Panel Size and Efficiency

Let the reader change the values of parameters in the analysis to see how different parameter values affect the results.

So far our calculations reflect the theoretical maximum solar radiation available for power conversion. However, solar panels do not convert 100% of available solar radiation into electricity. The efficiency of a solar panel is the fraction of the radiation reaching the panel that is converted into electricity. The efficiency of a solar panel depends on the design and materials of the cell.

We will use a typical 20 m^2 residential installation with an efficiency of 25% to calculate the expected power output. You can modify the parameters below to see how efficiency and size affect panel power generation.

```
eff = 0.25; % Panel efficiency  
pSize = 20; % Panel size in m^2  
pElec = eff*pSize*pRad; % Panel electric output in  
disp(['Expected electrical output = ' num2str(pElec) ' kW'])
```

```
Expected electrical output = 4.4945 kW
```

Power Generation Over Time

Create plots in the Live Editor together with the code that produced them. Double-click a plot to open it in a separate figure window.

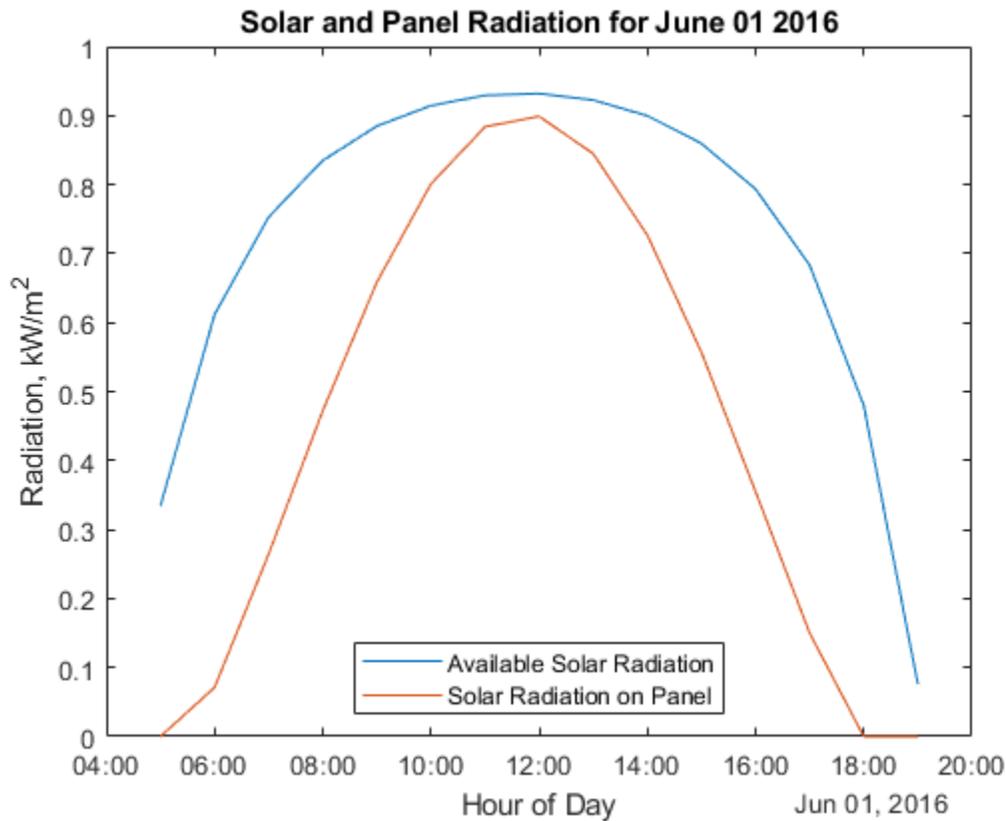
We can now calculate how much power a panel will generate on any day of the year. To simplify the analysis, we'll use a function called `hourlyPanelRadiation` that uses the equations above to calculate hourly total solar radiation and panel radiation for a given date and location. The parameter `isFixed` is set to 1 for fixed panels, 0 for tracking panels.

```
[time, sRad, pRad] = hourlyPanelRadiation(date, longitude, latitude,  
UTCoff, panelTilt, panelAzimuth, isFixed)
```

```
isFixed = 1;  
date = datetime(2016,6,1,'TimeZone',TZ); % Try other dates
```

```
[times, sRad, pRad] = hourlyPanelRadiation(date, lambda, phi, UTCoff, tau, beta, isFixe

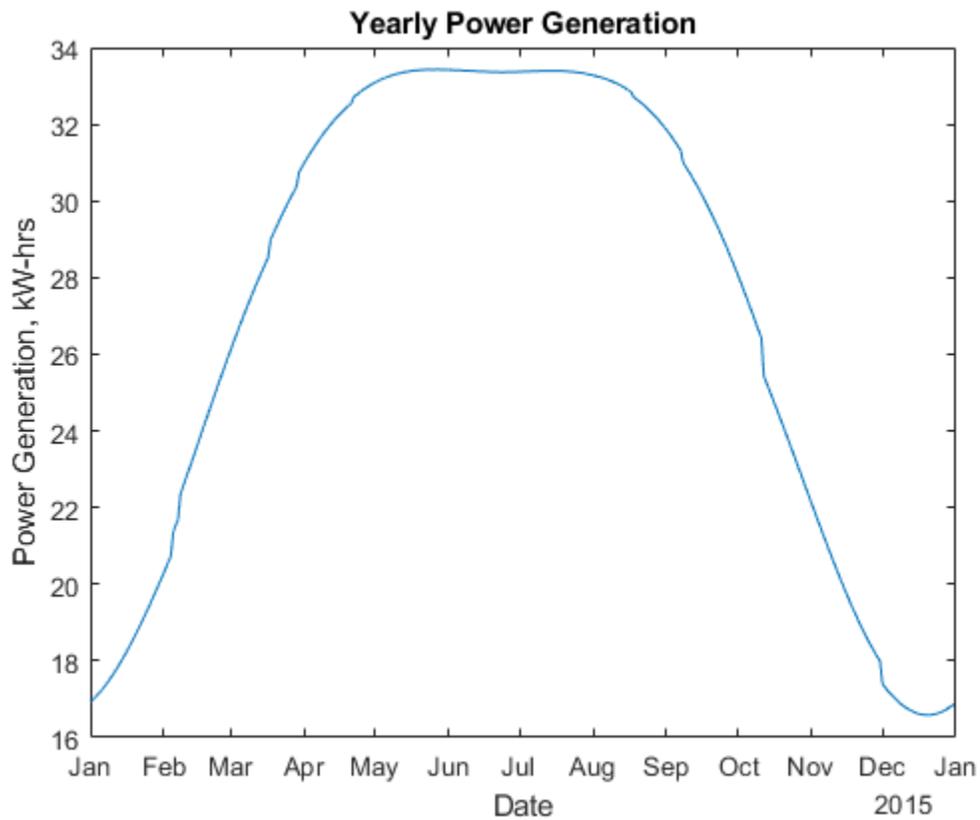
plot(times,sRad,times,pRad)
title(['Solar and Panel Radiation for ' datestr(date,'mmm dd yyyy')])
xlabel('Hour of Day');
ylabel('Radiation, kW/m^2')
legend('Available Solar Radiation','Solar Radiation on Panel', 'Location','South')
```



We can repeat the calculation to estimate power generation for each day of the year.

```
date = datetime(2015,1,1:365,'TimeZone',TZ); % Create a vector of days in the year
for i = 1:365
    [times, sRad, pRad] = hourlyPanelRadiation(date(i), lambda, phi, UTCoff, tau, beta, isFixe
    radiation = sum(pRad(1:end-1)+pRad(2:end))/2;
```

```
    dailyPower(i) = eff*pSize*radiation;  
end  
  
plot(date,dailyPower)  
title('Yearly Power Generation')  
xlabel('Date');  
ylabel('Power Generation, kW-hrs')
```



```
yearlyPower = sum(dailyPower);  
disp(['Expected annual power output = ' num2str(yearlyPower) ' kW-hrs'])  
  
Expected annual power output = 9954.3272 kW-hrs
```

Extending the Analysis

Share your analysis with colleagues. Invite them to reproduce or extend your analysis. Work collaboratively using the Live Editor.

In reality, true power output from a solar installation will be significantly affected by local weather conditions. An interesting extension of this analysis would be to see how cloud cover affects the results. In the US, we can use data from the following government websites.

- Use historical local weather data from the National Weather Service website.
- Use measured solar radiation data from the National Solar Radiation Database.

Teach with Live Scripts

The following is an example of how to use live scripts in the classroom. This example shows how to:

- Add equations to explain the underlying mathematics.
- Execute individual sections of MATLAB code.
- Include plots for visualization.
- Use links and images to provide supporting information.
- Experiment with MATLAB code interactively.
- Reinforce concepts with other examples.
- Use live scripts for assignments.

What does it mean to find the n th root of 1?

Add equations to explain the underlying mathematics for concepts that you want to teach. To add an equation, go to the **Live Editor** tab and click the **Equation** button. Then, select from the symbols and structures in the **Equation** tab.

Today we're going to talk about finding the roots of 1. What does it mean to find the n th root of 1? The n th roots of 1 are the solutions to the equation $x^n - 1 = 0$.

For square roots, this is easy. The values are $x = \pm\sqrt{1} = \pm 1$. For higher-order roots, it gets a bit more difficult. To find the cube roots of 1 we need to solve the equation $x^3 - 1 = 0$. We can factor this equation to get

$$(x - 1)(x^2 + x + 1) = 0.$$

So the first cube root is 1. Now we can use the quadratic formula to get the second and third cube roots.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Calculate the Cube Roots

To execute individual sections of MATLAB code, go to the **Live Editor** tab and click the **Run Section** button. Output appears together with the code that created it. Create sections using the **Section Break** button.

In our case a , b , and c are all equal to 1. The other two roots are calculated from these formulas:

```
a = 1 ; b = 1 ; c = 1;
roots = [];
roots(1) = 1;
roots(2) = (-b + sqrt(b^2 - 4*a*c))/(2*a);    % Use the quadratic formula
roots(3) = (-b - sqrt(b^2 - 4*a*c))/(2*a);
```

So the full set of cube roots of 1 are:

```
disp(roots')

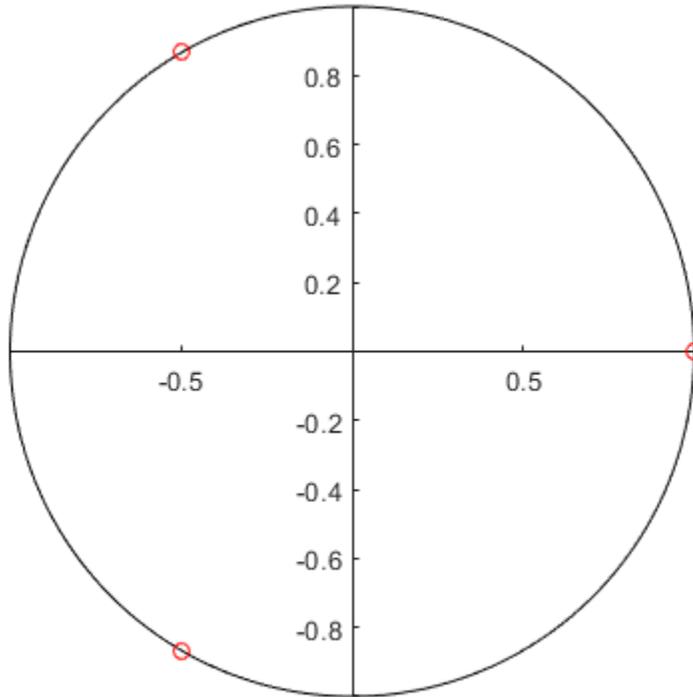
    1.0000 + 0.0000i
   -0.5000 - 0.8660i
   -0.5000 + 0.8660i
```

Displaying Roots in the Complex Plane

Include plots in the Live Editor so students can visualize important concepts.

We can visualize the roots in the complex plane to see their location.

```
range = 0:0.01:2*pi;
plot(cos(range),sin(range),'k')           % Plot the unit circle
axis square; box off
ax = gca;
ax.XAxisLocation = 'origin';
ax.YAxisLocation = 'origin';
hold on
plot(real(roots), imag(roots), 'ro')     % Plot the roots
```



Finding Higher Order Roots

To add supporting information, go to the **Live Editor** tab and click the **Hyperlink** and **Image** buttons. Students can use supporting information to explore lecture topics outside of the classroom.

Once you get past $n = 3$, things get even trickier. For 4th roots we could use the quartic formula discovered by Lodovico Ferrari in 1540. But this formula is long and unwieldy, and doesn't help us find roots higher than 4. Luckily, there is a better way, thanks to a 17th century French mathematician named Abraham de Moivre.

Abraham de Moivre was born in Vitry in Champagne on May 26, 1667. He was a contemporary and friend of Isaac Newton, Edmund Halley, and James Stirling. https://en.wikipedia.org/wiki/Abraham_de_Moivre



Abraham de Moivre

He is best known for de Moivre's theorem that links complex numbers and trigonometry, and for his work on the normal distribution and probability theory. De Moivre wrote a book on probability theory, *The Doctrine of Chances*, said to have been prized by gamblers. De Moivre first discovered Binet's formula, the closed-form expression for Fibonacci numbers linking the n th power of the golden ratio φ to the n th Fibonacci number. He was also the first to postulate the Central Limit Theorem, a cornerstone of probability theory.

de Moivre's theorem states that for any real x and any integer n ,

$$(\cos x + i \sin x)^n = \cos(nx) + i \sin(nx).$$

How does that help us solve our problem? We also know that for any integer k ,

$$1 = \cos(2k\pi) + i \sin(2k\pi).$$

So by de Moivre's theorem we get

$$1^{1/n} = (\cos(2k\pi) + i \sin(2k\pi))^{1/n} = \cos\left(\frac{2k\pi}{n}\right) + i \sin\left(\frac{2k\pi}{n}\right).$$

Calculating the n th Roots of 1

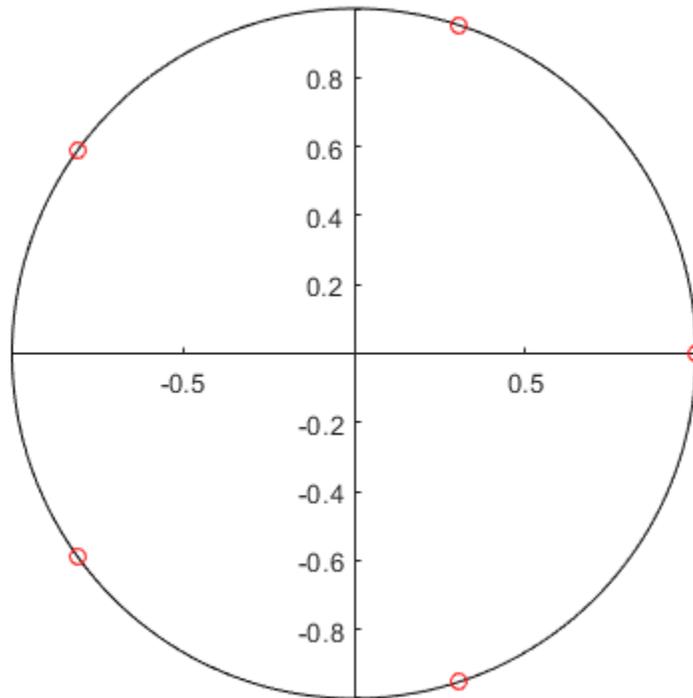
Use the Live Editor to experiment with MATLAB code interactively. Show students how important parameters affect the analysis.

We can use this last equation to find the n th roots of 1. For example, for any value of n , we can use the formula above with values of $k = 0 \dots n - 1$. We can use this MATLAB code to experiment with different values of n :

```
roots = [];  
n = 5;  
for k = 0:n-1  
    roots(k+1) = cos(2*k*pi/n) + 1i*sin(2*k*pi/n);    % Calculate the roots  
end  
disp(roots')  
  
    1.0000 + 0.0000i  
    0.3090 - 0.9511i  
   -0.8090 - 0.5878i  
   -0.8090 + 0.5878i  
    0.3090 + 0.9511i
```

Plotting the roots in the complex plane shows that the roots are equally spaced around the unit circle at intervals of $2\pi/n$.

```
cla  
plot(cos(range),sin(range),'k')                % Plot the unit circle  
hold on  
plot(real(roots),imag(roots),'ro')            % Plot the roots
```



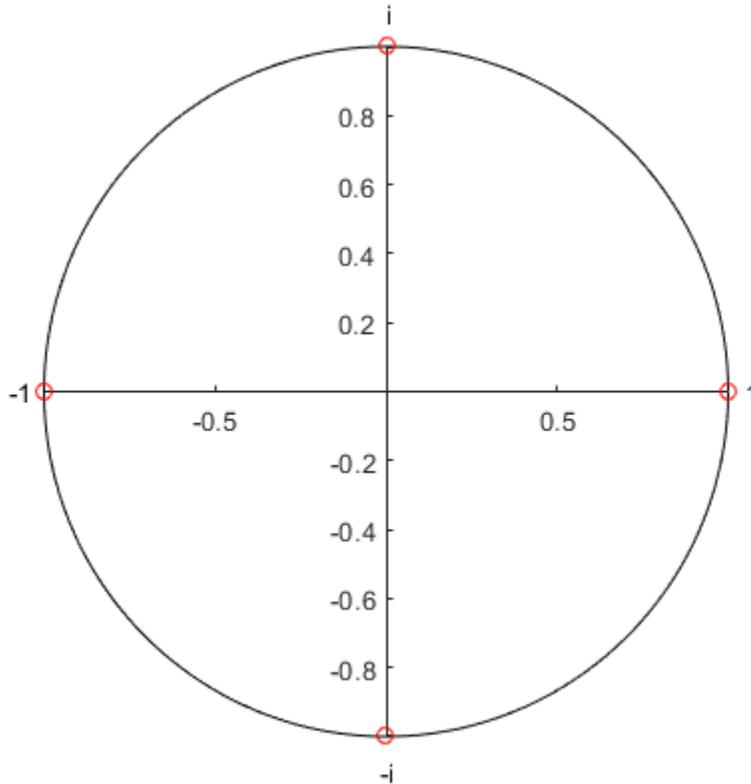
Finding the n th roots of -1 , i , and $-i$

Use additional examples to reinforce important concepts. Modify code during the lecture to answer questions or explore ideas in more depth.

We can find the roots of -1 , i , and $-i$ just by using extensions of the approach described above. If we look at the unit circle we see that the values of 1 , i , -1 , $-i$ appear at angles 0 , $\pi/2$, π , and $3\pi/2$ respectively.

```
r = ones(1,4);
theta = [0 pi/2 pi 3*pi/2];
[x,y] = pol2cart(theta,r);
cla
```

```
plot(cos(range),sin(range),'k')           % Plot the unit circle
hold on
plot(x, y, 'ro')                         % Plot the values of 1, i, -1, and -i
text(x(1)+0.05,y(1),'1')                 % Add text labels
text(x(2),y(2)+0.1,'i')
text(x(3)-0.1,y(3),'-1')
text(x(4)-0.02,y(4)-0.1,'-i')
```



Knowing this, we can write the following expression for i :

$$i = \cos((2k + 1/2)\pi) + i \sin((2k + 1/2)\pi).$$

Taking the n th root of both sides gives

$$i^{1/n} = (\cos((2k + 1/2)\pi) + i \sin((2k + 1/2)\pi))^{1/n}$$

and by de Moivre's theorem we get

$$i^{1/n} = (\cos((2k + 1/2)\pi) + i \sin((2k + 1/2)\pi))^{1/n} = \cos\left(\frac{(2k + 1/2)\pi}{n}\right) + i \sin\left(\frac{(2k + 1/2)\pi}{n}\right).$$

Homework

Use live scripts as the basis for assignments. Give students the live script used in the lecture and have them complete exercises that test their understanding of the material.

Use the techniques described above to complete the following exercises:

Exercise 1: Write MATLAB code to calculate the 3 cube roots of i .

`% Put your code here`

Exercise 2: Write MATLAB code to calculate the 5 fifth roots of -1 .

`% Put your code here`

Exercise 3: Describe the mathematical approach you would use to calculate the n th roots of an arbitrary complex number. Include the equations you used in your approach.

(Describe your approach here)

Function Basics

- “Create Functions in Files” on page 20-2
- “Add Help for Your Program” on page 20-6
- “Run Functions in the Editor” on page 20-8
- “Base and Function Workspaces” on page 20-10
- “Share Data Between Workspaces” on page 20-11
- “Check Variable Scope in Editor” on page 20-16
- “Types of Functions” on page 20-20
- “Anonymous Functions” on page 20-24
- “Local Functions” on page 20-30
- “Nested Functions” on page 20-32
- “Variables in Nested and Anonymous Functions” on page 20-39
- “Private Functions” on page 20-41
- “Function Precedence Order” on page 20-43

Create Functions in Files

Both scripts and functions allow you to reuse sequences of commands by storing them in program files. Scripts are the simplest type of program, since they store commands exactly as you would type them at the command line. Functions provide more flexibility, primarily because you can pass input values and return output values. For example, this function named `fact` computes the factorial of a number (n) and returns the result (f).

```
function f = fact(n)
    f = prod(1:n);
end
```

This type of function must be defined within a file, not at the command line. Often, you store a function in its own file. In that case, the best practice is to use the same name for the function and the file (in this example, `fact.m`), since MATLAB associates the program with the file name. Save the file either in the current folder or in a folder on the MATLAB search path.

You can call the function from the command line, using the same syntax rules that apply to functions installed with MATLAB. For instances, calculate the factorial of 5.

```
x = 5;
y = fact(5)

y =

    120
```

Starting in R2016b, another option for storing functions is to include them at the end of a script file. For instance, create a file named `mystats.m` with a few commands and two functions, `fact` and `perm`. The script calculates the permutation of (3,2).

```
x = 3;
y = 2;
z = perm(x,y)

function p = perm(n,r)
    p = fact(n)*fact(n-r);
end

function f = fact(n)
    f = prod(1:n);
end
```

Call the script from the command line.

```
mystats
```

```
z =
```

```
6
```

Syntax for Function Definition

The first line of every function is the definition statement, which includes the following elements.

function keyword (required)	Use lowercase characters for the keyword.
Output arguments (optional)	<p>If your function returns one output, you can specify the output name after the <code>function</code> keyword.</p> <pre>function myOutput = myFunction(x)</pre> <p>If your function returns more than one output, enclose the output names in square brackets.</p> <pre>function [one,two,three] = myFunction(x)</pre> <p>If there is no output, you can omit it.</p> <pre>function myFunction(x)</pre> <p>Or you can use empty square brackets.</p> <pre>function [] = myFunction(x)</pre>

Function name (required)	<p>Valid function names follow the same rules as variable names. They must start with a letter, and can contain letters, digits, or underscores.</p> <hr/> <p>Note To avoid confusion, use the same name for both the function file and the first function within the file. MATLAB associates your program with the <i>file</i> name, not the function name. Script files cannot have the same name as a function in the file.</p>
Input arguments (optional)	<p>If your function accepts any inputs, enclose their names in parentheses after the function name. Separate inputs with commas.</p> <pre>function y = myFunction(one, two, three)</pre> <p>If there are no inputs, you can omit the parentheses.</p>

Tip When you define a function with multiple input or output arguments, list any required arguments first. This ordering allows you to call your function without specifying optional arguments.

Contents of Functions and Files

The body of a function can include valid MATLAB expressions, control flow statements, comments, blank lines, and nested functions. Any variables that you create within a function are stored within a workspace specific to that function, which is separate from the base workspace.

Program files can contain multiple functions. If the file contains only function definitions, the first function is the main function, and is the function that MATLAB associates with the file name. Functions that follow the main function or script code are called local functions. Local functions are only available within the file.

End Statements

Functions end with either an `end` statement, the end of the file, or the definition line for a local function, whichever comes first. The end statement is required if:

- Any function in the file contains a nested function (a function completely contained within its parent).
- The function is a local function within a function file, and any local function in the file uses the `end` keyword.
- The function is a local function within a script file.

Although it is sometimes optional, use `end` for better code readability.

See Also

`function`

More About

- “Files and Folders that MATLAB Accesses”
- “Base and Function Workspaces” on page 20-10
- “Types of Functions” on page 20-20
- “Add Functions to Scripts” on page 18-18

Add Help for Your Program

This example shows how to provide help for the programs you write. Help text appears in the Command Window when you use the `help` function.

Create help text by inserting comments at the beginning of your program. If your program includes a function, position the help text immediately below the function definition line (the line with the `function` keyword).

For example, create a function in a file named `addme.m` that includes help text:

```
function c = addme(a,b)
% ADDME Add two values together.
% C = ADDME(A) adds A to itself.
%
% C = ADDME(A,B) adds A and B together.
%
% See also SUM, PLUS.

switch nargin
    case 2
        c = a + b;
    case 1
        c = a + a;
    otherwise
        c = 0;
end
```

When you type `help addme` at the command line, the help text displays in the Command Window:

```
addme Add two values together.
C = addme(A) adds A to itself.

C = addme(A,B) adds A and B together.

See also sum, plus.
```

The first help text line, often called the H1 line, typically includes the program name and a brief description. The Current Folder browser and the `help` and `lookfor` functions use the H1 line to display information about the program.

Create `See also` links by including function names at the end of your help text on a line that begins with `% See also`. 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, `help` prints the function names as they appear in the help text.

You can include hyperlinks (in the form of URLs) to Web sites in your help text. Create hyperlinks by including an HTML `<a>` anchor element. Within the anchor, use a `matlab:` statement to execute a `web` command. For example:

```
% For more information, see <a href="matlab:  
% web('https://www.mathworks.com')">the MathWorks Web site</a>.
```

End your help text with a blank line (without a `%`). The help system ignores any comment lines that appear after the help text block.

Note When multiple programs have the same name, the `help` command determines which help text to display by applying the rules described in “Function Precedence Order” on page 20-43. However, if a program has the same name as a MathWorks function, the **Help on Selection** option in context menus always displays documentation for the MathWorks function.

See Also

`help` | `lookfor`

Related Examples

- “Add Comments to Programs” on page 18-4
- “Create Help Summary Files — Contents.m” on page 30-12
- “Check Which Programs Have Help” on page 30-9
- “Display Custom Documentation” on page 30-27
- “Use Help Files with MEX Functions”

Run Functions in the Editor

This example shows how to run a function that requires some initial setup, such as input argument values, while working in the Editor.

- 1 Create a function in a program file named `myfunction.m`.

```
function y = myfunction(x)
y = x.^2 + x;
```

This function requires input `x`.

- 2 View the commands available for running the function by clicking **Run** on the **Editor** tab. The command at the top of the list is the command that the Editor uses by default when you click the **Run** icon.



- 3 Replace the text *type code to run* with an expression that allows you to run the function.

```
y = myfunction(1:10)
```

You can enter multiple commands on the same line, such as

```
x = 1:10; y = myfunction(x)
```

For more complicated, multiline commands, create a separate script file, and then run the script.

Note Run commands use the base workspace. Any variables that you define in a run command can overwrite variables in the base workspace that have the same name.

- 4 Run the function by clicking **Run** or a specific run command from the drop-down list. For `myfunction.m`, and an input of `1:10`, this result appears in the Command Window:

y = 2 6 12 20 30 42 56 72 90 110

When you select a run command from the list, it becomes the default for the **Run** button.

To edit or delete an existing run command, select the command, right-click, and then select **Edit** or **Delete**.

Base and Function Workspaces

This topic explains the differences between the base workspace and function workspaces, including workspaces for local functions, 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.

See Also

Related Examples

- “Share Data Between Workspaces” on page 20-11

More About

- “Nested Functions” on page 20-32

Share Data Between Workspaces

In this section...

“Introduction” on page 20-11
“Best Practice: Passing Arguments” on page 20-11
“Nested Functions” on page 20-12
“Persistent Variables” on page 20-12
“Global Variables” on page 20-13
“Evaluating in Another Workspace” on page 20-14

Introduction

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.

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 “Nested Functions” on page 20-32.

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 character vectors 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');
src.String = vars;
```

For other programming applications, consider argument passing and the techniques described in “Alternatives to the eval Function” on page 2-83.

See Also

More About

- “Base and Function Workspaces” on page 20-10

Check Variable Scope in Editor

In this section...

“Use Automatic Function and Variable Highlighting” on page 20-16

“Example of Using Automatic Function and Variable Highlighting” on page 20-17

Scoping issues can be the source of some coding problems. For instance, if you are unaware that nested functions share a particular variable, the results of running your code might not be as you expect. Similarly, mistakes in usage of local, global, and persistent variables can cause unexpected results.

The Code Analyzer does not always indicate scoping issues because sharing a variable across functions is not an error—it may be your intent. Use MATLAB function and variable highlighting features to identify when and where your code uses functions and variables. If you have an active Internet connection, you can watch the Variable and Function Highlighting video for an overview of the major features.

For conceptual information on nested functions and the various types of MATLAB variables, see “Sharing Variables Between Parent and Nested Functions” on page 20-33 and “Share Data Between Workspaces” on page 20-11.

Use Automatic Function and Variable Highlighting

By default, the Editor indicates functions, local variables, and variables with shared scope in various shades of blue. Variables with shared scope include: global variables on page 20-13, persistent variables on page 20-12, and variables within nested functions. (For more information, see “Nested Functions” on page 20-12.)

To enable and disable highlighting or to change the colors, click  **Preferences** and select **MATLAB > Colors > Programming tools**. In MATLAB Online, highlighting is enabled by default and changing the preferences for highlighting is not available.

By default, the Editor:

- Highlights all instances of a given function or local variable in sky blue when you place the cursor within a function or variable name. For instance:

```
collatz
```

- Displays a variable with shared scope in teal blue, regardless of the cursor location.
For instance:

x

Example of Using Automatic Function and Variable Highlighting

Consider the code for a function rowsum:

```
function rowTotals = rowsum
% Add the values in each row and
% store them in a new array

x = ones(2,10);
[n, m] = size(x);
rowTotals = zeros(1,n);
for i = 1:n
    rowTotals(i) = addToSum;
end

function colsum = addToSum
    colsum = 0;
    thisrow = x(i,:);
    for i = 1:m
        colsum = colsum + thisrow(i);
    end
end

end
```

When you run this code, instead of returning the sum of the values in each row and displaying:

```
ans =
    10    10
```

MATLAB displays:

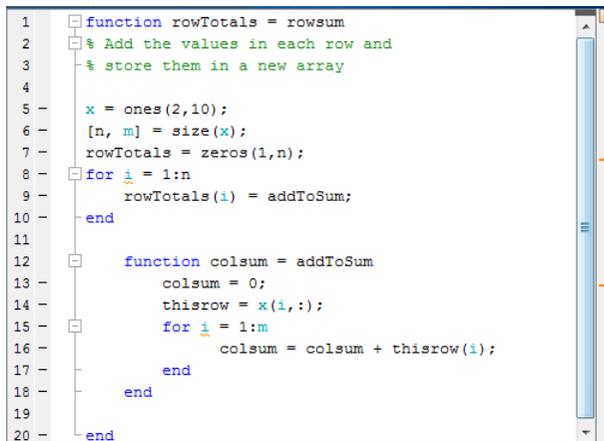
```
ans =
    0     0     0     0     0     0     0     0     0     10
```

Examine the code by following these steps:

- 1 On the **Home** tab in the **Environment** section, click  **Preferences** and select **MATLAB > Colors > Programming tools**. Ensure that **Automatically highlight** and **Variables with shared scope** are selected.
- 2 Copy the rowsum code into the Editor.

Notice the variable `i` appears in teal blue, which indicates `i` is not a local variable. Both the `rowTotals` function and the `addToSum` functions set and use the variable `i`.

The variable `n`, at line 6 appears in black, indicating that it does not span multiple functions.



```

1  function rowTotals = rowsum
2  % Add the values in each row and
3  % store them in a new array
4
5  x = ones(2,10);
6  [n, m] = size(x);
7  rowTotals = zeros(1,n);
8  for i = 1:n
9      rowTotals(i) = addToSum;
10 end
11
12 function colsum = addToSum
13     colsum = 0;
14     thisrow = x(i,:);
15     for i = 1:m
16         colsum = colsum + thisrow(i);
17     end
18 end
19
20 end

```

- 3 Hover the mouse pointer over an instance of variable `i`.
A tooltip appears: The scope of variable 'i' spans multiple functions.
- 4 Click the tooltip link for information about variables whose scope span multiple functions.
- 5 Click an instance of `i`.

Every reference to `i` highlights in sky blue and markers appear in the indicator bar on the right side of the Editor.

```
1  function rowTotals = rowsum
2  % Add the values in each row and
3  % store them in a new array
4
5  x = ones(2,10);
6  [n, m] = size(x);
7  rowTotals = zeros(1,n);
8  for i = 1:n
9      rowTotals(i) = addToSum;
10 end
11
12 function colsum = addToSum
13     colsum = 0;
14     thisrow = x(i,:);
15     for k = 1:m
16         colsum = colsum + thisrow(i);
17     end
18 end
19
20 end
```

- 6 Hover over one of the indicator bar markers.

A tooltip appears and displays the name of the function or variable and the line of code represented by the marker.

- 7 Click a marker to navigate to the line indicated in tooltip for that marker.

This is particularly useful when your file contains more code than you can view at one time in the Editor.

Fix the code by changing the instance of `i` at line 15 to `y`.

You can see similar highlighting effects when you click on a function reference. For instance, click on `addToSum`.

Types of Functions

In this section...
“Local and Nested Functions in a File” on page 20-20
“Private Functions in a Subfolder” on page 20-21
“Anonymous Functions Without a File” on page 20-21

Local and Nested Functions in a File

Program files can contain multiple functions. Local and nested functions are useful for dividing programs into smaller tasks, making it easier to read and maintain your code.

Local functions are subroutines that are available within the same file. Local functions are the most common way to break up programmatic tasks. In a function file, which contains only function definitions, local functions can appear in the file in any order after the main function in the file. In a script file, which contains commands and function definitions, local function must be at the end of the file. (Functions in scripts are supported in R2016b or later.)

For example, create a function file named `myfunction.m` that contains a main function, `myfunction`, and two local functions, `squareMe` and `doubleMe`:

```
function b = myfunction(a)
    b = squareMe(a)+doubleMe(a);
end
function y = squareMe(x)
    y = x.^2;
end
function y = doubleMe(x)
    y = x.*2;
end
```

You can call the main function from the command line or another program file, although the local functions are only available to `myfunction`:

```
myfunction(pi)
```

```
ans =
    16.1528
```

Nested functions are completely contained within another function. The primary difference between nested functions and local functions is that nested functions can use variables defined in parent functions without explicitly passing those variables as arguments.

Nested functions are useful when subroutines share data, such as applications that pass data between components. For example, create a function that allows you to set a value between 0 and 1 using either a slider or an editable text box. If you use nested functions for the callbacks, the slider and text box can share the value and each other's handles without explicitly passing them:

```
function myslider
value = 0;
f = figure;
s = uicontrol(f,'Style','slider','Callback',@slider);
e = uicontrol(f,'Style','edit','Callback',@edittext,...
             'Position',[100,20,100,20]);

    function slider(obj,~)
        value = obj.Value;
        e.String = num2str(value);
    end
    function edittext(obj,~)
        value = str2double(obj.String);
        s.Value = value;
    end
end
```

Private Functions in a Subfolder

Like local or nested functions, private functions are accessible only to functions in a specific location. However, private functions are not in the same file as the functions that can call them. Instead, they are in a subfolder named `private`. Private functions are available only to functions in the folder immediately above the `private` folder. Use private functions to separate code into different files, or to share code between multiple, related functions.

Anonymous Functions Without a File

Anonymous functions allow you to define a function without creating a program file, as long as the function consists of a single statement. A common application of anonymous

functions is to define a mathematical expression, and then evaluate that expression over a range of values using a MATLAB® *function function*, i.e., a function that accepts a function handle as an input.

For example, this statement creates a function handle named `s` for an anonymous function:

```
s = @(x) sin(1./x);
```

This function has a single input, `x`. The `@` operator creates the function handle.

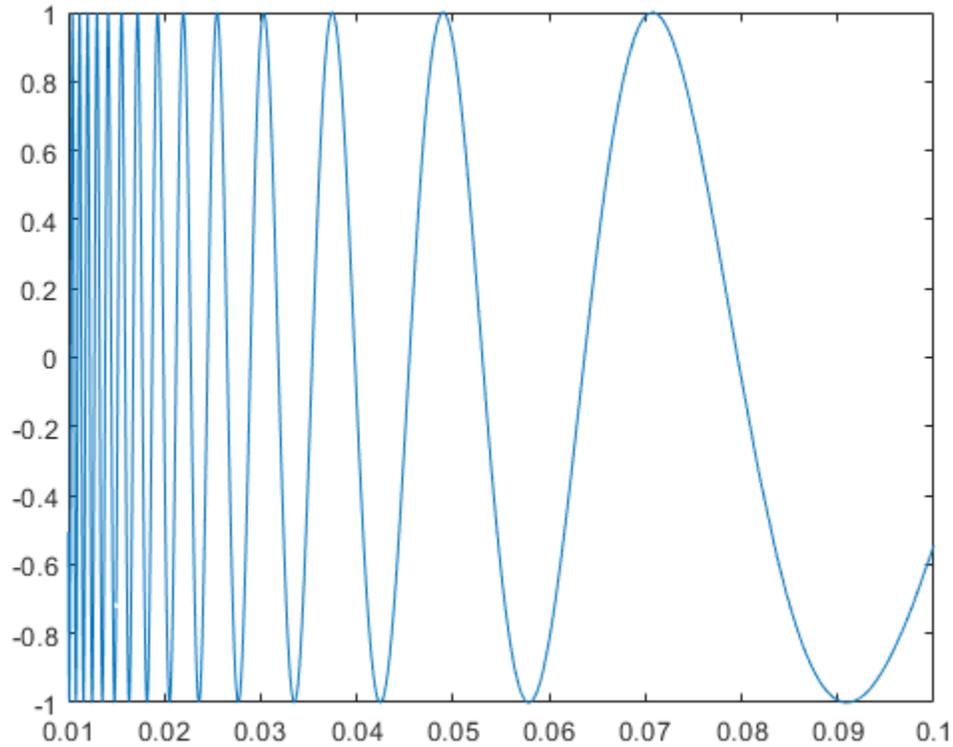
You can use the function handle to evaluate the function for particular values, such as

```
y = s(pi)
```

```
y = 0.3130
```

Or, you can pass the function handle to a function that evaluates over a range of values, such as `fplot`:

```
range = [0.01,0.1];  
fplot(s,range)
```



See Also

More About

- “Local Functions” on page 20-30
- “Nested Functions” on page 20-32
- “Private Functions” on page 20-41
- “Anonymous Functions” on page 20-24

Anonymous Functions

In this section...

“What Are Anonymous Functions?” on page 20-24

“Variables in the Expression” on page 20-25

“Multiple Anonymous Functions” on page 20-26

“Functions with No Inputs” on page 20-27

“Functions with Multiple Inputs or Outputs” on page 20-27

“Arrays of Anonymous Functions” on page 20-28

What Are Anonymous Functions?

An anonymous function is a function that is *not* stored in a program file, but is associated with a variable whose data type is `function_handle`. Anonymous functions can accept inputs and return outputs, just as standard functions do. However, they can contain only a single executable statement.

For example, create a handle to an anonymous function that finds the square of a number:

```
sqr = @(x) x.^2;
```

Variable `sqr` is a function handle. The `@` operator creates the handle, and the parentheses `()` immediately after the `@` operator include the function input arguments. This anonymous function accepts a single input `x`, and implicitly returns a single output, an array the same size as `x` that contains the squared values.

Find the square of a particular value (5) by passing the value to the function handle, just as you would pass an input argument to a standard function.

```
a = sqr(5)
```

```
a =  
    25
```

Many MATLAB functions accept function handles as inputs so that you can evaluate functions over a range of values. You can create handles either for anonymous functions or for functions in program files. The benefit of using anonymous functions is that you do not have to edit and maintain a file for a function that requires only a brief definition.

For example, find the integral of the `sqr` function from 0 to 1 by passing the function handle to the `integral` function:

```
q = integral(sqr,0,1);
```

You do not need to create a variable in the workspace to store an anonymous function. Instead, you can create a temporary function handle within an expression, such as this call to the `integral` function:

```
q = integral(@(x) x.^2,0,1);
```

Variables in the Expression

Function handles can store not only an expression, but also variables that the expression requires for evaluation.

For example, create a function handle to an anonymous function that requires coefficients `a`, `b`, and `c`.

```
a = 1.3;  
b = .2;  
c = 30;  
parabola = @(x) a*x.^2 + b*x + c;
```

Because `a`, `b`, and `c` are available at the time you create `parabola`, the function handle includes those values. The values persist within the function handle even if you clear the variables:

```
clear a b c  
x = 1;  
y = parabola(x)  
  
y =  
    31.5000
```

To supply different values for the coefficients, you must create a new function handle:

```
a = -3.9;  
b = 52;  
c = 0;  
parabola = @(x) a*x.^2 + b*x + c;  
  
x = 1;  
y = parabola(1)
```

```
y =  
    48.1000
```

You can save function handles and their associated values in a MAT-file and load them in a subsequent MATLAB session using the `save` and `load` functions, such as

```
save myfile.mat parabola
```

Use only explicit variables when constructing anonymous functions. If an anonymous function accesses any variable or nested function that is not explicitly referenced in the argument list or body, MATLAB throws an error when you invoke the function. Implicit variables and function calls are often encountered in the functions such as `eval`, `evalin`, `assignin`, and `load`. Avoid using these functions in the body of anonymous functions.

Multiple Anonymous Functions

The expression in an anonymous function can include another anonymous function. This is useful for passing different parameters to a function that you are evaluating over a range of values. For example, you can solve the equation

$$g(c) = \int_0^1 (x^2 + cx + 1) dx$$

for varying values of c by combining two anonymous functions:

```
g = @(c) (integral(@(x) (x.^2 + c*x + 1),0,1));
```

Here is how to derive this statement:

- 1 Write the integrand as an anonymous function,
`@(x) (x.^2 + c*x + 1)`
- 2 Evaluate the function from zero to one by passing the function handle to `integral`,
`integral(@(x) (x.^2 + c*x + 1),0,1)`
- 3 Supply the value for c by constructing an anonymous function for the entire equation,
`g = @(c) (integral(@(x) (x.^2 + c*x + 1),0,1));`

The final function allows you to solve the equation for any value of c . For example:

```
g(2)
```

```
ans =  
    2.3333
```

Functions with No Inputs

If your function does not require any inputs, use empty parentheses when you define and call the anonymous function. For example:

```
t = @() datestr(now);  
d = t()  
  
d =  
26-Jan-2012 15:11:47
```

Omitting the parentheses in the assignment statement creates another function handle, and does not execute the function:

```
d = t  
  
d =  
    @( ) datestr(now)
```

Functions with Multiple Inputs or Outputs

Anonymous functions require that you explicitly specify the input arguments as you would for a standard function, separating multiple inputs with commas. For example, this function accepts two inputs, x and y :

```
myfunction = @(x,y) (x^2 + y^2 + x*y);  
  
x = 1;  
y = 10;  
z = myfunction(x,y)  
  
z = 111
```

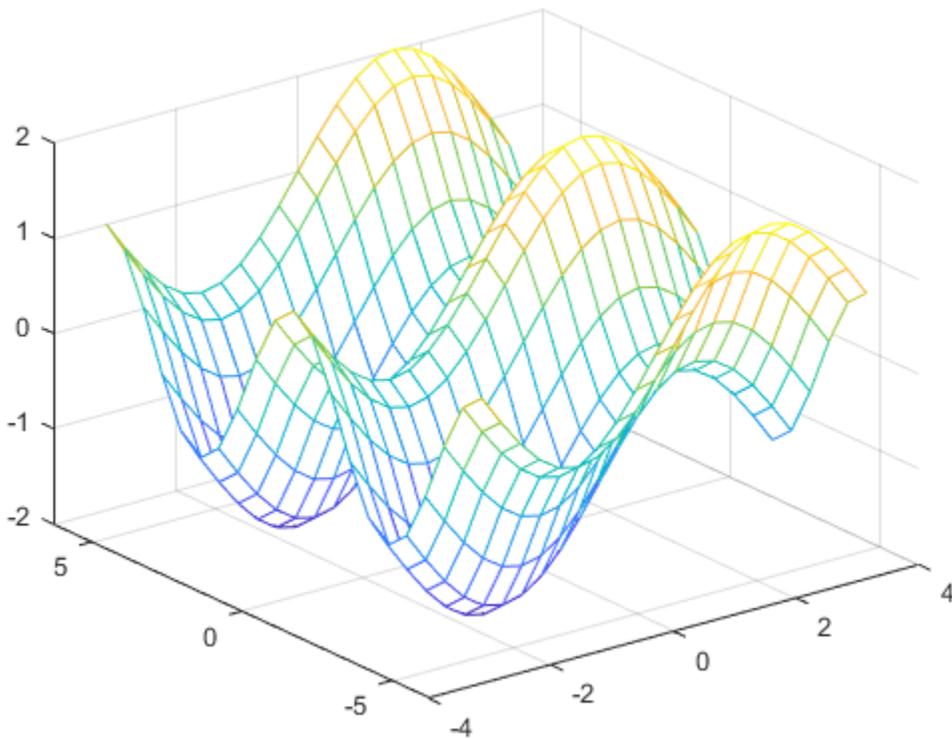
However, you do not explicitly define output arguments when you create an anonymous function. If the expression in the function returns multiple outputs, then you can request them when you call the function. Enclose multiple output variables in square brackets.

For example, the `ndgrid` function can return as many outputs as the number of input vectors. This anonymous function that calls `ndgrid` can also return multiple outputs:

```
c = 10;  
mygrid = @(x,y) ndgrid((-x:x/c:x),(-y:y/c:y));  
[x,y] = mygrid(pi,2*pi);
```

You can use the output from `mygrid` to create a mesh or surface plot:

```
z = sin(x) + cos(y);  
mesh(x,y,z)
```



Arrays of Anonymous Functions

Although most MATLAB fundamental data types support multidimensional arrays, function handles must be scalars (single elements). However, you can store multiple

function handles using a cell array or structure array. The most common approach is to use a cell array, such as

```
f = {@(x)x.^2;  
     @(y)y+10;  
     @(x,y)x.^2+y+10};
```

When you create the cell array, keep in mind that MATLAB interprets spaces as column separators. Either omit spaces from expressions, as shown in the previous code, or enclose expressions in parentheses, such as

```
f = {@(x) (x.^2);  
     @(y) (y + 10);  
     @(x,y) (x.^2 + y + 10)};
```

Access the contents of a cell using curly braces. For example, `f{1}` returns the first function handle. To execute the function, pass input values in parentheses after the curly braces:

```
x = 1;  
y = 10;  
  
f{1}(x)  
f{2}(y)  
f{3}(x,y)  
  
ans =  
     1  
  
ans =  
    20  
  
ans =  
    21
```

See Also

More About

- “Create Function Handle” on page 13-2

Local Functions

This topic explains the term local function, and shows how to create and use local functions.

MATLAB program files can contain code for more than one function. In a function file, the first function in the file is called the main function. This function is visible to functions in other files, or you can call it from the command line. Additional functions within the file are called local functions, and they can occur in any order after the main function. Local functions are only visible to other functions in the same file. They are equivalent to subroutines in other programming languages, and are sometimes called subfunctions.

As of R2016b, you can also create local functions in a script file, as long as they all appear after the last line of script code. For more information, see “Add Functions to Scripts” on page 18-18.

For example, create a function file named `mystats.m` that contains a main function, `mystats`, and two local functions, `mymean` and `mymedian`.

```
function [avg, med] = mystats(x)
n = length(x);
avg = mymean(x,n);
med = mymedian(x,n);
end

function a = mymean(v,n)
% MYMEAN Example of a local function.

a = sum(v)/n;
end

function m = mymedian(v,n)
% MYMEDIAN Another example of a local function.

w = sort(v);
if rem(n,2) == 1
    m = w((n + 1)/2);
else
    m = (w(n/2) + w(n/2 + 1))/2;
end
end
```

The local functions `mymean` and `mymedian` calculate the average and median of the input list. The main function `mystats` determines the length of the list `n` and passes it to the local functions.

Although you cannot call a local function from the command line or from functions in other files, you can access its help using the `help` function. Specify names of both the file and the local function, separating them with a `>` character:

```
help mystats>mymean
```

```
mymean Example of a local function.
```

Local functions in the current file have precedence over functions in other files. That is, when you call a function within a program file, MATLAB checks whether the function is a local function before looking for other main functions. Therefore, you can create an alternate version of a particular function while retaining the original in another file.

All functions, including local functions, have their own workspaces that are separate from the base workspace. Local functions cannot access variables used by other functions unless you pass them as arguments. In contrast, *nested* functions (functions completely contained within another function) can access variables used by the functions that contain them.

See Also

`localfunctions`

More About

- “Nested Functions” on page 20-32
- “Function Precedence Order” on page 20-43

Nested Functions

In this section...

“What Are Nested Functions?” on page 20-32

“Requirements for Nested Functions” on page 20-32

“Sharing Variables Between Parent and Nested Functions” on page 20-33

“Using Handles to Store Function Parameters” on page 20-34

“Visibility of Nested Functions” on page 20-37

What Are Nested Functions?

A nested function is a function that is completely contained within a parent function. Any function in a program file can include a nested function.

For example, this function named `parent` contains a nested function named `nestedfx`:

```
function parent
disp('This is the parent function')
nestedfx

    function nestedfx
        disp('This is the nested function')
    end
end
```

The primary difference between nested functions and other types of functions is that they can access and modify variables that are defined in their parent functions. As a result:

- Nested functions can use variables that are not explicitly passed as input arguments.
- In a parent function, you can create a handle to a nested function that contains the data necessary to run the nested function.

Requirements for Nested Functions

- Typically, functions do not require an end statement. However, to nest any function in a program file, *all* functions in that file must use an end statement.

- You cannot define a nested function inside any of the MATLAB program control statements, such as `if/elseif/else`, `switch/case`, `for`, `while`, or `try/catch`.
- You must call a nested function either directly by name (without using `feval`), or using a function handle that you created using the `@` operator (and not `str2func`).
- 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. (For more information, see “Variables in Nested and Anonymous Functions” on page 20-39.)

Sharing Variables Between Parent and Nested Functions

In general, variables in one function workspace are not available to other functions. However, nested functions can access and modify variables in the workspaces of the functions that contain them.

This means that both a nested function and a function that contains it can modify the same variable without passing that variable as an argument. For example, in each of these functions, `main1` and `main2`, both the main function and the nested function can access variable `x`:

```
function main1
x = 5;
nestfun1

    function nestfun1
        x = x + 1;
    end
end

function main2
    nestfun2

        function nestfun2
            x = 5;
        end

        x = x + 1;
    end
```

When parent functions do not use a given variable, the variable remains local to the nested function. For example, in this function named `main`, the two nested functions have their own versions of `x` that cannot interact with each other:

```
function main
    nestedfun1
    nestedfun2

    function nestedfun1
        x = 1;
    end
```

```
function nestedfun2
    x = 2;
end
end
```

Functions that return output arguments have variables for the outputs in their workspace. However, parent functions only have variables for the output of nested functions if they explicitly request them. For example, this function `parentfun` does *not* have variable `y` in its workspace:

```
function parentfun
    x = 5;
    nestfun;

    function y = nestfun
        y = x + 1;
    end
end
```

If you modify the code as follows, variable `z` is in the workspace of `parentfun`:

```
function parentfun
    x = 5;
    z = nestfun;

    function y = nestfun
        y = x + 1;
    end
end
```

Using Handles to Store Function Parameters

Nested functions can use variables from three sources:

- Input arguments
- Variables defined within the nested function
- Variables defined in a parent function, also called externally scoped variables

When you create a function handle for a nested function, that handle stores not only the name of the function, but also the values of externally scoped variables.

For example, create a function in a file named `makeParabola.m`. This function accepts several polynomial coefficients, and returns a handle to a nested function that calculates the value of that polynomial.

```
function p = makeParabola(a,b,c)
p = @parabola;

    function y = parabola(x)
    y = a*x.^2 + b*x + c;
    end

end
```

The `makeParabola` function returns a handle to the `parabola` function that includes values for coefficients `a`, `b`, and `c`.

At the command line, call the `makeParabola` function with coefficient values of 1.3, .2, and 30. Use the returned function handle `p` to evaluate the polynomial at a particular point:

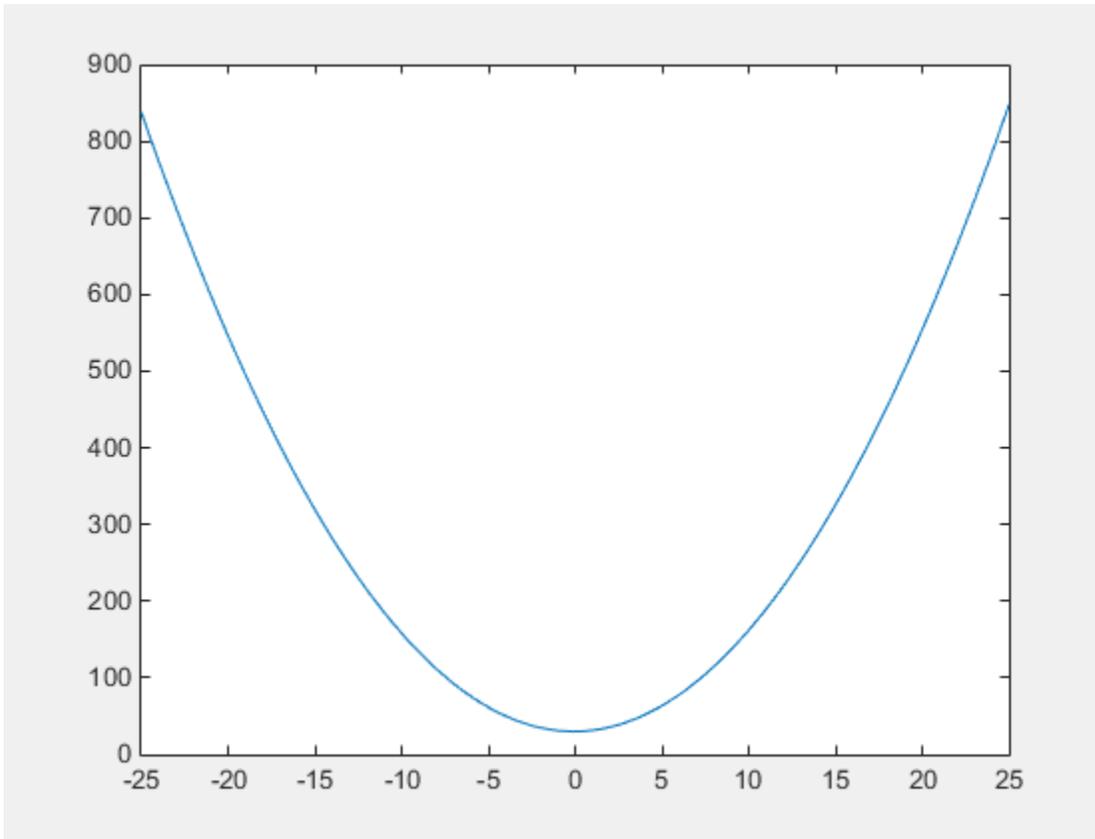
```
p = makeParabola(1.3, .2, 30);

X = 25;
Y = p(X)

Y =
    847.5000
```

Many MATLAB functions accept function handle inputs to evaluate functions over a range of values. For example, plot the parabolic equation from -25 to +25:

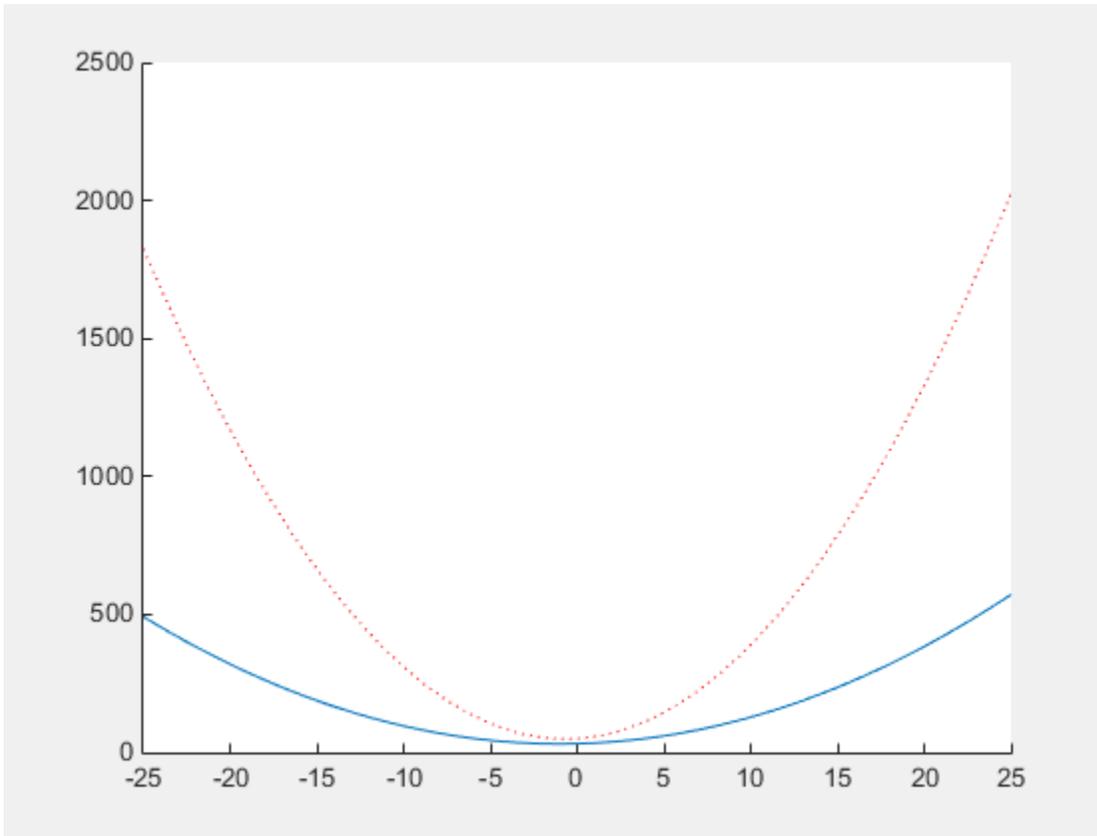
```
fplot(p, [-25,25])
```



You can create multiple handles to the `parabola` function that each use different polynomial coefficients:

```
firstp = makeParabola(0.8,1.6,32);  
secondp = makeParabola(3,4,50);  
range = [-25,25];
```

```
figure  
hold on  
fplot(firstp,range)  
fplot(secondp,range,'r:');  
hold off
```



Visibility of Nested Functions

Every function has a certain scope, that is, a set of other functions to which it is visible. A nested function is available:

- 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)           % Main 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
disp(x)
end
end
end
```

The easiest way to extend the scope of a nested function is to create a function handle and return it as an output argument, as shown in “Using Handles to Store Function Parameters” on page 20-34. Only functions that can call a nested function can create a handle to it.

See Also

More About

- “Variables in Nested and Anonymous Functions” on page 20-39
- “Create Function Handle” on page 13-2
- “Argument Checking in Nested Functions” on page 21-11

Variables in Nested and Anonymous Functions

The scoping rules for nested and anonymous functions require that all variables used within the function be present in the text of the code.

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, then MATLAB issues an error of the form

Attempt to add *variable* to a static workspace.

This table describes typical operations that attempt dynamic assignment, and the recommended ways to avoid it.

Type of Operation	Best Practice to Avoid Dynamic Assignment
load	Specify the variable name as an input to the load function. Or, assign the output from the load function to a structure array.
eval, evalin, or assignin	If possible, avoid using these functions altogether. See "Alternatives to the eval Function" on page 2-83.
Calling a MATLAB script that creates a variable	Convert the script to a function and pass the variable using arguments. This approach also clarifies the code.
Assigning to a variable in the MATLAB debugger	Assign the variable into the base workspace, such as K>> assignin('base','X',myvalue)

Another way to avoid dynamic assignment is to explicitly declare the variable within the function. For example, suppose a script named `makeX.m` assigns a value to variable `X`. A function that calls `makeX` and explicitly declares `X` avoids the dynamic assignment error because `X` is in the function workspace. A common way to declare a variable is to initialize its value to an empty array:

```
function noerror
X = [];
nestedfx

function nestedfx
```

```
        makeX  
    end  
end
```

See Also

More About

- “Base and Function Workspaces” on page 20-10

Private Functions

This topic explains the term private function, and shows how to create and use private functions.

Private functions are useful when you want to limit the scope of a function. You designate a function as private by storing it in a subfolder with the name `private`. Then, the function is available only to functions in the folder immediately above the `private` subfolder, or to scripts called by the functions that reside in the parent folder.

For example, within a folder that is on the MATLAB search path, create a subfolder named `private`. Do not add `private` to the path. Within the `private` folder, create a function in a file named `findme.m`:

```
function findme
% FINDME An example of a private function.

disp('You found the private function.')
```

Change to the folder that contains the `private` folder and create a file named `visible.m`.

```
function visible
findme
```

Change your current folder to any location and call the `visible` function.

```
visible
```

```
You found the private function.
```

Although you cannot call the private function from the command line or from functions outside the parent of the `private` folder, you can access its help:

```
help private/findme
```

```
findme An example of a private function.
```

Private functions have precedence over standard functions, so MATLAB finds a private function named `test.m` before a nonprivate program file named `test.m`. This allows you to create an alternate version of a particular function while retaining the original in another folder.

See Also

More About

- “Function Precedence Order” on page 20-43

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 Imported package functions

A package function is associated with a particular folder. When you import a package function using the `import` function, it has precedence over all other functions with the same name.

- 3 Nested functions within the current function

- 4 Local functions within the current file

- 5 Private functions

Private functions are functions in a subfolder named `private` that is immediately below the folder of the currently running file.

- 6 Object functions

An object function accepts a particular class of object in its input argument list. When there are multiple object functions with the same name, MATLAB checks the classes of the input arguments to determine which function to use.

- 7 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.

- 8 Loaded Simulink® models
- 9 Functions in the current folder
- 10 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 files that are not loaded, with file types in this order:
 - a SLX file
 - b MDL file
- 4 App file (.mlapp) created using MATLAB App Designer
- 5 Program file with a .mlx extension
- 6 P-file (that is, an encoded program file with a .p extension)
- 7 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.

See Also

More About

- “What Is the MATLAB Search Path?”
- Variables on page 1-5
- “Types of Functions” on page 20-20
- “Class Precedence and MATLAB Path”

Function Arguments

- “Find Number of Function Arguments” on page 21-2
- “Support Variable Number of Inputs” on page 21-4
- “Support Variable Number of Outputs” on page 21-6
- “Validate Number of Function Arguments” on page 21-8
- “Argument Checking in Nested Functions” on page 21-11
- “Ignore Function Inputs” on page 21-13
- “Check Function Inputs with `validateattributes`” on page 21-14
- “Parse Function Inputs” on page 21-17
- “Input Parser Validation Functions” on page 21-21

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 Cell Array” on page 12-5

More About

- “Argument Checking in Nested Functions” on page 21-11
- “Comma-Separated Lists” on page 2-74

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 Cell Array” on page 12-5

More About

- “Argument Checking in Nested Functions” on page 21-11

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 21-4
- “Support Variable Number of Outputs” on page 21-6

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`

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 vector 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 character vectors 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 character vector `''` for the fourth input to `validateattributes` is a placeholder for an optional function name. 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 `addParameter`. 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 local function).

In the `printPhoto` function, `filename` is a required input. Define `finish` and `color` as optional inputs, 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)
addParameter(p, 'width', defaultWidth, @isnumeric)
addParameter(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 `addParameter` 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 or character vectors and parameter name and value pairs, specify validation functions for the optional inputs. Otherwise, the Input Parser interprets the optional strings or character vectors 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)
The value of 'filename' is invalid. It must satisfy the function: ischar.
```

```
printPhoto('myfile.jpg', 'satin')
```

```
Error using printPhoto (line 23)
The value of 'finish' is invalid. 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')    % positional
```

```
printPhoto('myfile.gif', 'color', 'CMYK')    % name and value
```

See Also

`inputParser` | `varargin`

More About

- “Input Parser Validation Functions” on page 21-21

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 `addParameter` each accept an optional handle to a validation function. Designate function handles with an `@` 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')
```

The value of 'num' is invalid. It must satisfy the function: `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))
```

The value of 'num' is invalid. It must satisfy the function: `@(x) isnumeric(x) && isscalar(x) && (x>0)`.

- Define your own function, typically a local function in the same file as your primary function. For example, in a file named `usenum.m`, define a local function 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
end
```

Call the function with an invalid input:

```
usenum(-1)
```

```
Error using usenum (line 4)
The value of 'num' is invalid. Input must be > 0
```

See Also

[inputParser](#) | [is*](#) | [validateattributes](#)

Related Examples

- “Parse Function Inputs” on page 21-17
- “Create Function Handle” on page 13-2

More About

- “Anonymous Functions” on page 20-24

Debugging MATLAB Code

- “Debug a MATLAB Program” on page 22-2
- “Set Breakpoints” on page 22-8
- “Examine Values While Debugging” on page 22-15

Debug a MATLAB Program

To debug your MATLAB program graphically, use the Editor/Debugger. Alternatively, you can use debugging functions in the Command Window. Both methods are interchangeable.

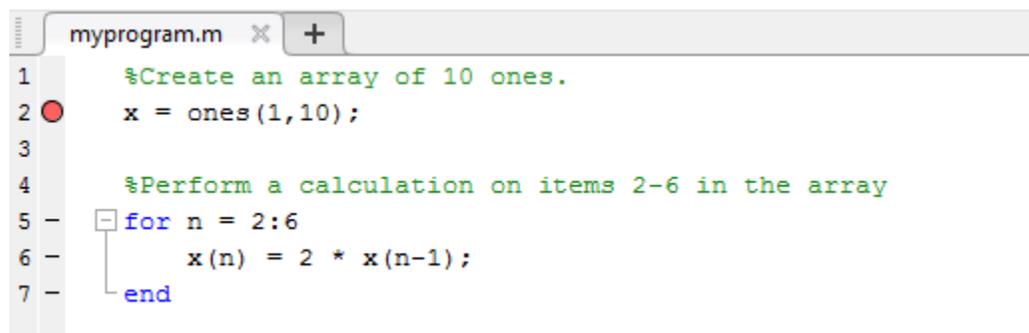
Before you begin debugging, make sure that your program is saved and that the program and any files it calls exist on your search path or in the current folder.

- If you run a file with unsaved changes from within the Editor, then the file is automatically saved before it runs.
- If you run a file with unsaved changes from the Command Window, then MATLAB software runs the saved version of the file. Therefore, you do not see the results of your changes.

Set Breakpoint

Set breakpoints to pause the execution of a MATLAB file so you can examine the value or variables where you think a problem could be. You can set breakpoints using the Editor, using functions in the Command Window, or both.

There are three different types of breakpoints: standard, conditional, and error. To add a *standard* breakpoint in the Editor, click the breakpoint alley at an executable line where you want to set the breakpoint. The breakpoint alley is the narrow column on the left side of the Editor, to the right of the line number. Executable lines are indicated by a dash (—) in the breakpoint alley. For example, click the breakpoint alley next to line 2 in the code below to add a breakpoint at that line.



```
myprogram.m x +
1      %Create an array of 10 ones.
2  ●    x = ones(1,10);
3
4      %Perform a calculation on items 2-6 in the array
5  -    for n = 2:6
6  -        x(n) = 2 * x(n-1);
7  -    end
```

If an executable statement spans multiple lines, you can set a breakpoint at each line in that statement, even though the additional lines do not have a — (dash) in the breakpoint alley. For example, in this code, you can set a breakpoint at all four lines:

```
1 -   if a ...
2         && b
3 -       c = 1;
4 -   end|
```

For more information on the different types of breakpoints, see “Set Breakpoints” on page 22-8.

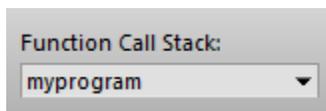
Run File

After setting breakpoints, run the file from the Command Window or the Editor. Running the file produces these results:

- The **Run**  button changes to a **Pause**  button.
- The prompt in the Command Window changes to `K>>` indicating that MATLAB is in debug mode and that the keyboard is in control.
- MATLAB pauses at the first breakpoint in the program. In the Editor, a green arrow just to the right of the breakpoint indicates the pause. The program does not execute the line where the pause occurs until it resumes running. For example, here the debugger pauses before the program executes `x = ones(1,10);`.

```
1   % Create an array of 10 ones.
2   x = ones(1,10);
```

- MATLAB displays the current workspace in the **Function Call Stack**, on the **Editor** tab in the **Debug** section.



If you use debugging functions from the Command Window, use `dbstack` to view the Function Call Stack.

For more information on using the Function Call Stack, see “Select Workspace” on page 22-15

Pause a Running File

To pause the execution of a program while it is running, go to the **Editor** tab and click the **Pause**  button. MATLAB pauses execution at the next executable line, and the **Pause**  button changes to a **Continue**  button. To continue execution, press the **Continue**  button.

Pausing is useful if you want to check on the progress of a long running program to ensure that it is running as expected.

Note Clicking the pause button can cause MATLAB to pause in a file outside your own program file. Pressing the **Continue**  button resumes normal execution without changing the results of the file.

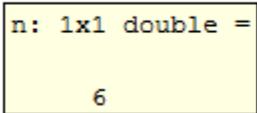
Find and Fix a Problem

While your code is paused, you can view or change the values of variables, or you can modify the code.

View or Change Variable While Debugging

View the value of a variable while debugging to see whether a line of code has produced the expected result or not. To do this, position your mouse pointer to the left of the variable. The current value of the variable appears in a data tip.

```
for n = 2:6
    n: 1x1 double = ) ;
end
```



The data tip stays in view until you move the pointer. If you have trouble getting the data tip to appear, click the line containing the variable, and then move the pointer next to the variable. For more information, see “Examine Values While Debugging” on page 22-15.

You can change the value of a variable while debugging to see if the new value produces expected results. With the program paused, assign a new value to the variable in the Command Window, Workspace browser, or Variables Editor. Then, continue running or stepping through the program.

For example, here MATLAB is paused inside a for loop where $n = 2$:

```

myprogram.m x +
1   %Create an array of 10 ones.
2   x = ones(1,10);
3
4   %Perform a calculation on items 2-6 in the array
5   for n = 2:6
6   |   x(n) = 2 * x(n-1);
7   |   end

```

n: 1x1 double =
2

- Type `n = 7;` in the command line to change the current value of `n` from 2 to 7.
- Press **Continue**  to run the next line of code.

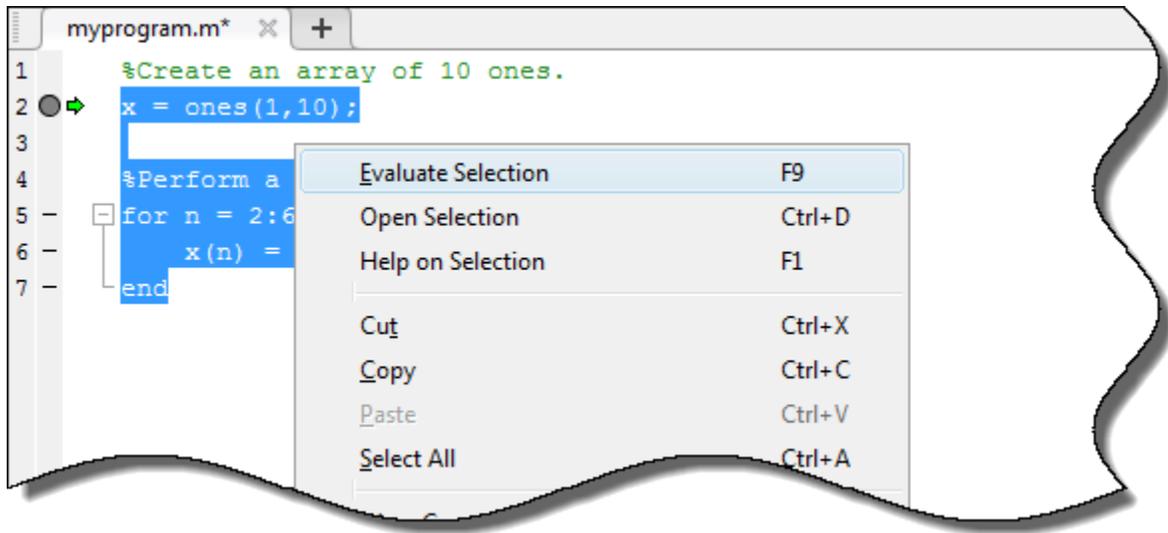
MATLAB runs the code line `x(n) = 2 * x(n-1);` with $n = 7$.

Modify Section of Code While Debugging

You can modify a section of code while debugging to test possible fixes without having to save your changes. Usually, it is a good practice to modify a MATLAB file after you quit debugging, and then save the modification and run the file. Otherwise, you might get unexpected results. However, there are situations where you want to experiment during debugging.

To modify a program while debugging:

- 1 While your code is paused, modify a part of the file that has not yet run.
Breakpoints turn gray, indicating they are invalid.
- 2 Select all the code after the line at which MATLAB is paused, right-click, and then select **Evaluate Selection** from the context menu.



After the code evaluation is complete, stop debugging and save or undo any changes made before continuing the debugging process.

Step Through File

While debugging, you can step through a MATLAB file, pausing at points where you want to examine values.

This table describes available debugging actions and the different methods you can use to execute them.

Description	Toolbar Button	Function Alternative
Continue execution of file until the line where the cursor is positioned. Also available on the context menu.	 Run to Cursor	None

Description	Toolbar Button	Function Alternative
Execute the current line of the file.	 Step	dbstep
Execute the current line of the file and, if the line is a call to another function, step into that function.	 Step In	dbstep in
Resume execution of file until completion or until another breakpoint is encountered.	 Continue	dbcont
After stepping in, run the rest of the called function or local function, leave the called function, and pause.	 Step Out	dbstep out
Pause debug mode.	 Pause	None
Exit debug mode.	 Quit Debugging	dbquit

End Debugging Session

After you identify a problem, end the debugging session by going to the **Editor** tab and clicking **Quit Debugging** . You must end a debugging session if you want to change and save a file, or if you want to run other programs in MATLAB.

After you quit debugging, pause indicators in the Editor display no longer appear, and the normal >> prompt reappears in the Command Window in place of the K>>. You no longer can access the call stack.

If MATLAB software becomes nonresponsive when it pauses at a breakpoint, press **Ctrl+c** to return to the MATLAB prompt.

See Also

Related Examples

- “Set Breakpoints” on page 22-8
- “Examine Values While Debugging” on page 22-15

Set Breakpoints

In this section...

- “Standard Breakpoints” on page 22-9
- “Conditional Breakpoints” on page 22-10
- “Error Breakpoints” on page 22-11
- “Breakpoints in Anonymous Functions” on page 22-11
- “Invalid Breakpoints” on page 22-12
- “Disable Breakpoints” on page 22-13
- “Clear Breakpoints” on page 22-13

Setting breakpoints pauses the execution of your MATLAB program so that you can examine values where you think a problem might be. You can set breakpoints using the Editor or by using functions in the Command Window.

There are three types of breakpoints:

- Standard breakpoints
- Conditional breakpoints
- Error breakpoints

You can set breakpoints only at executable lines in saved files that are in the current folder or in folders on the search path. You can set breakpoints at any time, whether MATLAB is idle or busy running a file.

By default, MATLAB automatically opens files when it reaches a breakpoint. To disable this option:

- 1 From the **Home** tab, in the **Environment** section, click  **Preferences**.

The Preferences dialog box opens.
- 2 Select **MATLAB > Editor/Debugger**.
- 3 Clear the **Automatically open file when MATLAB reaches a breakpoint** option and click **OK**.

Standard Breakpoints

A standard breakpoint pauses at a specified line in a file.

To set a standard breakpoint click the breakpoint alley at an executable line where you want to set the breakpoint. The breakpoint alley is the narrow column on the left side of the Editor, to the right of the line number. Executable lines are indicated by a — (dash) in the breakpoint alley. If an executable statement spans multiple lines, you can set a breakpoint at each line in that statement, even though the additional lines do not have a — (dash) in the breakpoint alley. For example, in this code, you can set a breakpoint at all four lines:

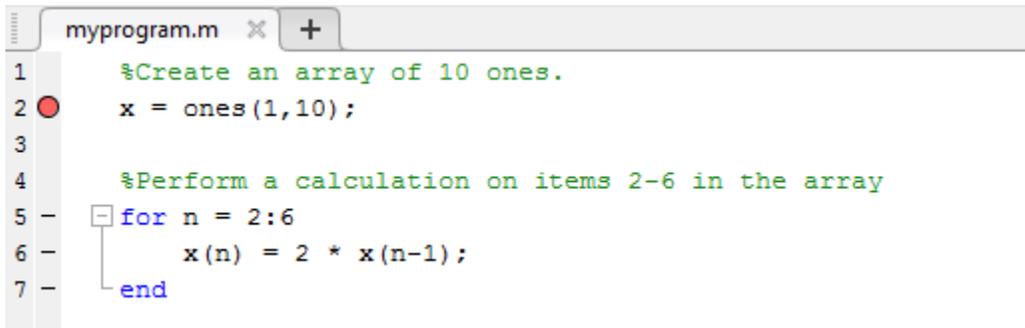
```
1 -   if a ...
2       && b
3 -       c = 1;
4 -   end
```

If you attempt to set a breakpoint at a line that is not executable, such as a comment or a blank line, MATLAB sets it at the next executable line.

You also can set a standard breakpoint programmatically using the `dbstop` function. For example, to add a breakpoint at line 2 in a file named `myprogram.m`, type:

```
dbstop in myprogram at 2
```

MATLAB adds a breakpoint at line 2 in the function `myprogram`.



The screenshot shows the MATLAB Editor window for `myprogram.m`. The code is as follows:

```
1   %Create an array of 10 ones.
2   x = ones(1,10);
3
4   %Perform a calculation on items 2-6 in the array
5   for n = 2:6
6       x(n) = 2 * x(n-1);
7   end
```

A red circle breakpoint is visible in the breakpoint alley to the left of line 2. A mouse cursor is hovering over the breakpoint alley for line 5, which is the start of a `for` loop.

To examine values at increments in a `for` loop, set the breakpoint within the loop, rather than at the start of the loop. If you set the breakpoint at the start of the `for` loop, and then

step through the file, MATLAB pauses at the `for` statement only once. However, if you place the breakpoint within the loop, MATLAB pauses at each pass through the loop.

```

4      % Perform a calculation on items 2 - 6 in the array
5 -   [ ] for n = 2:6
6     [●] x(n) = 2 * x(n - 1);
7 -   [ ] end

```

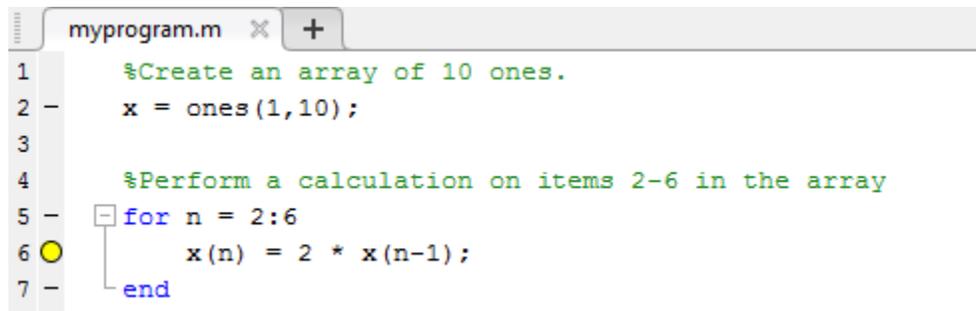
Conditional Breakpoints

A conditional breakpoint causes MATLAB to pause at a specified line in a file only when the specified condition is met. Use conditional breakpoints when you want to examine results after some iterations in a loop.

To set a conditional breakpoint, right-click the breakpoint alley at an executable line where you want to set the breakpoint and select **Set/Modify Condition**.

When the Editor dialog box opens, enter a condition and click **OK**. A condition is any valid MATLAB expression that returns a logical scalar value.

As noted in the dialog box, MATLAB evaluates the condition before running the line. For example, suppose that you have a file called `myprogram.m`.



```

myprogram.m  x  +
1      %Create an array of 10 ones.
2 -   x = ones(1,10);
3
4      %Perform a calculation on items 2-6 in the array
5 -   [ ] for n = 2:6
6     [●] x(n) = 2 * x(n-1);
7 -   [ ] end

```

Add a breakpoint with the following condition at line 6:

```
n >= 4
```

A yellow, conditional breakpoint icon appears in the breakpoint alley at that line.

You also can set a conditional breakpoint programmatically using the `dbstop` function. For example, to add a conditional breakpoint in `myprogram.m` at line 6 type:

```
dbstop in myprogram at 6 if n>=4
```

When you run the file, MATLAB enters debug mode and pauses at the line when the condition is met. In the `myprogram` example, MATLAB runs through the `for` loop twice and pauses on the third iteration at line 6 when `n` is 4. If you continue executing, MATLAB pauses again at line 6 on the fourth iteration when `n` is 5.

Error Breakpoints

An error breakpoint causes MATLAB to pause program execution and enter debug mode if MATLAB encounters a problem. Unlike standard and conditional breakpoints, you do not set these breakpoints at a specific line in a specific file. When you set an error breakpoint, MATLAB pauses at any line in any file if the error condition specified occurs. MATLAB then enters debug mode and opens the file containing the error, with the execution arrow at the line containing the error.

To set an error breakpoint, on the **Editor** tab, click  **Run** and select from these options:

- **Pause on Errors** to pause on all errors.
- **Pause on Warnings** to pause on all warnings.
- **Pause on NaN or Inf** to pause on NaN (not-a-number) or Inf (infinite) values.

You also can set a breakpoint programmatically by using the `dbstop` function with a specified condition. For example, to pause execution on all errors, type

```
dbstop if error
```

To pause execution at the first run-time error within the `try` portion of a `try/catch` block that has a message ID of `MATLAB:ls:InputsMustBeStrings`, type

```
dbstop if caught error MATLAB:ls:InputsMustBeStrings
```

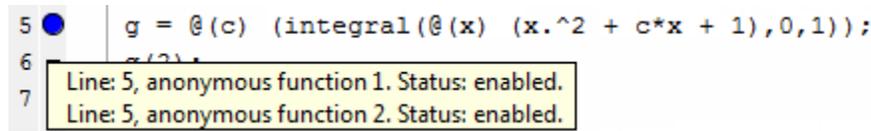
Breakpoints in Anonymous Functions

You can set multiple breakpoints in a line of MATLAB code that contains anonymous functions. For example, you can set a breakpoint for the line itself, where MATLAB

software pauses at the start of the line. Or, alternatively, you can set a breakpoint for each anonymous function in the line.

When you add a breakpoint to a line containing an anonymous function, the Editor asks where in the line you want to add the breakpoint. If there is more than one breakpoint in a line, the breakpoint icon is blue, regardless of the status of any of the breakpoints on that line.

To view information about all the breakpoints on a line, hover your pointer on the breakpoint icon. A tooltip appears with available information. For example, in this code, line 5 contains two anonymous functions, with a breakpoint at each one. The tooltip tells us that both breakpoints are enabled.

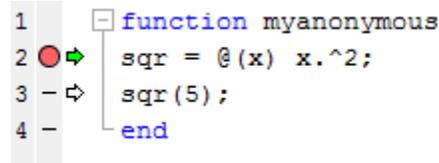


```

5 ● g = @(c) (integral(@(x) (x.^2 + c*x + 1),0,1));
6
7

```

When you set a breakpoint in an anonymous function, MATLAB pauses when the anonymous function is called. A green arrow shows where the code defines the anonymous function. A white arrow shows where the code calls the anonymous functions. For example, in this code, MATLAB pauses the program at a breakpoint set for the anonymous function `sqr`, at line 2 in a file called `myanonymous.m`. The white arrow indicates that the `sqr` function is called from line 3.



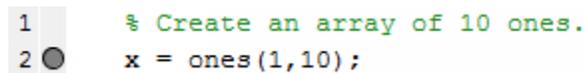
```

1 function myanonymous
2 ● → sqr = @(x) x.^2;
3 - ⇨ sqr(5);
4 - end

```

Invalid Breakpoints

A gray breakpoint indicates an invalid breakpoint.



```

1 % Create an array of 10 ones.
2 ● x = ones(1,10);

```

Breakpoints are invalid for these reasons:

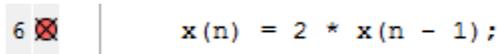
- There are unsaved changes in the file. To make breakpoints valid, save the file. The gray breakpoints become red, indicating that they are now valid.
- There is a syntax error in the file. When you set a breakpoint, an error message appears indicating where the syntax error is. To make the breakpoint valid, fix the syntax error and save the file.

Disable Breakpoints

You can disable selected breakpoints so that your program temporarily ignores them and runs uninterrupted. For example, you might disable a breakpoint after you think you identified and corrected a problem, or if you are using conditional breakpoints.

To disable a breakpoint, right-click the breakpoint icon, and select **Disable Breakpoint** from the context menu.

An X appears through the breakpoint icon to indicate that it is disabled.



To reenable a breakpoint, right-click the breakpoint icon and select **Enable Breakpoint** from the context menu.

The X no longer appears on the breakpoint icon and program execution pauses at that line.

To enable or disable all breakpoints in the file, select **Enable All in File** or **Disable All in File**. These options are only available if there is at least one breakpoint to enable or disable.

Clear Breakpoints

All breakpoints remain in a file until you clear (remove) them or until they are cleared automatically at the end of your MATLAB session.

To clear a breakpoint, right-click the breakpoint icon and select **Clear Breakpoint** from the context menu.

You also can use the `dbclear` function. For example, to clear the breakpoint at line 6 in a file called `myprogram.m`, type

```
dbclear in myprogram at 6
```

To clear all breakpoints in the file, right-click the breakpoint alley and select **Clear All in File**. You can also use the `dbclear all` command. For example, to clear all the breakpoints in a file called `myprogram.m`, type

```
dbclear all in myprogram
```

To clear all breakpoints in *all* files, including error breakpoints, right-click the breakpoint alley and select **Clear All**. You also can use the `dbclear all` command.

Breakpoints clear automatically when you end a MATLAB session. To save your breakpoints for future sessions, see the `dbstatus` function.

See Also

Related Examples

- “Debug a MATLAB Program” on page 22-2
- “Examine Values While Debugging” on page 22-15

Examine Values While Debugging

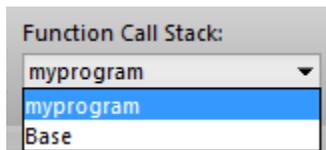
While your program is paused, you can view the value of any variable currently in the workspace. Examine values when you want to see whether a line of code produces the expected result or not. If the result is as expected, continue running or step to the next line. If the result is not as you expect, then that line, or a previous line, might contain an error.

Select Workspace

To examine a variable during debugging, you must first select its workspace. Variables that you assign through the Command Window or create using scripts belong to the base workspace. Variables that you create in a function belong to their own function workspace. To view the current workspace, select the **Editor** tab. The **Function Call Stack** field shows the current workspace. Alternatively, you can use the `dbstack` function in the Command Window.

To select or change the workspace for the variable you want to view, use either of these methods:

- From the **Editor** tab, in the **Debug** section, choose a workspace from the **Function Call Stack** menu list.



- From the Command Window, use the `dbup` and `dbdown` functions to select the previous or next workspace in the Function Call Stack.

To list the variables in the current workspace, use `who` or `whos`.

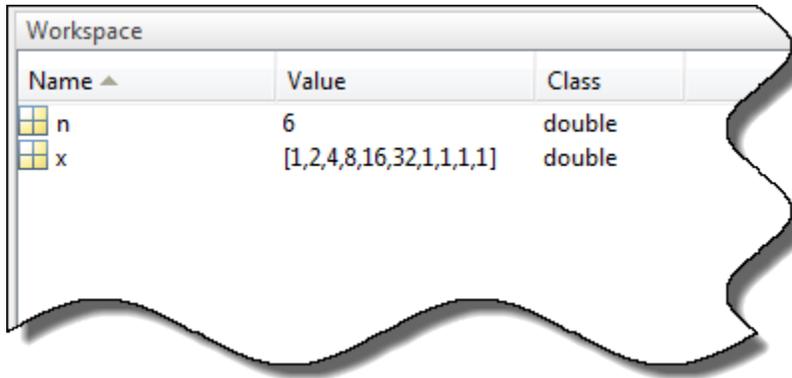
View Variable Value

There are several ways to view the value of a variable while debugging a program:

- View variable values in the Workspace browser and Variables Editor.

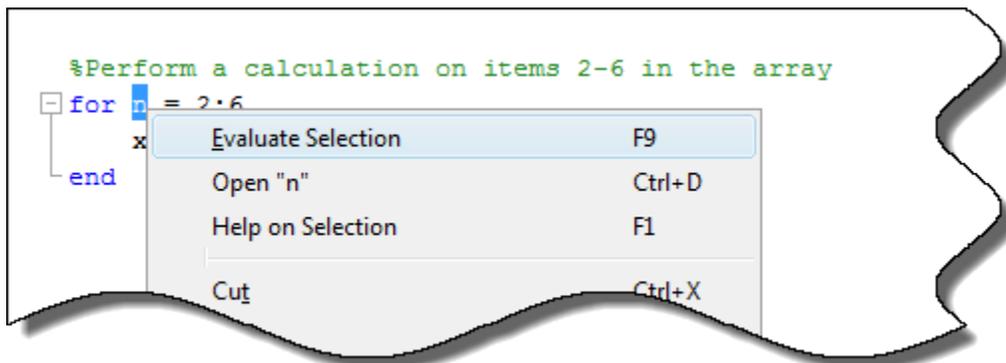
The Workspace browser displays all variables in the current workspace. The **Value** column of the Workspace browser shows the current value of the variable. To see more

details, double-click the variable. The Variables Editor opens, displaying the content for that variable. You also can use the `openvar` function to open a variable in the Variables Editor.



- View variable values in the MATLAB Editor.

Use your mouse to select the variable or equation. Right-click and select **Evaluate Selection** from the context menu. The Command Window displays the value of the variable or equation.



Note You cannot evaluate a selection while MATLAB is busy, for example, running a file.

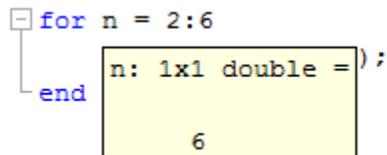
- View variable values as a data tip in the MATLAB Editor.

To do this, position your mouse pointer over the variable. The current value of the variable appears in a data tip. The data tip stays in view until you move the pointer. If you have trouble getting the data tip to appear, click the line containing the variable, and then move the pointer next to the variable.

```

for n = 2:6
    n: 1x1 double = 6;
end

```



Data tips are always enabled when debugging a file in the Editor. To view data tips when editing a file in the Editor, enable them in your MATLAB preferences.

- 1 On the **Home** tab, in the **Environment** section, click  **Preferences**. Then select **MATLAB > Editor/Debugger > Display**.
 - 2 Under **General display options**, select **Enable datatips in edit mode**.
- View variable values in the Command Window.

To see all the variables currently in the workspace, call the `who` function. To view the current value of a variable, type the variable name in the Command Window. For the example, to see the value of a variable `n`, type `n` and press **Enter**. The Command Window displays the variable name and its value.

When you set a breakpoint in a function and attempt to view the value of a variable in a parent workspace, the value of that variable might not be available. This error occurs when you attempt to access a variable while MATLAB is in the process of overwriting it. In such cases, MATLAB returns the following message, where `x` represents the variable whose value you are trying to examine.

```

K>> x
Reference to a called function result under construction x.

```

The error occurs whether you select the parent workspace by using the `dbup` command or by using **Function Call Stack** field in the **Debug** section of the **Editor** tab.

See Also

Related Examples

- “Debug a MATLAB Program” on page 22-2
- “Set Breakpoints” on page 22-8

Presenting MATLAB Code

MATLAB software enables you to present your MATLAB code in various ways. You can share your code and results with others, even if they do not have MATLAB software. You can save MATLAB output in various formats, including HTML, XML, and LaTeX. If Microsoft Word or Microsoft PowerPoint applications are on your Microsoft Windows system, you can publish to their formats as well.

- “Options for Presenting Your Code” on page 23-2
- “Publishing MATLAB Code” on page 23-4
- “Publishing Markup” on page 23-7
- “Output Preferences for Publishing” on page 23-27

Options for Presenting Your Code

MATLAB provides options for presenting your code to others, including using publishing and creating live scripts and functions in the Live Editor.

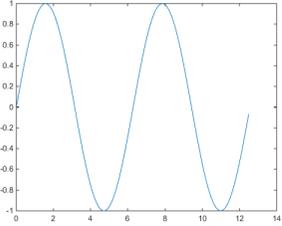
Method	Description	Output Formats	Details
Command-line help	Use comments at the start of a MATLAB file to display help comments when you type <code>help file_name</code> in the Command Window.	<ul style="list-style-type: none"> • ASCII text 	“Add Help for Your Program” on page 20-6
Live Editor	Use the Live Editor to create cohesive, shareable documents that include executable MATLAB code, embedded output, and formatted text.	<ul style="list-style-type: none"> • MLX • HTML • PDF • LaTeX 	“Live Scripts and Functions”
Publish	Use comments with basic markup to publish a document that includes text, bulleted or numbered lists, MATLAB code, and code results.	<ul style="list-style-type: none"> • XML • HTML • LaTeX • Microsoft Word (.doc/.docx) • Microsoft PowerPoint (ppt) • PDF 	“Publishing MATLAB Code” on page 23-4 Publishing MATLAB Code from the Editor video
Help Browser Topics	Create HTML and XML files to provide your own MATLAB help topics for viewing from the MATLAB Help browser or the web.	<ul style="list-style-type: none"> • HTML 	“Display Custom Documentation” on page 30-27

Method	Description	Output Formats	Details
MATLAB Report Generator™	Use MATLAB Report Generator to build complex reports. You must have MATLAB Report Generator software installed.	<ul style="list-style-type: none">• RTF• PDF• Word• HTML• XML	MATLAB Report Generator

Publishing MATLAB Code

Publishing a MATLAB Code file (.m) creates a formatted document that includes your code, comments, and output. Common reasons to publish code are to share the documents with others for teaching or demonstration, or to generate readable, external documentation of your code. To create an interactive document that contains your code, formatted content, and output together in the MATLAB Editor, see “Create Live Scripts in the Live Editor” on page 19-7.

This code demonstrates the Fourier series expansion for a square wave.

MATLAB Code with Markup	Published Document
<pre> %% Square Waves from Sine Waves % The Fourier series expansion for a square-wave is % made up of a sum of odd harmonics, as shown here % using MATLAB(R). ... %% Add an Odd Harmonic and Plot It t = 0:1:pi*4; y = sin(t); plot(t,y); %% % In each iteration of the for loop add an odd % harmonic to y. As _K_ increases, the output % approximates a square wave with increasing accuracy. for k = 3:2:9 %% % Perform the following mathematical operation % at each iteration: % %% $y = y + \frac{\sin(kt)}{k}$ % % Display every other plot: % y = y + sin(k*t)/k; if mod(k,4)=1 display(sprintf('When k = %.1f',k)); display('Then the plot is:'); cla plot(t,y) end end </pre> <p>%% Note About Gibbs Phenomenon % Even though the approximations are constantly % improving, they will never be exact because of the % Gibbs phenomenon, or ringing.</p>	<p>Square Waves from Sine Waves</p> <p>The Fourier series expansion for a square-wave is made up of a sum of odd harmonics, as shown here using MATLAB(R).</p> <p>Contents</p> <ul style="list-style-type: none"> • Add an Odd Harmonic and Plot It • Note About Gibbs Phenomenon <p>Add an Odd Harmonic and Plot It</p> <pre> t = 0:1:pi*4; y = sin(t); plot(t,y); </pre>  <p>In each iteration of the for loop add an odd harmonic to y. As k increases, the output approximates a square wave with increasing accuracy.</p> <pre> for k = 3:2:9 </pre> <p>Perform the following mathematical operation at each iteration:</p> $y = y + \frac{\sin kt}{k}$

To publish your code:

- 1 Create a MATLAB script or function. Divide the code into steps or sections by inserting two percent signs (%%) at the beginning of each section.
- 2 Document the code by adding explanatory comments at the beginning of the file and within each section.

Within the comments at the top of each section, you can add markup that enhances the readability of the output. For example, the code in the preceding table includes the following markup.

Titles	%% Square Waves from Sine Waves %% Add an Odd Harmonic and Plot It %% Note About Gibbs Phenomenon
Variable name in italics	% As <i>_k_</i> increases, ...
LaTeX equation	% $y = y + \frac{\sin(k*t)}{k}$

Note When you have a file containing text that has characters in a different encoding than that of your platform, when you save or publish your file, MATLAB displays those characters as garbled text.

- 3 Publish the code. On the **Publish** tab, click **Publish**.

By default, MATLAB creates a subfolder named `html`, which contains an HTML file and files for each graphic that your code creates. The HTML file includes the code, formatted comments, and output. Alternatively, you can publish to other formats, such as PDF files or Microsoft PowerPoint presentations. For more information on publishing to other formats, see “Specify Output File” on page 23-28.

In MATLAB Online, MATLAB publishes to either HTML or PDF, and stores the output and supporting files in your `Published` folder. To allow MATLAB to open the output automatically, enable pop-up windows in your Web browser. Results are available publicly using a URL of the form `https://matlab.mathworks.com/users/userid/Published/filename/index.html`.

The sample code that appears in the previous figure is part of the installed documentation. You can view the code in the Editor by running this command:

```
edit(fullfile(matlabroot, 'help', 'techdoc', 'matlab_env', ...
    'examples', 'fourier_demo2.m'))
```

See Also

`publish`

More About

- “Options for Presenting Your Code” on page 23-2
- “Publishing Markup” on page 23-7
- “Output Preferences for Publishing” on page 23-27

Publishing Markup

In this section...

“Markup Overview” on page 23-7
“Sections and Section Titles” on page 23-10
“Text Formatting” on page 23-11
“Bulleted and Numbered Lists” on page 23-12
“Text and Code Blocks” on page 23-13
“External File Content” on page 23-14
“External Graphics” on page 23-15
“Image Snapshot” on page 23-17
“LaTeX Equations” on page 23-18
“Hyperlinks” on page 23-20
“HTML Markup” on page 23-23
“LaTeX Markup” on page 23-24

Markup Overview

To insert markup, you can:

- Use the formatting buttons and drop-down menus on the **Publish** tab to format the file. This method automatically inserts the text markup for you.
- Select markup from the **Insert Text Markup** list in the right click menu.
- Type the markup directly in the comments.

The following table provides a summary of the text markup options. Refer to this table if you are not using the MATLAB Editor, or if you do not want to use the **Publish** tab to apply the markup.

Note When working with markup:

- Spaces following the comment symbols (%) often determine the format of the text that follows.

- Starting new markup often requires preceding blank comment lines, as shown in examples.
 - Markup only works in comments that immediately follow a section break.
-

Result in Output	Example of Corresponding File Markup
"Sections and Section Titles" on page 23-10	<pre> %% SECTION TITLE % DESCRIPTIVE TEXT %%% SECTION TITLE WITHOUT SECTION BREAK % DESCRIPTIVE TEXT </pre>
"Text Formatting" on page 23-11	<pre> % _ITALIC TEXT_ % *BOLD TEXT* % MONOSPACED TEXT % Trademarks: % TEXT(TM) % TEXT(R) </pre>
"Bulleted and Numbered Lists" on page 23-12	<pre> %% Bulleted List % % * BULLETED ITEM 1 % * BULLETED ITEM 2 % %% Numbered List % % # NUMBERED ITEM 1 % # NUMBERED ITEM 2 % </pre>

Result in Output	Example of Corresponding File Markup
"Text and Code Blocks" on page 23-13	<pre>%% % % PREFORMATTED % TEXT % %% MATLAB(R) Code % % for i = 1:10 % disp x % end %</pre>
"External File Content" on page 23-14	<pre>% % <include>filename.m</include> %</pre>
"External Graphics" on page 23-15	<pre>% % <<FILENAME.PNG>> %</pre>
"Image Snapshot" on page 23-17	<pre>snapnow;</pre>
"LaTeX Equations" on page 23-18	<pre>%% Inline Expression % \$x^2+e^{\pi i}\$ %% Block Equation % % \$\$e^{\pi i} + 1 = 0\$\$ %</pre>
"Hyperlinks" on page 23-20	<pre>% <https://www.mathworks.com MathWorks> % <matlab:FUNCTION DISPLAYED_TEXT></pre>
"HTML Markup" on page 23-23	<pre>% % <html> % <table border=1><tr> % <td>one</td> % <td>two</td></tr></table> % </html> %</pre>

Result in Output	Example of Corresponding File Markup
"LaTeX Markup" on page 23-24	<pre>%% LaTeX Markup Example % <latex> % \begin{tabular}{ r r } % \hline \$n\$&\$n!\$\\ % \hline 1&1\\ 2&2\\ 3&6\\ % \hline % \end{tabular} % </latex> %</pre>

Sections and Section Titles

Code sections allow you to organize, add comments, and execute portions of your code. Code sections begin with double percent signs (%%) followed by an optional section title. The section title displays as a top-level heading (h1 in HTML), using a larger, bold font.

Note You can add comments in the lines immediately following the title. However, if you want an overall document title, you cannot add any MATLAB code before the start of the next section (a line starting with %%).

For instance, this code produces a polished result when published.

```
%% Vector Operations
% You can perform a number of binary operations on vectors.
%%
A = 1:3;
B = 4:6;
%% Dot Product
% A dot product of two vectors yields a scalar.
% MATLAB has a simple command for dot products.
s = dot(A,B);
%% Cross Product
% A cross product of two vectors yields a third
% vector perpendicular to both original vectors.
% Again, MATLAB has a simple command for cross products.
v = cross(A,B);
```

By saving the code in an Editor and clicking the **Publish** button on the **Publish** tab, MATLAB produces the output as shown in this figure. Notice that MATLAB automatically inserts a Contents menu from the section titles in the MATLAB file.

Vector Operations

You can perform a number of binary operations on vectors.

Contents

- [Dot Product](#)
- [Cross Product](#)

```
A = 1:3;
B = 4:6;
```

Dot Product

A dot product of two vectors yields a scalar. MATLAB has a simple command for dot products.

```
s = dot(A,B);
```

Cross Product

A cross product of two vectors yields a third vector perpendicular to both original vectors. Again, MATLAB has a simple command for cross products.

```
v = cross(A,B);
```

Text Formatting

You can mark selected text in the MATLAB comments so that they display in italic, bold, or monospaced text when you publish the file. Simply surround the text with `_`, `*`, or `|` for italic, bold, or monospaced text, respectively.

For instance, these lines display each of the text formatting syntaxes if published.

```
%% Calculate and Plot Sine Wave
% _Define_ the *range* for |x|
```

Calculate and Plot Sine Wave

Define the range for x

Trademark Symbols

If the comments in your MATLAB file include trademarked terms, you can include text to produce a trademark symbol (™) or registered trademark symbol (®) in the output. Simply add (R) or (TM) directly after the term in question, without any space in between.

For example, suppose that you enter these lines in a file.

```
%% Basic Matrix Operations in MATLAB(R)
% This is a demonstration of some aspects of MATLAB(R)
% and the Symbolic Math Toolbox(TM).
```

If you publish the file to HTML, it appears in the MATLAB web browser.



Bulleted and Numbered Lists

MATLAB allows bulleted and numbered lists in the comments. You can use this syntax to produce bulleted and numbered lists.

```
%% Two Lists
%
% * ITEM1
% * ITEM2
%
% # ITEM1
% # ITEM2
%
```

Publishing the example code produces this output.

Two Lists

- ITEM1
 - ITEM2
1. ITEM1
 2. ITEM2

Text and Code Blocks

Preformatted Text

Preformatted text appears in monospace font, maintains white space, and does not wrap long lines. Two spaces must appear between the comment symbol and the text of the first line of the preformatted text.

Publishing this code produces a preformatted paragraph.

```
%%
% Many people find monospaced texts easier to read:
%
% A dot product of two vectors yields a scalar.
% MATLAB has a simple command for dot products.
```

Many people find monospaced texts easier to read:

A dot product of two vectors yields a scalar.
MATLAB has a simple command for dot products.

Syntax Highlighted Sample Code

Executable code appears with syntax highlighting in published documents. You also can highlight *sample code*. Sample code is code that appears within comments.

To indicate sample code, you must put three spaces between the comment symbol and the start of the first line of code. For example, clicking the **Code** button on the **Publish** tab inserts the following sample code in your Editor.

```
%%  
%  
%   for i = 1:10  
%       disp(x)  
%   end  
%
```

Publishing this code to HTML produces output in the MATLAB web browser.



```
for i = 1:10  
    disp(x)  
end
```

External File Content

To add external file content into MATLAB published code, use the `<include>` markup. Specify the external file path relative to the location of the published file. Included MATLAB code files publish as syntax highlighted code. Any other files publish as plain text.

For example, this code inserts the contents of `sine_wave.m` into your published output:

```
%% External File Content Example  
% This example includes the file contents of sine_wave.m into published  
% output.  
%  
% <include>sine_wave.m</include>  
%  
% The file content above is properly syntax highlighted
```

Publish the file to HTML.

External File Content Example

This example includes the file contents of `sine_wave.m` into published output.

```
% Define the range for x.
% Calculate and plot y = sin(x).
x = 0:1:6*pi;
y = sin(x);
plot(x,y)
title('Sine Wave')
xlabel('x')
ylabel('sin(x)')
fig = gcf;
fig.MenuBar = 'none';
```

The file content above is properly syntax highlighted

External Graphics

To publish an image that the MATLAB code does not generate, use text markup. By default, MATLAB already includes code-generated graphics.

This code inserts a generic image called `FILENAME.PNG` into your published output.

```
%%
%
% <<FILENAME.PNG>>
%
```

MATLAB requires that `FILENAME.PNG` be a relative path from the output location to your external image or a fully qualified URL. Good practice is to save your image in the same folder that MATLAB publishes its output. For example, MATLAB publishes HTML documents to a subfolder `html`. Save your image file in the same subfolder. You can change the output folder by changing the publish configuration settings. In MATLAB Online, save your image file to your `Published` folder, which is located in your root folder.

External Graphics Example Using `surf(peaks)`

This example shows how to insert `surfpeaks.jpg` into a MATLAB file for publishing.

To create the `surfpeaks.jpg`, run this code in the Command Window.

```
saveas(surf(peaks), 'surfpeaks.jpg');
```

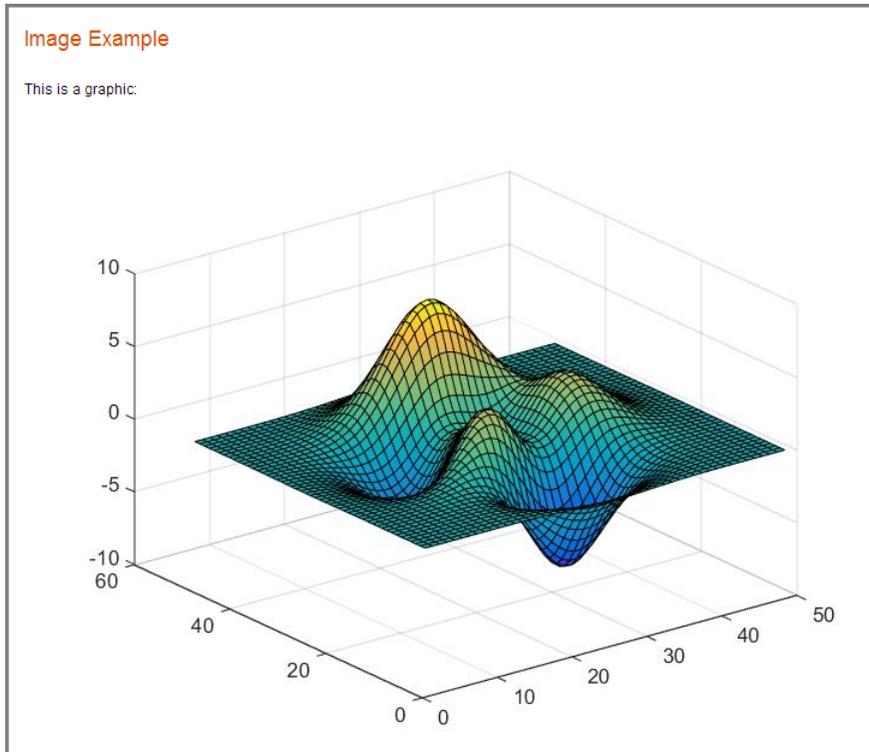
To produce an HTML file containing `surfpeaks.jpg` from a MATLAB file:

- 1 Create a subfolder called `html` in your current folder.
- 2 Create `surfpeaks.jpg` by running this code in the Command Window.

```
saveas(surf(peaks), 'html/surfpeaks.jpg');
```

- 3 Publish this MATLAB code to HTML.

```
%% Image Example  
% This is a graphic:  
%  
% <<surfpeaks.jpg>>  
%
```



Valid Image Types for Output File Formats

The type of images you can include when you publish depends on the output type of that document as indicated in this table. For greatest compatibility, best practice is to use the default image format for each output type.

Output File Format	Default Image Format	Types of Images You Can Include
doc	png	Any format that your installed version of Microsoft Office supports.
html	png	All formats publish successfully. Ensure that the tools you use to view and process the output files can display the output format you specify.
latex	png or epsc2	All formats publish successfully. Ensure that the tools you use to view and process the output files can display the output format you specify.
pdf	bmp	bmp and jpg.
ppt	png	Any format that your installed version of Microsoft Office supports.
xml	png	All formats publish successfully. Ensure that the tools you use to view and process the output files can display the output format you specify.

Image Snapshot

You can insert code that captures a snapshot of your MATLAB output. This is useful, for example, if you have a `for` loop that modifies a figure that you want to capture after each iteration.

The following code runs a `for` loop three times and produces output after every iteration. The `snapnow` command captures all three images produced by the code.

```

%% Scale magic Data and Display as Image

for i=1:3
    imagesc(magic(i))
    snapnow;
end

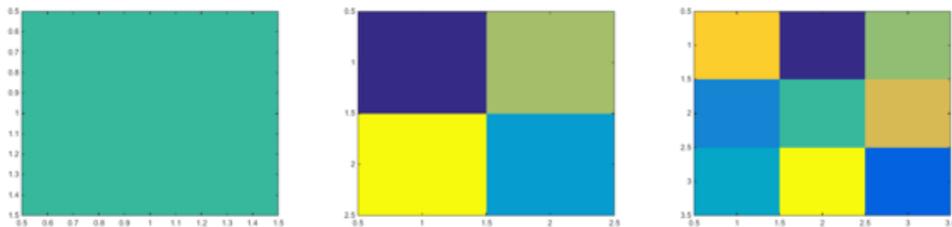
```

If you publish the file to HTML, it resembles the following output. By default, the images in the HTML are larger than shown in the figure. To resize images generated by MATLAB

code, use the **Max image width** and **Max image height** fields in the **Publish settings** pane, as described in “Output Preferences for Publishing” on page 23-27.

Scale magic Data and Display as Image

```
for i=1:3
    imagesc(magic(i))
    snapnow;
end
```



LaTeX Equations

Inline LaTeX Expression

MATLAB enables you to include an inline LaTeX expression in any code that you intend to publish. To insert an inline expression, surround your LaTeX markup with dollar sign characters (\$). The \$ must immediately precede the first word of the inline expression, and immediately follow the last word of the inline expression, without any space in between.

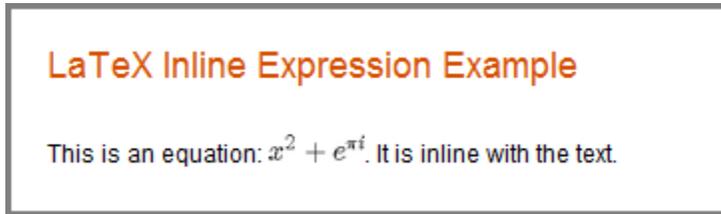
Note

- All publishing output types support LaTeX expressions, except Microsoft PowerPoint.
 - MATLAB publishing supports standard LaTeX math mode directives. Text mode directives or directives that require additional packages are not supported.
-

This code contains a LaTeX expression:

```
%% LaTeX Inline Expression Example
%
% This is an equation:  $x^2+e^{\pi i}$ . It is
% inline with the text.
```

If you publish the sample text markup to HTML, this is the resulting output.



LaTeX Display Equation

MATLAB enables you to insert LaTeX symbols in blocks that are offset from the main comment text. Two dollar sign characters (\$\$) on each side of an equation denote a block LaTeX equation. Publishing equations in separate blocks requires a blank line in between blocks.

This code is a sample text markup.

```
%% LaTeX Equation Example
%
% This is an equation:
%
% 
$$e^{\pi i} + 1 = 0$$

%
% It is not in line with the text.
```

If you publish to HTML, the expression appears as shown here.

LaTeX Equation Example

This is an equation:

$$e^{\pi i} + 1 = 0$$

It is not in line with the text.

Hyperlinks

Static Hyperlinks

You can insert static hyperlinks within a MATLAB comment, and then publish the file to HTML, XML, or Microsoft Word. When specifying a static hyperlink to a web location, include a complete URL within the code. This is useful when you want to point the reader to a web location. You can display or hide the URL in the published text. Consider excluding the URL, when you are confident that readers are viewing your output online and can click the hyperlink.

Enclose URLs and any replacement text in angled brackets.

```
%%  
% For more information, see our web site:  
% <https://www.mathworks.com MathWorks>
```

Publishing the code to HTML produces this output.

For more information, see our Web site: [MathWorks](https://www.mathworks.com)

Eliminating the text MathWorks after the URL produces this modified output.

For more information, see our Web site: <http://www.mathworks.com>

Note If your code produces hyperlinked text in the MATLAB Command Window, the output shows the HTML code rather than the hyperlink.

Dynamic Hyperlinks

You can insert dynamic hyperlinks, which MATLAB evaluates at the time a reader clicks that link. Dynamic hyperlinks enable you to point the reader to MATLAB code or documentation, or enable the reader to run code. You implement these links using `matlab:` syntax. If the code that follows the `matlab:` declaration has spaces in it, replace them with `%20`.

Note Dynamic links only work when viewing HTML in the MATLAB web browser.

Diverse uses of dynamic links include:

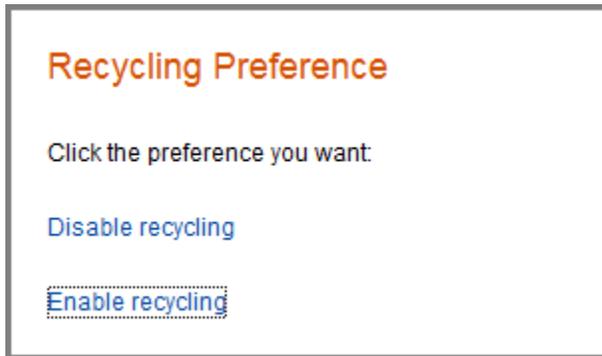
- “Dynamic Link to Run Code” on page 23-21
- “Dynamic Link to a File” on page 23-22
- “Dynamic Link to a MATLAB Function Reference Page” on page 23-22

Dynamic Link to Run Code

You can specify a dynamic hyperlink to run code when a user clicks the hyperlink. For example, this `matlab:` syntax creates hyperlinks in the output, which when clicked either enable or disable recycling:

```
%% Recycling Preference
% Click the preference you want:
%
% <matlab:recycle('off') Disable recycling>
%
% <matlab:recycle('on') Enable recycling>
```

The published result resembles this HTML output.



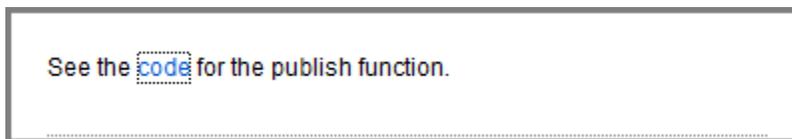
When you click one of the hyperlinks, MATLAB sets the `recycle` command accordingly. After clicking a hyperlink, run `recycle` in the Command Window to confirm that the setting is as you expect.

Dynamic Link to a File

You can specify a link to a file that you know is in the `matlabroot` of your reader. You do not need to know where each reader installed MATLAB. For example, link to the function code for `publish`.

```
%%  
% See the  
% <matlab:edit(fullfile(matlabroot,'toolbox','matlab','codetools','publish.m')) code>  
% for the publish function.
```

Next, publish the file to HTML.



When you click the `code` link, the MATLAB Editor opens and displays the code for the `publish` function. On the reader's system, MATLAB issues the command (although the command does not appear in the reader's Command Window).

Dynamic Link to a MATLAB Function Reference Page

You can specify a link to a MATLAB function reference page using `matlab:` syntax. For example, suppose that your reader has MATLAB installed and running. Provide a link to the `publish` reference page.

```
%%  
% See the help for the <matlab:doc('publish') publish> function.
```

Publish the file to HTML.



See the help for the [publish](#) function.

When you click the `publish` hyperlink, the MATLAB Help browser opens and displays the reference page for the `publish` function. On the reader's system, MATLAB issues the command, although the command does not appear in the Command Window.

HTML Markup

You can insert HTML markup into your MATLAB file. You must type the HTML markup since no button on the **Publish** tab generates it.

Note When you insert text markup for HTML code, the HTML code publishes only when the specified output file format is HTML.

This code includes HTML tagging.

```
%% HTML Markup Example  
% This is a table:  
%  
% <html>  
% <table border=1><tr><td>one</td><td>two</td></tr>  
% <tr><td>three</td><td>four</td></tr></table>  
% </html>  
%
```

If you publish the code to HTML, MATLAB creates a single-row table with two columns. The table contains the values one, two, three, and four.

HTML Markup Example

This is a table:

one	two
three	four

If a section produces command-window output that starts with `<html>` and ends with `</html>`, MATLAB includes the source HTML in the published output. For example, MATLAB displays the `disp` command and makes a table from the HTML code if you publish this code:

```
disp('<html><table><tr><td>1</td><td>2</td></tr></table></html>')
```

```
disp('<html><table><tr><td>1</td><td>2</td></tr></table></html>')
```

1	2
---	---

LaTeX Markup

You can insert LaTeX markup into your MATLAB file. You must type all LaTeX markup since no button on the **Publish** tab generates it.

Note When you insert text markup for LaTeX code, that code publishes only when the specified output file format is LaTeX.

This code is an example of LaTeX markup.

```

%% LaTeX Markup Example
% This is a table:
%
% <latex>
% \begin{tabular}{|c|c|} \hline
% $n$ & $n!$ \\ \hline
% 1 & 1 \\
% 2 & 2 \\
% 3 & 6 \\ \hline
% \end{tabular}
% </latex>

```

If you publish the file to LaTeX, then the Editor opens a new .tex file containing the LaTeX markup.

```

% This LaTeX was auto-generated from MATLAB code.
% To make changes, update the MATLAB code and republish this document.

```

```

\documentclass{article}
\usepackage{graphicx}
\usepackage{color}

\sloppy
\definecolor{lightgray}{gray}{0.5}
\setlength{\parindent}{0pt}

\begin{document}

\section*{LaTeX Markup Example}

\begin{par}
This is a table:
\end{par} \vspace{1em}
\begin{par}

\begin{tabular}{|c|c|} \hline
$n$ & $n!$ \\ \hline
1 & 1 \\
2 & 2 \\
3 & 6 \\ \hline
\end{tabular}

```

```
\end{par} \vspace{1em}
```

```
\end{document}
```

MATLAB includes any additional markup necessary to compile this file with a LaTeX program.

See Also

More About

- “Options for Presenting Your Code” on page 23-2
- “Publishing MATLAB Code” on page 23-4
- “Output Preferences for Publishing” on page 23-27

Output Preferences for Publishing

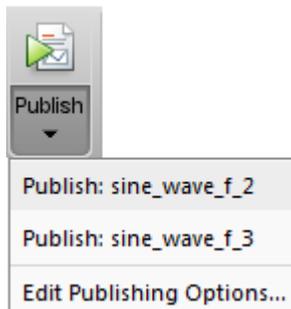
In this section...

- “How to Edit Publishing Options” on page 23-27
- “Specify Output File” on page 23-28
- “Run Code During Publishing” on page 23-29
- “Manipulate Graphics in Publishing Output” on page 23-31
- “Save a Publish Setting” on page 23-36
- “Manage a Publish Configuration” on page 23-38

How to Edit Publishing Options

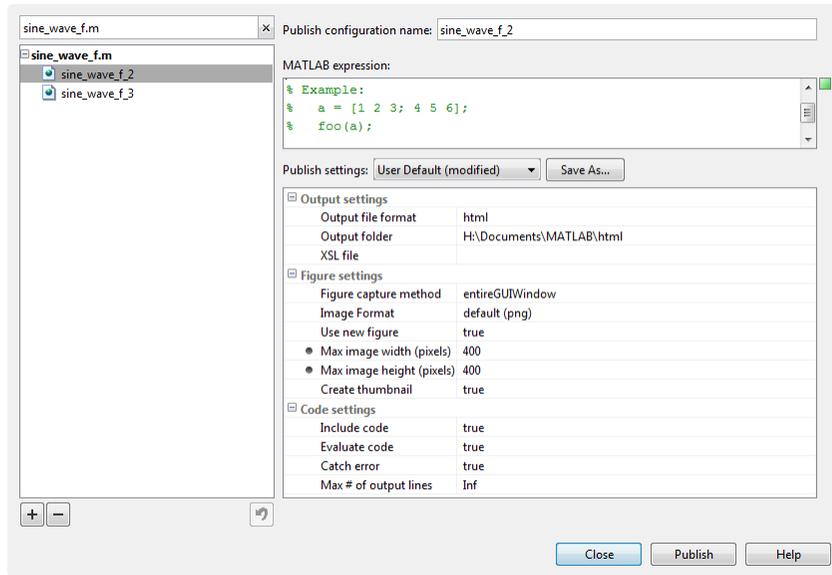
Use the default publishing preferences if your code requires no input arguments and you want to publish to HTML. However, if your code requires input arguments, or if you want to specify output settings, code execution, or figure formats, then specify a custom configuration.

- 1 Locate the **Publish** tab and click the **Publish** button arrow ↕.



- 2 Select **Edit Publishing Options**.

The Edit Configurations dialog box opens. Specify output preferences.



The **MATLAB expression** pane specifies the code that executes during publishing. The **Publish settings** pane contains output, figure, and code execution options. Together, they make what MATLAB refers to as a *publish configuration*. MATLAB associates each publish configuration with an `.m` file. The name of the publish configuration appears in the top left pane.

Specify Output File

You specify the output format and location on the **Publish settings** pane.

MATLAB publishes to these formats.

Format	Notes
html	Publishes to an HTML document. You can use an Extensible Stylesheet Language (XSL) file.
xml	Publishes to XML document. You can use an Extensible Stylesheet Language (XSL) file.
latex	Publishes to LaTeX document. Does <i>not</i> preserve syntax highlighting. You can use an Extensible Stylesheet Language (XSL) file.

Format	Notes
doc	Publishes to a Microsoft Word document. Does <i>not</i> preserve syntax highlighting. This format is only available on Windows platforms.
ppt	Publishes to a Microsoft PowerPoint document. Does <i>not</i> preserve syntax highlighting. This format is only available on Windows platforms.
pdf	Publishes to a PDF document.

Note XSL files allow you more control over the appearance of the output document. For more details, see <http://docbook.sourceforge.net/release/xsl/current/doc/>.

Run Code During Publishing

- “Specifying Code” on page 23-29
- “Evaluating Code” on page 23-30
- “Including Code” on page 23-30
- “Catching Errors” on page 23-31
- “Limiting the Amount of Output” on page 23-31

Specifying Code

By default, MATLAB executes the `.m` file that you are publishing. However, you can specify any valid MATLAB code in the **MATLAB expression** pane. For example, if you want to publish a function that requires input, then run the command `function(input)`. Additional code, whose output you want to publish, appears after the functions call. If you clear the **MATLAB expression** area, then MATLAB publishes the file without evaluating any code.

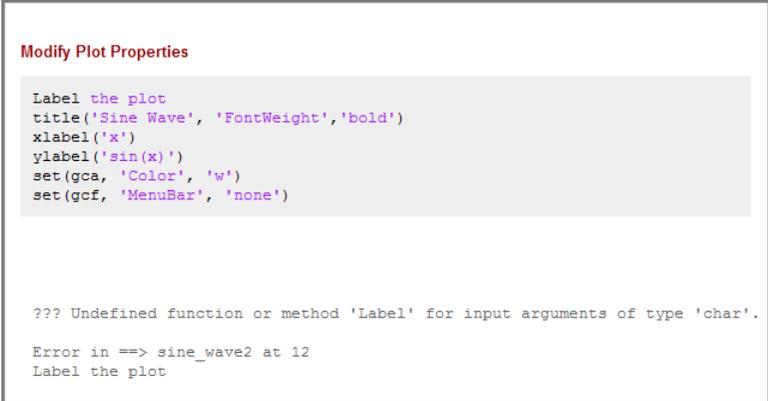
Note Publish configurations use the base MATLAB workspace. Therefore, a variable in the **MATLAB expression** pane overwrites the value for an existing variable in the base workspace.

Evaluating Code

Another way to affect what MATLAB executes during publishing is to set the **Evaluate code** option in the **Publish setting** pane. This option indicates whether MATLAB evaluates the code in the `.m` file that is publishing. If set to `true`, MATLAB executes the code and includes the results in the output document.

Because MATLAB does not evaluate the code nor include code results when you set the **Evaluate code** option to `false`, there can be invalid code in the file. Therefore, consider first running the file with this option set to `true`.

For example, suppose that you include comment text, `Label the plot`, in a file, but forget to preface it with the comment character. If you publish the document to HTML, and set the **Evaluate code** option to `true`, the output includes an error.

A screenshot of a MATLAB error message window titled "Modify Plot Properties". The window contains a code editor with the following MATLAB code:

```
Label the plot
title('Sine Wave', 'FontWeight','bold')
xlabel('x')
ylabel('sin(x)')
set(gca, 'Color', 'w')
set(gcf, 'MenuBar', 'none')
```

Below the code editor, the error message reads: "??? Undefined function or method 'Label' for input arguments of type 'char'." Below the error message, the stack trace shows: "Error in ==> sine_wave2 at 12" and "Label the plot".

Use the `false` option to publish the file that contains the `publish` function. Otherwise, MATLAB attempts to publish the file recursively.

Including Code

You can specify whether to display MATLAB code in the final output. If you set the **Include code** option to `true`, then MATLAB includes the code in the published output document. If set to `false`, MATLAB excludes the code from all output file formats, except HTML.

If the output file format is HTML, MATLAB inserts the code as an HTML comment that is not visible in the web browser. If you want to extract the code from the output HTML file, use the MATLAB `grabcode` function.

For example, suppose that you publish `H:/my_matlabfiles/my_mfiles/sine_wave.m` to HTML using a publish configuration with the **Include code** option set to `false`. If you share the output with colleagues, they can view it in a web browser. To see the MATLAB code that generated the output, they can issue the following command from the folder containing `sine_wave.html`:

```
grabcode('sine_wave.html')
```

MATLAB opens the file that created `sine_wave.html` in the Editor.

Catching Errors

You can catch and publish any errors that occur during publishing. Setting the **Catch error** option to `true` includes any error messages in the output document. If you set **Catch error** to `false`, MATLAB terminates the publish operation if an error occurs during code evaluation. However, this option has no effect if you set the **Evaluate code** property to `false`.

Limiting the Amount of Output

You can limit the number of lines of code output that is included in the output document by specifying the **Max # of output lines** option in the **Publish settings** pane. Setting this option is useful if a smaller, representative sample of the code output suffices.

For example, the following loop generates 100 lines in a published output unless **Max # of output lines** is set to a lower value.

```
for n = 1:100
    disp(x)
end;
```

Manipulate Graphics in Publishing Output

- “Choosing an Image Format” on page 23-32
- “Setting an Image Size” on page 23-32
- “Capturing Figures” on page 23-33
- “Specifying a Custom Figure Window” on page 23-33
- “Creating a Thumbnail” on page 23-36

Choosing an Image Format

When publishing, you can choose the image format that MATLAB uses to store any graphics generated during code execution. The available image formats in the drop-down list depend on the setting of the **Figure capture method** option. For greatest compatibility, select the default as specified in this table.

Output File Format	Default Image Format	Types of Images You Can Include
doc	png	Any format that your installed version of Microsoft Office supports.
html	png	All formats publish successfully. Ensure that the tools you use to view and process the output files can display the output format you specify.
latex	png or eps	All formats publish successfully. Ensure that the tools you use to view and process the output files can display the output format you specify.
pdf	bmp	bmp and jpg.
ppt	png	Any format that your installed version of Microsoft Office supports.
xml	png	All formats publish successfully. Ensure that the tools you use to view and process the output files can display the output format you specify.

Setting an Image Size

You set the size of MATLAB generated images in the **Publish settings** pane on the Edit Configurations dialog window. You specify the image size in pixels to restrict the width and height of images in the output. The pixel values act as a maximum size value because MATLAB maintains an image's aspect ratio. MATLAB ignores the size setting for the following cases:

- When working with external graphics as described in “External Graphics” on page 23-15
- When using vector formats, such as .eps

- When publishing to .pdf

Capturing Figures

You can capture different aspects of the Figure window by setting the **Figure capture method** option. This option determines the window decorations (title bar, toolbar, menu bar, and window border) and plot backgrounds for the Figure window.

This table summarizes the effects of the various Figure capture methods.

Use This Figure Capture Method	To Get Figure Captures with These Appearance Details	
	Window Decorations	Plot Backgrounds
entireGUIWindow	Included for dialog boxes; Excluded for figures	Set to white for figures; matches the screen for dialog boxes
print	Excluded for dialog boxes and figures	Set to white
getframe	Excluded for dialog boxes and figures	Matches the screen plot background
entireFigureWindow	Included for dialog boxes and figures	Matches the screen plot background

Note Typically, MATLAB figures have the `HandleVisibility` property set to `on`. Dialog boxes are figures with the `HandleVisibility` property set to `off` or `callback`. If your results are different from the results listed in the preceding table, the `HandleVisibility` property of your figures or dialog boxes might be atypical. For more information, see `HandleVisibility`.

Specifying a Custom Figure Window

MATLAB allows you to specify custom appearance for figures it creates. If the **Use new figure** option in the **Publish settings** pane is set to `true`, then in the published output, MATLAB uses a Figure window at the default size and with a white background. If the **Use new figure** option is set to `false`, then MATLAB uses the properties from an open Figure window to determine the appearance of code-generated figures. This preference does not apply to figures included using the syntax in “External Graphics” on page 23-15.

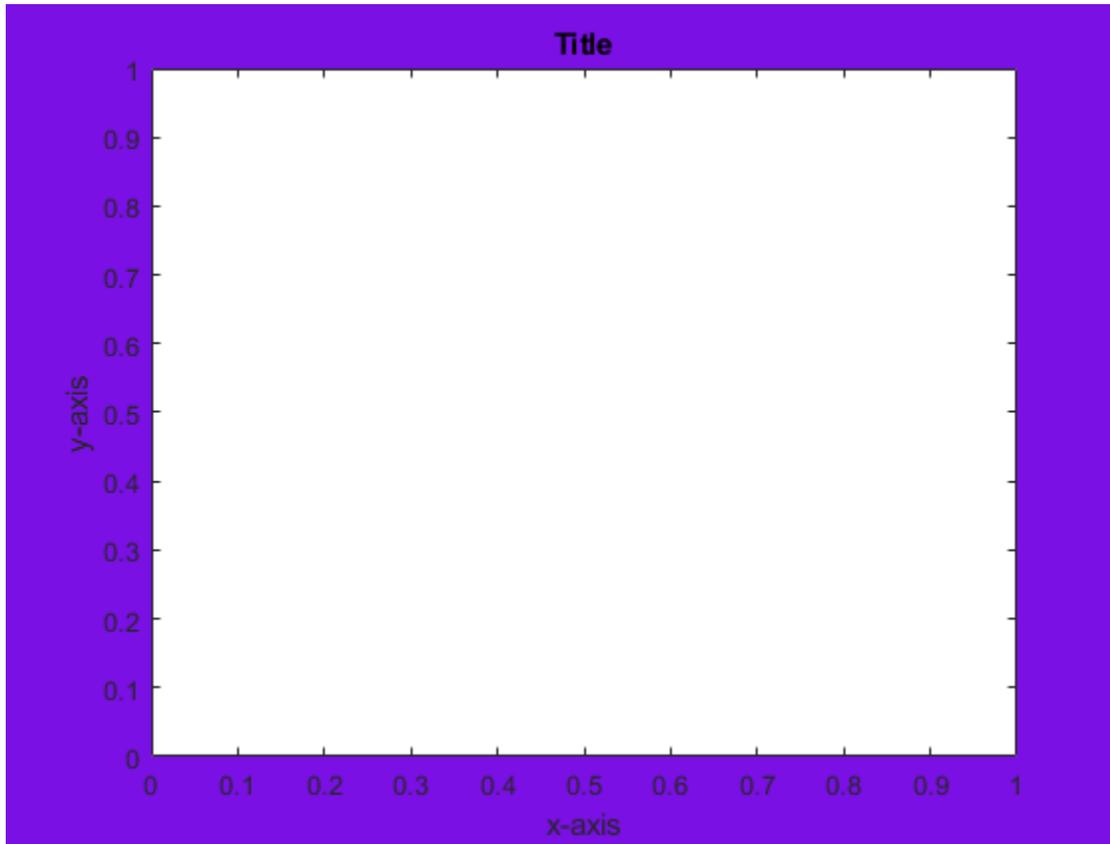
Use the following code as a template to produce Figure windows that meet your needs.

```
% Create figure
figure1 = figure('Name', 'purple_background', ...
'Color', [0.4784 0.06275 0.8941]);
colormap('hsv');

% Create subplot
subplot(1,1,1, 'Parent', figure1);
box('on');

% Create axis labels
xlabel('x-axis');
ylabel({'y-axis'})

% Create title
title({'Title'});
```



```
% Enable printed output to match colors on screen  
set(figure1, 'InvertHardcopy', 'off')
```

By publishing your file with this window open and the **Use new figure** option set to false, any code-generated figure takes the properties of the open Figure window.

Note You must set the **Figure capture method** option to **entireFigureWindow** for the final published figure to display all the properties of the open Figure window.

Creating a Thumbnail

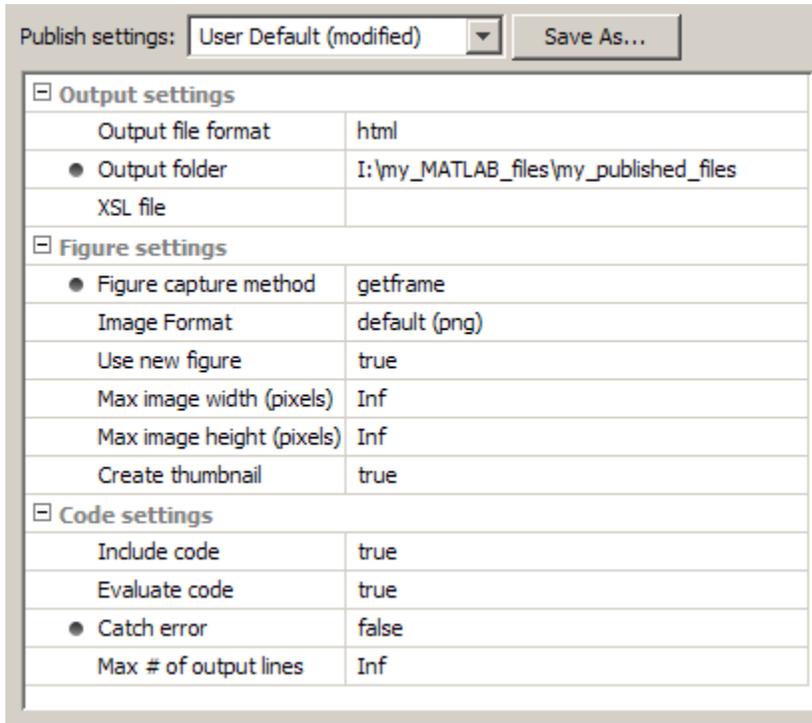
You can save the first code-generated graphic as a thumbnail image. You can use this thumbnail to represent your file on HTML pages. To create a thumbnail, follow these steps:

- 1** On the **Publish** tab, click the Publish button drop-down arrow ▾ and select **Edit Publishing Options**. The Edit Configurations dialog box opens.
- 2** Set the **Image Format** option to a bitmap format, such as `.png` or `.jpg`. MATLAB creates thumbnail images in bitmap formats.
- 3** Set the **Create thumbnail** option to `true`.

MATLAB saves the thumbnail image in the folder specified by the **Output folder** option in the **Publish settings** pane.

Save a Publish Setting

You can save your publish settings, which allows you to reproduce output easily. It can be useful to save your commonly used publish settings.



When the **Publish settings** options are set, you can follow these steps to save the settings:

- 1 Click **Save As** when the options are set in the manner you want.

The **Save Publish Settings As** dialog box opens and displays the names of all the currently defined publish settings. By default the following publish settings install with MATLAB:

- Factory Default

You cannot overwrite the Factory Default and can restore them by selecting Factory Default from the **Publish settings** list.

- User Default

Initially, User Default settings are identical to the Factory Default settings. You can overwrite the User Default settings.

- 2 In the **Settings Name** field, enter a meaningful name for the settings. Then click **Save**.

You can now use the publish settings with other MATLAB files.

You also can overwrite the publishing properties saved under an existing name. Select the name from the **Publish settings** list, and then click **Overwrite**.

Manage a Publish Configuration

- “Running an Existing Publish Configuration” on page 23-38
- “Creating Multiple Publish Configurations for a File” on page 23-38
- “Reassociating and Renaming Publish Configurations” on page 23-39
- “Using Publish Configurations Across Different Systems” on page 23-40

Together, the code in the **MATLAB expression** pane and the settings in the **Publish settings** pane make a publish configuration that is associated with one file. These configurations provide a simple way to refer to publish preferences for individual files.

To create a publish configuration, click the **Publish** button drop-down arrow \blacktriangledown on the **Publish** tab, and select **Edit Publishing Options**. The Edit Configurations dialog box opens, containing the default publish preferences. In the **Publish configuration name** field, type a name for the publish configuration, or accept the default name. The publish configuration saves automatically.

Running an Existing Publish Configuration

After saving a publish configuration, you can run it without opening the Edit Configurations dialog box:

- 1 Click the **Publish** button drop-down arrow \blacktriangledown . If you position your mouse pointer on a publish configuration name, MATLAB displays a tooltip showing the MATLAB expression associated with the specific configuration.
- 2 Select a configuration name to use for the publish configuration. MATLAB publishes the file using the code and publish settings associated with the configuration.

Creating Multiple Publish Configurations for a File

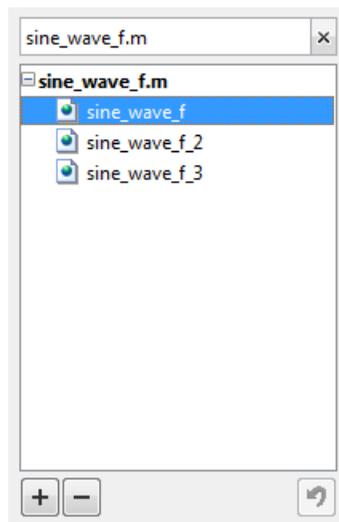
You can create multiple publish configurations for a given file. You might do this to publish the file with different values for input arguments, with different publish setting

property values, or both. Create a named configuration for each purpose, all associated with the same file. Later you can run whichever particular publish configuration you want.

Use the following steps as a guide to create new publish configurations.

- 1 Open a file in your Editor.
- 2 Click the **Publish** button drop-down arrow, and select **Edit Publishing Options**. The Edit Configurations dialog box opens.
- 3 Click the **Add** button **+** located on the left pane.

A new name appears on the configurations list, *filename_n*, where the value of *n* depends on the existing configuration names.



- 4 If you modify settings in the **MATLAB expression** or **Publish setting** pane, MATLAB automatically saves the changes.

Reassociating and Renaming Publish Configurations

Each publish configuration is associated with a specific file. If you move or rename a file, redefine its association. If you delete a file, consider deleting the associated configurations, or associating them with a different file.

When MATLAB cannot associate a configuration with a file, the Edit Configurations dialog box displays the file name in red and a **File Not Found** message. To reassociate a configuration with another file, perform the following steps.

- 1** Click the Clear search button  on the left pane of the Edit Configurations dialog box.
- 2** Select the file for which you want to reassociate publish configurations.
- 3** In the right pane of the Edit Configurations dialog box, click **Choose....** In the Open dialog box, navigate to and select the file with which you want to reassociate the configurations.

You can rename the configurations at any time by selecting a configuration from the list in the left pane. In the right pane, edit the value for the **Publish configuration name**.

Note To run correctly after a file name change, you might need to change the code statements in the **MATLAB expression** pane. For example, change a function call to reflect the new file name for that function.

Using Publish Configurations Across Different Systems

Each time you create or save a publish configuration using the Edit Configurations dialog box, the Editor updates the `publish_configurations.m` file in your preferences folder. (This is the folder that MATLAB returns when you run the MATLAB `prefdir` function.)

Although you can port this file from the preferences folder on one system to another, only one `publish_configurations.m` file can exist on a system. Therefore, only move the file to another system if you have not created any publish configurations on the second system. In addition, because the `publish_configurations.m` file might contain references to file paths, be sure that the specified files and paths exist on the second system.

MathWorks recommends that you not update `publish_configurations.m` in the MATLAB Editor or a text editor. Changes that you make using tools other than the Edit Configurations dialog box might be overwritten later.

See Also

More About

- “Options for Presenting Your Code” on page 23-2
- “Publishing MATLAB Code” on page 23-4
- “Publishing Markup” on page 23-7

Coding and Productivity Tips

- “Open and Save Files in the Editor” on page 24-2
- “Check Code for Errors and Warnings” on page 24-6
- “Improve Code Readability” on page 24-20
- “Find and Replace Text in Files” on page 24-27
- “Go To Location in File” on page 24-32
- “Display Two Parts of a File Simultaneously” on page 24-38
- “Add Reminders to Files” on page 24-41
- “MATLAB Code Analyzer Report” on page 24-44
- “MATLAB Code Compatibility Report” on page 24-48

Open and Save Files in the Editor

In this section...

“Open Existing Files” on page 24-2

“Save Files” on page 24-3

Open Existing Files

To open an existing file or files in the Editor, choose the option that achieves your goals, as described in this table.

Goal	Steps	Additional Information
<p>Open with associated tool</p> <p>Open a file using the appropriate MATLAB tool for the file type.</p>	<p>On the Editor, Live Editor, or Home tab, in the File section, click .</p> <p>You also can double-click the file in the Current Folder browser.</p>	<p>For example, this option opens a file with a <code>.m</code> or <code>.mlx</code> extension in the Editor and loads a MAT-file into the Workspace browser.</p>
<p>Open as text file</p> <p>Open a file in the Editor as a text file, even if the file type is associated with another application or tool.</p>	<p>On the Editor tab, in the File section, click Open , and select Open as Text.</p>	<p>This is useful, for example, if you have imported a tab-delimited data file (<code>.dat</code>) into the workspace and you find you want to add a data point. Open the file as text in the Editor, make your addition, and then save the file.</p>
<p>Open function from within file</p> <p>Open a local function or function file from within a file in the Editor.</p>	<p>Position the cursor on the name within the open file, and then right-click and select Open file-name from the context menu.</p>	<p>You also can use this method to open a variable or Simulink model.</p> <p>For details, see “Open a File or Variable from Within a File” on page 24-36.</p>

Goal	Steps	Additional Information
<p>Reopen file</p> <p>Reopen a recently used file.</p>	<p>At the bottom of the Open ▾ drop-down list, select a file under Recent Files.</p>	<p>To change the number of files on the list, click  Preferences, and then select MATLAB and Editor/Debugger. Under Most recently used file list, change the value for Number of entries.</p>
<p>Reopen files at startup</p> <p>At startup, automatically open the files that were open when the previous MATLAB session ended.</p>	<p>On the Home tab, in the Environment section, click  Preferences and select MATLAB and Editor/Debugger. Then, select On restart reopen files from previous MATLAB session.</p>	
<p>Open file displaying in another tool</p> <p>Open a file name displaying in another MATLAB desktop tool or Microsoft tool.</p>	<p>Drag the file from the other tool into the Editor.</p>	<p>For example, drag files from the Current Folder browser or from Windows Explorer.</p>
<p>Open file using a function</p>	<p>Use the edit or open function.</p>	<p>For example, type the following to open <code>collatz.m</code>:</p> <pre>edit collatz.m</pre> <p>If <code>collatz.m</code> is not on the search path or in the current folder, use the relative or absolute path for the file.</p>

For special considerations on the Macintosh platform, see “Navigating Within the MATLAB Root Folder on Macintosh Platforms”.

Some options for opening files are not available in MATLAB Online.

Save Files

After you modify a file in the Editor, an asterisk (*) follows the file name. This asterisk indicates that there are unsaved changes to the file.

You can perform four different types of save operations, which have various effects, as described in this table.

Save Option	Steps
Save file to disk and keep file open in the Editor.	On the Editor or Live Editor tab, in the File section, click  .
Rename file, save it to disk, and make it the active Editor document. Original file remains unchanged on disk.	<ol style="list-style-type: none"> 1 On the Editor or Live Editor tab, in the File section, click Save  and select Save As. 2 Specify a new name, type, or both for the file, and then click Save.
Save file to disk under new name. Original file remains open and unsaved.	<ol style="list-style-type: none"> 1 On the Editor tab, in the File section, click Save  and select Save Copy As. MATLAB opens the Select File for Backup dialog box. 2 Specify a name and type for the backup file, and then click Save.
Save changes to all open files using current file names. All files remain open.	On the Editor tab, in the File section, click Save  and select Save All .

Some options for saving files are not available in MATLAB Online.

Recommendations on Saving Files

MathWorks recommends that you save files you create and files from MathWorks that you edit to a folder that is not in the *matlabroot* folder tree, where *matlabroot* is the folder returned when you type `matlabroot` in the Command Window. If you keep your files in *matlabroot* folders, they can be overwritten when you install a new version of MATLAB software.

At the beginning of each MATLAB session, MATLAB loads and caches in memory the locations of files in the *matlabroot* folder tree. Therefore, if you:

- Save files to *matlabroot* folders using an external editor, run `rehash toolbox` before you use the files in the current session.
- Add or remove files from *matlabroot* folders using file system operations, run `rehash toolbox` before you use the files in the current session.
- Modify existing files in *matlabroot* folders using an external editor, run `clear function-name` before you use these files in the current session.

For more information, see `rehash` or “Toolbox Path Caching in MATLAB”.

Backing Up Files

When you modify a file in the Editor, the Editor saves a copy of the file using the same file name but with an `.asv` extension every 5 minutes. The backup version is useful if you have system problems and lose changes you made to your file. In that event, you can open the backup version, `filename.asv`, and then save it as `filename.m` to use the last good version of `filename`.

Note The Live Editor and MATLAB Online do not automatically save copies of files.

To select preferences, click  **Preferences**, and then select **MATLAB > Editor/Debugger > Backup Files** on the **Home** tab, in the **Environment** section. You can then:

- Turn the backup feature on or off.
- Automatically delete backup files when you close the corresponding source file.

By default, MATLAB automatically deletes backup files when you close the Editor. It is best to keep backup-to-file relationships clear and current. Therefore, when you rename or remove a file, consider deleting or renaming the corresponding backup file.

- Specify the number of minutes between backup saves.
- Specify the file extension for backup files.
- Specify a location for backup files

If you edit a file in a read-only folder and the back up **Location** preference is **Source file directories**, then the Editor does not create a backup copy of the file.

Check Code for Errors and Warnings

MATLAB Code Analyzer can automatically check your code for coding problems.

Automatically Check Code in the Editor — Code Analyzer

You can view warning and error messages about your code, and modify your file based on the messages. The messages update automatically and continuously so you can see if your changes addressed the issues noted in the messages. Some messages offer additional information, automatic code correction, or both.

Enable Continuous Code Checking

To enable continuous code checking in a MATLAB code file in the Editor:

- 1 On the **Home** tab, in the **Environment** section, click  **Preferences**.
- 2 Select **MATLAB > Code Analyzer**, and then select the **Enable integrated warning and error messages** check box.
- 3 Set the **Underlining** option to Underline warnings and errors, and then click **OK**.

Note Changing the continuous code checking preferences is not supported in MATLAB Online. Continuous code checking is always enabled.

Use Continuous Code Checking

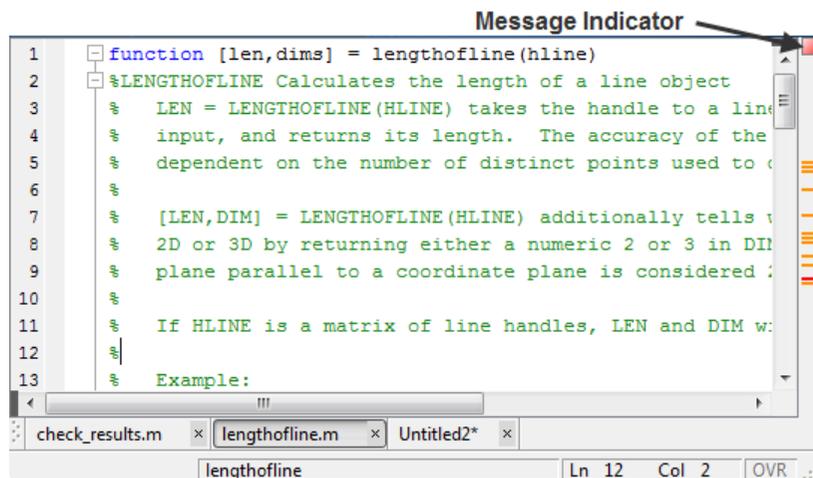
You can use continuous code checking in MATLAB code files in the Editor:

- 1 Open a MATLAB code file in the Editor. This example uses the sample file `lengthofline.m` that ships with the MATLAB software:
 - a Open the example file:

```
open(fullfile(matlabroot,'help','techdoc','matlab_env',...
    'examples','lengthofline.m'))
```
 - b Save the example file to a folder to which you have write access. For the example, `lengthofline.m` is saved to `C:\my_MATLAB_files`.

- 2 Examine the message indicator at the top of the message bar to see the Code Analyzer messages reported for the file:
 - **Red** indicates that syntax errors were detected. Another way to detect some of these errors is using syntax highlighting to identify unterminated character vectors, and delimiter matching to identify unmatched keywords, parentheses, braces, and brackets.
 - **Orange** indicates warnings or opportunities for improvement, but no errors, were detected.
 - **Green** indicates no errors, warnings, or opportunities for improvement were detected.

In this example, the indicator is red, meaning that there is at least one error in the file.



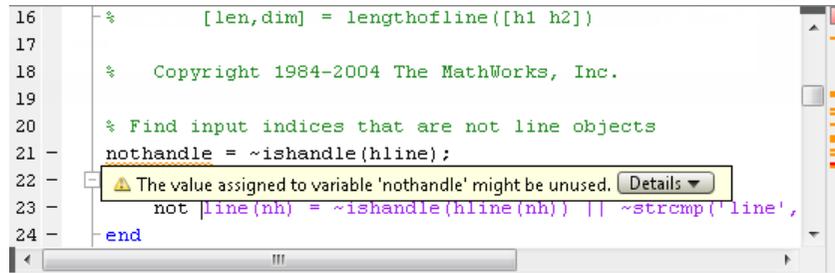
- 3 Click the message indicator to go to the next code fragment containing a message. The next code fragment is relative to the current cursor position, viewable in the status bar.

In the `lengthofline` example, the first message is at line 21. The cursor moves to the beginning of line 21.

The code fragment for which there is a message is underlined in either red for errors or orange for warnings and improvement opportunities.

- View the message by moving the mouse pointer within the underlined code fragment.

The message opens in a tooltip and contains a **Details** button that provides access to additional information by extending the message. Not all messages have additional information.



```

16 - %         [len,dim] = lengthofline([h1 h2])
17
18 - % Copyright 1984-2004 The MathWorks, Inc.
19
20 - % Find input indices that are not line objects
21 - nothandle = ~ishandle(hline);
22 -
23 - not |line(nh) = ~ishandle(hline(nh)) || ~strcmp('line',
24 - end
  
```

The tooltip for line 21 displays: **Warning** The value assigned to variable 'nothandle' might be unused. A **Details** button is visible in the tooltip.

- Click the **Details** button.

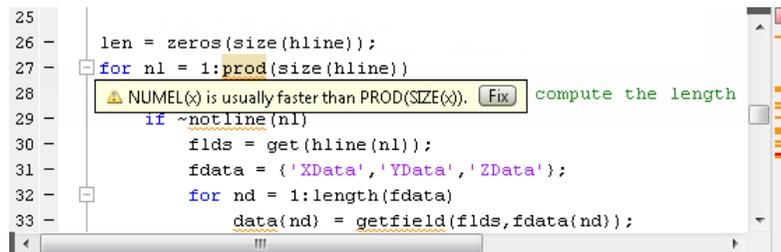
The window expands to display an explanation and user action.

- Modify your code, if needed.

The message indicator and underlining automatically update to reflect changes you make, even if you do not save the file.

- On line 27, hover over `prod`.

The code is underlined because there is a warning message, and it is highlighted because an automatic fix is available. When you view the message, it provides a button to apply the automatic fix.



```

25 -
26 - len = zeros(size(hline));
27 - for nl = 1:prod(size(hline))
28 -
29 -     if ~notline(nl)
30 -         flds = get(hline(nl));
31 -         fdata = {'XData','YData','ZData'};
32 -         for nd = 1:length(fdata)
33 -             data(nd) = getfield(flds,fdata(nd));
  
```

The tooltip for line 27 displays: **Warning** NUMEL(x) is usually faster than PROD(SIZE(x)). A **Fix** button is visible in the tooltip.

- Fix the problem by doing one of the following:

- If you know what the fix is (from previous experience), click **Fix**.
- If you are unfamiliar with the fix, view, and then apply it as follows:

- a Right-click the highlighted code (for a single-button mouse, press **Ctrl**+click), and then view the first item in the context menu.
- b Click the fix.

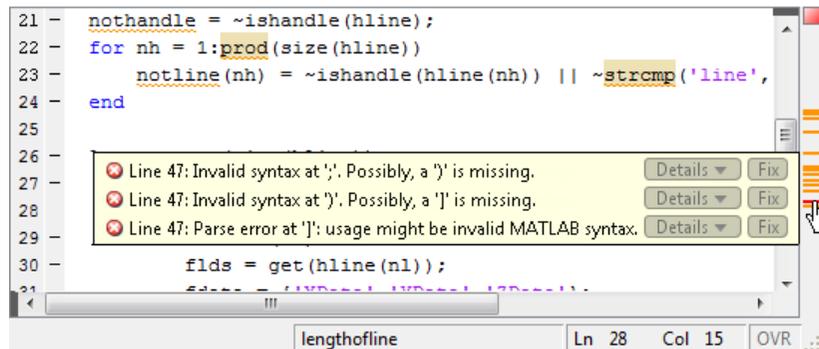
MATLAB automatically corrects the code.

In this example, MATLAB replaces `prod(size(hline))` with `numel(hline)`.

9 Go to a different message by doing one of the following:

- To go to the next message, click the message indicator or the next underlined code fragment.
- To go to a line that a marker represents, click a red or orange line in the indicator bar.

To see the first error in `lengthofline`, click the first red marker in the message bar. The cursor moves to the first suspect code fragment in line 47. The **Details** and **Fix** buttons are dimmed (or not visible if in MATLAB Online), indicating that there is no more information about this message and there is no automatic fix.



Multiple messages can represent a single problem or multiple problems. Addressing one might address all of them, or after addressing one, the other messages might change or what you need to do might become clearer.

10 Modify the code to address the problem noted in the message—the message indicators update automatically.

On line 47, the message suggests a delimiter imbalance. To investigate this message, in the Editor, move the arrow key over each of the delimiters to see if MATLAB

indicates a mismatch. For instructions on how to enable delimiter matching on arrow, see “Set Keyboard Preferences”.

It might appear that there are no mismatched delimiters. However, code analysis detects the semicolon in parentheses: `data{3} (;)`, and interprets it as the end of a statement. The message reports that the two statements on line 47 each have a delimiter imbalance.

To fix the problem, in line 47, change `data{3} (;)` to `data{3} (:)`. Now, the underline no longer appears in line 47. The single change addresses the issues in both of the messages for line 47. Because the change removed the only error in the file, the message indicator at the top of the bar changes from red to orange, indicating that only warnings and potential improvements remain.

After modifying the code to address all the messages, or disabling designated messages, the message indicator becomes green. The example file with all messages addressed has been saved as `lengthofline2.m`. Open the corrected example file with the command:

```
open(fullfile(matlabroot, 'help', 'techdoc', ...  
            'matlab_env', 'examples', 'lengthofline2.m'))
```

Create a Code Analyzer Message Report

You can create a report of messages for an individual file, or for all files in a folder using one of these methods:

- Run a report for an individual MATLAB code file:
 - 1 On the Editor window, click  and select **Show Code Analyzer Report**.
A Code Analyzer Report appears in the MATLAB Web Browser.
 - 2 Modify your file based on the messages in the report.
 - 3 Save the file.
 - 4 Rerun the report to see if your changes addressed the issues noted in the messages.
- Run a report for all files in a folder:
 - 1 On the Current Folder browser, click .
 - 2 Select **Reports > Code Analyzer Report**.

- 3 Modify your files based on the messages in the report.

For details, see “MATLAB Code Analyzer Report” on page 24-44.

- 4 Save the modified file(s).
- 5 Rerun the report to see if your changes addressed the issues noted in the messages.

Adjust Code Analyzer Message Indicators and Messages

Depending on the stage at which you are in completing a MATLAB file, you might want to restrict the code underlining. You can do this by using the Code Analyzer preference referred to in step 1, in “Check Code for Errors and Warnings” on page 24-6. For example, when first coding, you might prefer to underline only errors because warnings would be distracting.

Code analysis does not provide perfect information about every situation and sometimes, you might not want to change the code based on a message. If you do not want to change the code, and you do not want to see the indicator and message for that line, suppress them. For the `lengthofline` example, in line 49, the first message is `Terminate statement with semicolon to suppress output (in functions)`. Adding a semicolon to the end of a statement suppresses output and is a common practice. Code analysis alerts you to lines that produce output, but lack the terminating semicolon. If you want to view output from line 49, do not add the semicolon as the message suggests.

There are a few different ways to suppress (turn off) the indicators for warning and error messages:

- “Suppress an Instance of a Message in the Current File” on page 24-12
- “Suppress All Instances of a Message in the Current File” on page 24-12
- “Suppress All Instances of a Message in All Files” on page 24-13
- “Save and Reuse Code Analyzer Message Settings” on page 24-13

You cannot suppress error messages such as syntax errors. Therefore, instructions on suppressing messages do not apply to those types of messages.

Note Changing the Code Analyzer Message preferences is not supported in MATLAB Online. All Code Analyzer messages are always enabled.

Suppress an Instance of a Message in the Current File

You can suppress a specific instance of a Code Analyzer message in the current file. For example, using the code presented in “Check Code for Errors and Warnings” on page 24-6, follow these steps:

- 1 In line 49, right-click at the first underline (for a single-button mouse, press **Ctrl** +click).
- 2 From the context menu, select **Suppress 'Terminate statement with semicolon...'** > **On This Line**.

The comment `%#ok<NOPRT>` appears at the end of the line, which instructs MATLAB not to check for a terminating semicolon at that line. The underline and mark in the indicator bar for that message disappear.

- 3 If there are two messages on a line that you do not want to display, right-click separately at each underline and select the appropriate entry from the context menu.

The `%#ok` syntax expands. For the example, in the code presented in “Check Code for Errors and Warnings” on page 24-6, ignoring both messages for line 49 adds `%#ok<NBRAK, NOPRT>`.

Even if Code Analyzer preferences are set to enable this message, the specific instance of the message suppressed in this way does not appear because the `%#ok` takes precedence over the preference setting. If you later decide you want to check for a terminating semicolon at that line, delete `%#ok<NOPRT>` from the line.

Suppress All Instances of a Message in the Current File

You can suppress all instances of a specific Code Analyzer message in the current file. For example, using the code presented in “Check Code for Errors and Warnings” on page 24-6, follow these steps:

- 1 In line 49, right-click at the first underline (for a single-button mouse, press **Ctrl** +click).
- 2 From the context menu, select **Suppress 'Terminate statement with semicolon...'** > **In This File**.

The comment `%#ok< *NOPRT>` appears at the end of the line, which instructs MATLAB not to check for a terminating semicolon throughout the file. All underlines and marks in the message indicator bar that correspond to this message disappear.

If there are two messages on a line that you do not want to display anywhere in the current file, right-click separately at each underline, and then select the appropriate entry from the context menu. The `%#ok` syntax expands. For the example, in the code presented in “Check Code for Errors and Warnings” on page 24-6, ignoring both messages for line 49 adds `%#ok<*NBRAK, *NOPRT>`.

Even if Code Analyzer preferences are set to enable this message, the message does not appear because the `%#ok` takes precedence over the preference setting. If you later decide you want to check for a terminating semicolon in the file, delete `%#ok<*NOPRT>` from the line.

Suppress All Instances of a Message in All Files

You can disable all instances of a Code Analyzer message in all files. For example, using the code presented in “Check Code for Errors and Warnings” on page 24-6, follow these steps:

- 1 In line 49, right-click at the first underline (for a single-button mouse, press **Ctrl** +click).
- 2 From the context menu, select **Suppress 'Terminate statement with semicolon...'** > **In All Files**.

This modifies the Code Analyzer preference setting.

If you know which message or messages you want to suppress, you can disable them directly using Code Analyzer preferences, as follows:

- 1 On the **Home** tab, in the **Environment** section, click  **Preferences**.
- 2 Select **MATLAB > Code Analyzer**.
- 3 Search the messages to find the ones you want to suppress.
- 4 Clear the check box associated with each message you want to suppress in all files.
- 5 Click **OK**.

Save and Reuse Code Analyzer Message Settings

You can specify that you want certain Code Analyzer messages enabled or disabled, and then save those settings to a file. When you want to use a settings file with a particular file, you select it from the Code Analyzer preferences pane. That setting file remains in effect until you select another settings file. Typically, you change the settings file when you have a subset of files for which you want to use a particular settings file.

Follow these steps:

- 1 On the **Home** tab, in the **Environment** section, click  **Preferences**.
The Preferences dialog box opens.
- 2 Select **MATLAB > Code Analyzer**.
- 3 Enable or disable specific messages, or categories of messages.
- 4 Click the Actions button , select **Save as**, and then save the settings to a `txt` file.
- 5 Click **OK**.

You can reuse these settings for any MATLAB file, or provide the settings file to another user.

To use the saved settings:

- 1 On the **Home** tab, in the **Environment** section, click  **Preferences**.
The Preferences dialog box opens.
- 2 Select **MATLAB > Code Analyzer**.
- 3 Use the **Active Settings** drop-down list to select **Browse...**.
The Open dialog box appears.
- 4 Choose from any of your settings files.

The settings you choose are in effect for all MATLAB files until you select another set of Code Analyzer settings.

Understand Code Containing Suppressed Messages

If you receive code that contains suppressed messages, you might want to review those messages without the need to unsuppress them first. A message might be in a suppressed state for any of the following reasons:

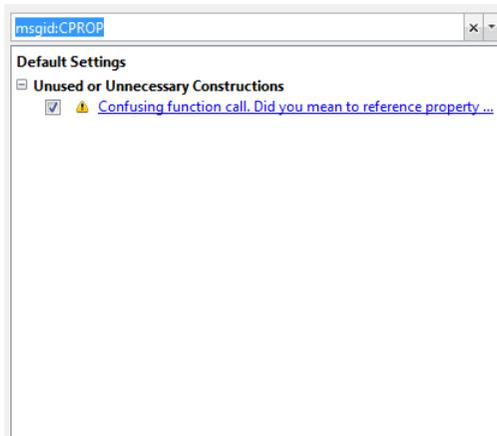
- One or more `%#ok<message - ID>` directives are on a line of code that elicits a message specified by `<message - ID>`.
- One or more `%#ok<*message - ID>` directives are in a file that elicits a message specified by `<message - ID>`.

- It is cleared in the Code Analyzer preferences pane.
- It is disabled by default.

To determine the reasons why some messages are suppressed:

- 1 Search the file for the `%#ok` directive and create a list of all the message IDs associated with that directive.
- 2 On the **Home** tab, in the **Environment** section, click  **Preferences**.
The Preferences dialog box opens.
- 3 Select **MATLAB > Code Analyzer**.
- 4 In the search field, type `msgid:` followed by one of the message IDs, if any, you found in step 1.

The message list now contains only the message that corresponds to that ID. If the message is a hyperlink, click it to see an explanation and suggested action for the message. This can provide insight into why the message is suppressed or disabled. The following image shows how the Preferences dialog box appears when you enter `msgid:CPROP` in the search field.



- 5 Click the  button to clear the search field, and then repeat step 4 for each message ID you found in step 1.
- 6 Display messages that are disabled by default and disabled in the Preferences pane by clicking the down arrow to the right of the search field. Then, click **Show Disabled Messages**.

- 7 Review the message associated with each message ID to understand why it is suppressed in the code or disabled in Preferences.

Understand the Limitations of Code Analysis

Code analysis is a valuable tool, but there are some limitations:

- Sometimes, it fails to produce Code Analyzer messages where you expect them.

By design, code analysis attempts to minimize the number of incorrect messages it returns, even if this behavior allows some issues to go undetected.

- Sometimes, it produces messages that do not apply to your situation.

When provided with message, click the **Detail** button for additional information, which can help you to make this determination. Error messages are almost always problems. However, many warnings are suggestions to look at something in the code that is unusual and therefore suspect, but might be correct in your case.

Suppress a warning message if you are certain that the message does not apply to your situation. If your reason for suppressing a message is subtle or obscure, include a comment giving the rationale. That way, those who read your code are aware of the situation.

For details, see “Adjust Code Analyzer Message Indicators and Messages” on page 24-11.

These sections describe code analysis limitations regarding the following:

- “Distinguish Function Names from Variable Names” on page 24-16
- “Distinguish Structures from Handle Objects” on page 24-17
- “Distinguish Built-In Functions from Overloaded Functions” on page 24-18
- “Determine the Size or Shape of Variables” on page 24-18
- “Analyze Class Definitions with Superclasses” on page 24-18
- “Analyze Class Methods” on page 24-18

Distinguish Function Names from Variable Names

Code analysis cannot always distinguish function names from variable names. For the following code, if the Code Analyzer message is enabled, code analysis returns the

message, Code Analyzer cannot determine whether `xyz` is a variable or a function, and assumes it is a function. Code analysis cannot make a determination because `xyz` has no obvious value assigned to it. However, the program might have placed the value in the workspace in a way that code analysis cannot detect.

```
function y=foo(x)
    .
    .
    .
    y = xyz(x);
end
```

For example, in the following code, `xyz` can be a function, or can be a variable loaded from the MAT-file. Code analysis has no way of making a determination.

```
function y=foo(x)
    load abc.mat
    y = xyz(x);
end
```

Variables might also be undetected by code analysis when you use the `eval`, `evalc`, `evalin`, or `assignin` functions.

If code analysis mistakes a variable for a function, do one of the following:

- Initialize the variable so that code analysis does not treat it as a function.
- For the `load` function, specify the variable name explicitly in the `load` command line. For example:

```
function y=foo(x)
    load abc.mat xyz
    y = xyz(x);
end
```

Distinguish Structures from Handle Objects

Code analysis cannot always distinguish structures from handle objects. In the following code, if `x` is a structure, you might expect a Code Analyzer message indicating that the code never uses the updated value of the structure. If `x` is a handle object, however, then this code can be correct.

```
function foo(x)
    x.a = 3;
end
```

Code analysis cannot determine whether *x* is a structure or a handle object. To minimize the number of incorrect messages, code analysis returns no message for the previous code, even though it might contain a subtle and serious bug.

Distinguish Built-In Functions from Overloaded Functions

If some built-in functions are overloaded in a class or on the path, Code Analyzer messages might apply to the built-in function, but not to the overloaded function you are calling. In this case, suppress the message on the line where it appears or suppress it for the entire file.

For information on suppressing messages, see “Adjust Code Analyzer Message Indicators and Messages” on page 24-11.

Determine the Size or Shape of Variables

Code analysis has a limited ability to determine the type of variables and the shape of matrices. Code analysis might produce messages that are appropriate for the most common case, such as for vectors. However, these messages might be inappropriate for less common cases, such as for matrices.

Analyze Class Definitions with Superclasses

Code Analyzer has limited capabilities to check class definitions with superclasses. For example, Code Analyzer cannot always determine if the class is a handle class, but it can sometimes validate custom attributes used in a class if the attributes are inherited from a superclass. When analyzing class definitions, Code Analyzer tries to use information from the superclasses but often cannot get enough information to make a certain determination.

Analyze Class Methods

Most class methods must contain at least one argument that is an object of the same class as the method. But it does not always have to be the first argument. When it is, code analysis can determine that an argument is an object of the class you are defining, and it can do various checks. For example, it can check that the property and method names exist and are spelled correctly. However, when code analysis cannot determine that an object is an argument of the class you are defining, then it cannot provide these checks.

Enable MATLAB Compiler Deployment Messages

You can switch between showing or hiding Compiler deployment messages when you work on a file by changing the Code Analyzer preference for this message category. Your

choice likely depends on whether you are working on a file to be deployed. When you change the preference, it also changes the setting in the Editor. The converse is also true —when you change the setting from the Editor, it effectively changes this preference. However, if the dialog box is open at the time you modify the setting in the Editor, you will not see the changes reflected in the Preferences dialog box. Whether you change the setting from the Editor or from the Preferences dialog box, it applies to the Editor and to the Code Analyzer Report.

To enable MATLAB Compiler™ deployment messages:

- 1** On the **Home** tab, in the **Environment** section, click  **Preferences**.
The Preferences dialog box opens.
- 2** Select **MATLAB > Code Analyzer**.
- 3** Click the down arrow next to the search field, and then select **Show Messages in Category > MATLAB Compiler (Deployment) Messages**.
- 4** Click the **Enable Category** button.
- 5** Clear individual messages that you do not want to display for your code (if any).
- 6** Decide if you want to save these settings, so you can reuse them next time you work on a file to be deployed.

The settings txt file, which you can create as described in “Save and Reuse Code Analyzer Message Settings” on page 24-13, includes the status of this setting.

Improve Code Readability

In this section...
“Indenting Code” on page 24-20
“Right-Side Text Limit Indicator” on page 24-22
“Code Folding — Expand and Collapse Code Constructs” on page 24-22

Indenting Code

Indenting code makes reading statements such as `while` loops easier. To set and apply indenting preferences to code in the Editor:

- 1 On the **Home** tab, in the **Environment** section, click  **Preferences**.
The Preferences dialog box opens.
- 2 Select **MATLAB > Editor/Debugger > Language**.
- 3 Choose a computer language from the **Language** drop-down list.
- 4 In the **Indenting** section, select or clear **Apply smart indenting while typing**, depending on whether you want indenting applied automatically, as you type. Smart indenting is always applied automatically in the Live Editor.

If you clear this option, you can manually apply indenting by selecting the lines in the Editor to indent, right-clicking, and then selecting **Smart Indent** from the context menu.

- 5 Do one of the following:
 - If you chose any language other than **MATLAB** in step 2, click **OK**.
 - If you chose **MATLAB** in step 2, select a **Function indenting format**, and then click **OK**. Function indent formats are:
 - **Classic** — The Editor aligns the function code with the function declaration.
 - **Indent nested functions** — The Editor indents the function code within a nested function.
 - **Indent all functions** — The Editor indents the function code for both main and nested functions.

This image illustrates the function indenting formats.

```

1  % Indenting Preferences
2
3  % Classic
4  function classic_one
5 - disp('Main function code')
6     function classic_two
7 -     disp('Nested function code')
8 -     end
9 - end
10
11 % Indent Nested Functions
12 function nested_one
13 - disp('Main function code')
14     function nested_two
15 -     disp('Nested function code')
16 -     end
17 - end
18
19 % Indent All Functions
20 function all_one
21 -     disp('Main function code')
22     function all_two
23 -     disp('Nested function code')
24 -     end
25 - end
26 |

```

Note Indenting preferences are not supported for TLC, VHDL, or Verilog and are not available in MATLAB Online

Regardless of whether you apply indenting automatically or manually, you can move selected lines further to the left or right, by doing one of the following:

- On the **Editor** tab, in the **Edit** section, click , , or . For live scripts and functions, this functionality is available on the **Live Editor** tab, in the **Code** section.
- Pressing the **Tab** key or the **Shift+Tab** key, respectively.

This works differently if you select the Editor/Debugger **Tab** preference for **Emacs-style Tab key smart indenting**—when you position the cursor in any line or select a group of lines and press **Tab**, the lines indent according to smart indenting practices.

Right-Side Text Limit Indicator

By default, a light gray vertical line (rule) appears at column 75 in the Editor, indicating where a line exceeds 75 characters. You can set this text limit indicator to another value, which is useful, for example, if you want to view the code in another text editor that has a different line width limit. The right-side text limit indicator is not supported in the Live Editor or in MATLAB Online.

To hide, or change the appearance of the vertical line:

- 1 On the **Home** tab, in the **Environment** section, click  **Preferences**.
The Preferences dialog box opens.
- 2 Select **MATLAB > Editor/Debugger > Display**.
- 3 Adjust the settings in the **Right-hand text limit** section.

Note This limit is a visual cue only and does not prevent text from exceeding the limit. To wrap comment text at a specified column number automatically, adjust the settings in the **Comment formatting** section under **MATLAB > Editor/Debugger > Language** in the Preferences dialog box.

Code Folding — Expand and Collapse Code Constructs

Code folding is the ability to expand and collapse certain MATLAB programming constructs. This improves readability when a file contains numerous functions or other blocks of code that you want to hide when you are not currently working with that part of the file. MATLAB programming constructs include:

- Code sections for running and publishing code
- Class code
- `for` and `parfor` blocks
- Function and class help
- Function code

To see the entire list of constructs, select **Editor/Debugger > Code Folding** in the Preferences dialog box.

To expand or collapse code, click the plus \oplus or minus sign \ominus that appears to the left of the construct in the Editor. You can also press **Ctrl+Shift+.** (**period**) or **Ctrl+.** (**period**).

To expand or collapse all of the code in a file, place your cursor anywhere within the file, right-click, and then select **Code Folding > Expand All** or **Code Folding > Fold All** from the context menu. You can also press **Ctrl+Shift+=** or **Ctrl+=**.

Note Code folding is not supported in the Live Editor

View Folded Code in a Tooltip

You can view code that is currently folded by positioning the pointer over its ellipsis \dots . The code appears in a tooltip.

The following image shows the tooltip that appears when you place the pointer over the ellipsis on line 23 of `lengthoffline.m` when a for loop is folded.

The screenshot shows a MATLAB code editor window with a file named `lengthoffline.m`. The code is partially folded. Line 23, which contains a `for` loop, is collapsed and has an ellipsis \dots to its left. A tooltip is displayed over this ellipsis, showing the full code for the loop: `for nh = 1:prod(size(hline))
 notline(nh) = ~ishandle(hline(nh)) || ~strcmp('line', lower(get(hline(nh), 'type')));
end`. The rest of the code in the editor is visible, including comments and other function calls.

```

20
21 % Find input indices that are not line objects
22 nothandle = ~ishandle(hline);
23 for nh = 1:prod(size(hline)) ...
26
27 len = zeros(size(hline));
28 for nl = 1:prod(size(hline))
29     for nh = 1:prod(size(hline))
30         notline(nh) = ~ishandle(hline(nh)) || ~strcmp('line', lower(get(hline(nh), 'type')));
31     end
32 end
33
34 % If some indices are not lines, fill the results with NaNs.
35 if any(notline(:))
36     warning('lengthoffline:FillWithNaNs', ...
37         '\n%s of non-line objects are being filled with %s.', ...
38         'Lengths', 'NaNs', 'Dimensions', 'NaNs')
39 end
40 len(notline) = NaN;

```

Print Files with Collapsed Code

If you print a file with one or more collapsed constructs, those constructs are expanded in the printed version of the file.

Code Folding Behavior for Functions that Have No Explicit End Statement

If you enable code folding for functions and a function in your code does not end with an explicit end statement, you see the following behavior:

- If a line containing only comments appears at the end of such a function, then the Editor does not include that line when folding the function. MATLAB does not include

trailing white space and comments in a function definition that has no explicit end statement.

“Code Folding Enabled for Function Code Only” on page 24-24 illustrates this behavior. Line 13 is excluded from the fold for the `foo` function.

- If a fold for a code section overlaps the function code, then the Editor does not show the fold for the overlapping section.

The three figures that follow illustrate this behavior. The first two figures, “Code Folding Enabled for Function Code Only” on page 24-24 and “Code Folding Enabled for Sections Only” on page 24-25 illustrate how the code folding appears when you enable it for function code only and then section only, respectively. The last figure, “Code Folding Enabled for Both Functions and Sections” on page 24-26, illustrates the effects when code folding is enabled for both. Because the fold for section 3 (lines 11-13) overlaps the fold for function `foo` (lines 4-12), the Editor does not display the fold for section 3.

```
1  function main
2  disp function main | % end of function main
3
4  function foo
5  %% section 1
6  disp section 1
7  % end of cell 1
8  %% section 2
9  disp section2
10 % end of cell 2
11 %% section 3
12 disp section 3      % end of function foo
13 % end of cell 3
14
15 function bar
16 disp 'function bar' % end of function bar
17
```

Code Folding Enabled for Function Code Only

```
1 function main
2 disp function main % end of function main
3
4 function foo
5 %% section 1
6 disp section 1
7 % end of cell 1
8 %% section 2
9 disp section2
10 % end of cell 2
11 %% section 3
12 disp section 3 % end of function foo
13 % end of cell 3
14
15 function bar
16 disp 'function bar' % end of function bar
17
```

Code Folding Enabled for Sections Only

```
1  [-] function main
2      disp function main % end of function main
3
4  [-] function foo
5      [-] %% section 1
6          disp section 1
7          % end of cell 1
8      [-] %% section 2
9          disp section2
10         % end of cell 2
11     [-] %% section 3
12         disp section 3      % end of function foo
13         % end of cell 3
14
15 [-] function bar
16     disp 'function bar' % end of function bar
17
```

Code Folding Enabled for Both Functions and Sections

Find and Replace Text in Files

In this section...

“Find Any Text in the Current File” on page 24-27

“Find and Replace Functions or Variables in the Current File” on page 24-27

“Automatically Rename All Functions or Variables in a File” on page 24-29

“Find and Replace Any Text” on page 24-31

“Find Text in Multiple File Names or Files” on page 24-31

“Function Alternative for Finding Text” on page 24-31

“Perform an Incremental Search in the Editor” on page 24-31

Find Any Text in the Current File

You can search for text in your files using the Find & Replace tool.

- 1 Within the current file, select the text you want to find.
- 2 On the **Editor** or **Live Editor** tab, in the **Navigate** section, click  Find ⇩, and then select **Find...**

A Find & Replace dialog box opens.

- 3 Click **Find Next** to continue finding more occurrences of the text.

To find the previous occurrence of selected text (find backwards) in the current file, click **Find Previous** on the Find & Replace dialog box.

Find and Replace Functions or Variables in the Current File

To search for references to a particular function or variable, use the automatic highlighting feature for variables and functions. This feature is more efficient than using the text finding tools. Function and variable highlighting indicates only references to a particular function or variable, not other occurrences. For instance, it does not find instances of the function or variable name in comments. Furthermore, variable highlighting only includes references to the *same* variable. That is, if two variables use the same name, but are in different scopes on page 20-11, highlighting one does not cause the other to highlight.

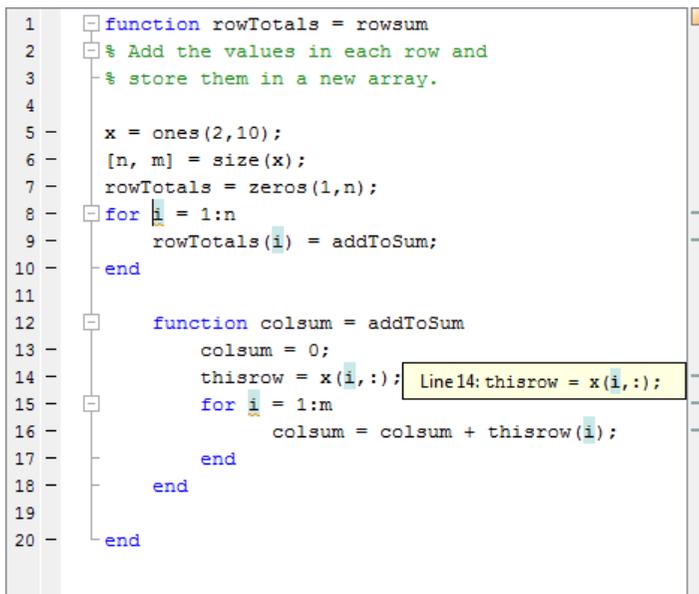
Find references to a function or variable using automatic highlighting by following these steps:

- 1 In a file open in the Editor, click an instance of the variable you want to find throughout the file. MATLAB indicates all occurrences of that variable within the file by:
 - Highlighting them in teal blue (by default) throughout the file
 - Adding a marker for each in the indicator bar

If a code analyzer indicator and a variable indicator appear on the same line in a file, the marker for the variable takes precedence.

- 2 Hover over a marker in the indicator bar to see the line it represents.
- 3 Click a marker in the indicator bar to navigate to that occurrence of the variable.
- 4 Replace an instance of a function or variable by editing the occurrence at a line to which you have navigated.

This image shows an example of how the Editor looks with variable highlighting enabled. In the image, the variable `i` appears highlighted in sky blue, and the indicator bar contains three variable markers.



The screenshot shows the MATLAB Editor interface with a code file open. The code is as follows:

```
1 function rowTotals = rowsum
2 % Add the values in each row and
3 % store them in a new array.
4
5 x = ones(2,10);
6 [n, m] = size(x);
7 rowTotals = zeros(1,n);
8 for i = 1:n
9     rowTotals(i) = addToSum;
10 end
11
12 function colsum = addToSum
13     colsum = 0;
14     thisrow = x(i,:);
15     for i = 1:m
16         colsum = colsum + thisrow(i);
17     end
18 end
19
20 end
```

The variable `i` is highlighted in sky blue in the code. In the indicator bar on the right, there are three markers: one for the `for` loop on line 8, one for the `addToSum` function call on line 9, and one for the `addToSum` function definition on line 12. A tooltip is visible over the marker on line 14, displaying the text "Line14: thisrow = x(i,:);".

To disable automatic highlighting, go to the **Home** tab and in the **Environment** section, click  **Preferences**. In **MATLAB > Colors > Programming Tools**, clear the **Automatically highlight** option.

Automatically Rename All Functions or Variables in a File

To help prevent typographical errors, MATLAB provides a feature that helps rename multiple references to a function or variable within a file when you manually change any of the following:

Function or Variable Renamed	Example
Function name in a function declaration	Rename foo in: <code>function foo(m)</code>
Input or output variable name in a function declaration	Rename y or m in: <code>function y = foo(m)</code>
Variable name on the left side of assignment statement	Rename y in: <code>y = 1</code>

As you rename such a function or variable, a tooltip opens if there is more than one reference to that variable or function in the file. The tooltip indicates that MATLAB will rename all instances of the function or variable in the file when you press **Shift + Enter**.

```

1  function rowTotals = rowsum
2  % Add the values in each row and
3  % store them in a new array.
4
5  x = ones(2,10);
6  [n, m] = size(x);
7  rowTotals = zeros(1,n);
8  for j = 1:n
9      Press Shift+Enter to rename 5 instances of "i" to "j"
10     end
11
12     function colsum = addToSum
13         colsum = 0;
14         thisrow = x(i,:);
15         for i = 1:m
16             colsum = colsum + thisrow(i);
17         end
18     end
19
20 end

```

Typically, multiple references to a function appear when you use nested functions or local functions.

Note MATLAB does *not* prompt you when you change:

- The name of a global variable.
 - The function input and output arguments, `varargin` and `varargout`.
-

To undo automatic name changes, click  once.

Automatic variable and function renaming is enabled by default. To disable it:

- 1 On the **Home** tab, in the **Environment** section, click  **Preferences**.
- 2 Select **MATLAB > Editor/Debugger > Language**.
- 3 In the **Language** field, select **MATLAB**.
- 4 Clear **Enable automatic variable and function renaming**.

Changing the automatic variable and function renaming preference is not supported in the Live Editor or in MATLAB Online.

Find and Replace Any Text

You can search for, and optionally replace specified text within a file. On the **Editor** or **Live Editor** tab, in the **Navigate** section, click  **Find** to open and use the Find & Replace dialog box.

Find Text in Multiple File Names or Files

You can find folders and file names that include specified text, or whose contents contain specified text. On the **Editor** or **Live Editor** tab, in the **File** section, click  **Find Files** to open the Find Files dialog box. For details, see “Find Files and Folders”.

Function Alternative for Finding Text

Use `lookfor` to search for the specified text in the first line of help for all files with the `.m` extension on the search path.

Perform an Incremental Search in the Editor

When you perform an incremental search, the cursor moves to the next or previous occurrence of the specified text in the current file. It is similar to the Emacs search feature. In the Editor, incremental search uses the same controls as incremental search in the Command Window. For details, see “Search Using Keyboard Shortcuts”.

Go To Location in File

In this section...

“Navigate to a Specific Location” on page 24-32

“Set Bookmarks” on page 24-35

“Navigate Backward and Forward in Files” on page 24-35

“Open a File or Variable from Within a File” on page 24-36

Navigate to a Specific Location

This table summarizes the steps for navigating to a specific location within a file open in the Editor and Live Editor.

Go To	Steps	Notes
Line Number	<ol style="list-style-type: none"> 1 On the Editor or Live Editor tab, in the Navigate section, click  Go To ▾ 2 Select Go to Line... 3 Specify the line to which you want to navigate. 	None
Function definition	<ol style="list-style-type: none"> 1 On the Editor or Live Editor tab, in the Navigate section, click  Go To ▾. 2 Under the heading Function, select the local function or nested function to which you want to navigate. 	<p>Includes local functions and nested functions.</p> <p>For both class and function files, the functions list in alphabetical order—except that in function files, the name of the main function always appears at the top of the list.</p>

Go To	Steps	Notes
	<ol style="list-style-type: none"> <li data-bbox="372 296 857 401">1 In the Current Folder browser, click the name of the file open in the Editor. <li data-bbox="372 406 857 510">2 Click the up arrow  at the bottom of Current Folder browser to open the detail panel. <li data-bbox="372 515 857 704">3 In the detail panel, double-click the function icon  corresponding to the title of the function or local function to which you want to navigate. 	Functions list in order of appearance within your file.
Code Section	<ol style="list-style-type: none"> <li data-bbox="372 710 857 814">1 On the Editor or Live Editor tab, in the Navigate section, click  Go To . <li data-bbox="372 819 857 944">2 Under Sections, select the title of the code section to which you want to navigate. 	For more information, see “Divide Your File into Code Sections” on page 18-7.
	<ol style="list-style-type: none"> <li data-bbox="372 949 857 1053">1 In the Current Folder browser, click the name of the file that is open in the Editor. <li data-bbox="372 1058 857 1163">2 Click the up arrow  at the bottom of Current Folder browser to open the detail panel. <li data-bbox="372 1168 857 1312">3 In the detail panel, double-click the section icon  corresponding to the title of the section to which you want to navigate. 	

Go To	Steps	Notes
Property	<ol style="list-style-type: none"> 1 In the Current Folder browser, click the name of the file that is open in the Editor. 2 Click the up arrow  at the bottom of Current Folder browser to open the detail panel. 3 On the detail panel, double-click the property icon  corresponding to the name of the property to which you want to navigate. 	For more information, see “Ways to Use Properties”.
Method	<ol style="list-style-type: none"> 1 In the Current Folder browser, click the name of the file that is open in the Editor. 2 Click the up arrow  at the bottom of Current Folder browser to open the detail panel. 3 In the detail panel, double-click the icon  corresponding to the name of the method to which you want to navigate. 	For more information, see “Methods in Class Design”.
Bookmark	<ol style="list-style-type: none"> 1 On the Editor tab, in the Navigate section, click  Go To. 2 Under Bookmarks, select the bookmark to which you want to navigate. 	For information on setting and clearing bookmarks, see “Set Bookmarks” on page 24-35.

Note The Details Panel and Bookmarks are not available in the Live Editor or in MATLAB Online.

Set Bookmarks

You can set a bookmark at any line in a file in the Editor so you can quickly navigate to the bookmarked line. This is particularly useful in long files. For example, suppose while working on a line, you want to look at another part of the file, and then return. Set a bookmark at the current line, go to the other part of the file, and then use the bookmark to return.

Bookmarks are not available in the Live Editor or in MATLAB Online.

To set a bookmark:

- 1 Position the cursor anywhere on the line.
- 2 On the **Editor** tab, in the **Navigate** section, click  **Go To**.
- 3 Under **Bookmarks**, select **Set/Clear**

A bookmark icon  appears to the left of the line.

To clear a bookmark, position the cursor anywhere on the line. Click  **Go To** and select **Set/Clear** under **Bookmarks**.

MATLAB does not maintain bookmarks after you close a file.

Navigate Backward and Forward in Files

To access lines in a file in the same sequence that you previously navigated or edited them, use  and .

Backward and forward navigation is not supported in the Live Editor or in MATLAB Online.

Interrupting the Sequence of Go Back and Go Forward

The back and forward sequence is interrupted if you:

- 1 Click .
- 2 Click .

- 3 Edit a line or navigate to another line using the list of features described in “Navigate to a Specific Location” on page 24-32.

You can still go to the lines preceding the interruption point in the sequence, but you cannot go to any lines after that point. Any lines you edit or navigate to after interrupting the sequence are added to the sequence after the interruption point.

For example:

- 1 Open a file.
- 2 Edit line 2, line 4, and line 6.
- 3 Click  to return to line 4, and then to return to line 2.
- 4 Click  to return to lines 4 and 6.
- 5 Click  to return to line 1.
- 6 Edit at 3.

This interrupts the sequence. You can no longer use  to return to lines 4 and 6.

You can, however, click  to return to line 1.

Open a File or Variable from Within a File

You can open a function, file, variable, or Simulink model from within a file in the Editor. Position the cursor on the name, and then right-click and select **Open selection** from the context menu. Based on what the selection is, the Editor performs a different action, as described in this table.

Item	Action
Local function	Navigates to the local function within the current file, if that file is a MATLAB code file. If no function by that name exists in the current file, the Editor runs the open function on the selection, which opens the selection in the appropriate tool.
Text file	Opens in the Editor.
Figure file (.fig)	Opens in a figure window.
MATLAB variable that is in the current workspace	Opens in the Variables Editor.

Item	Action
Model	Opens in Simulink. Not available in MATLAB Online.
Other	If the selection is some other type, Open selection looks for a matching file in a private folder in the current folder and performs the appropriate action.

Display Two Parts of a File Simultaneously

You can simultaneously display two different parts of a file in the Editor by splitting the screen display, as shown in the image that follows. This feature makes it easy to compare different lines in a file or to copy and paste from one part of a file to another.

Displaying two parts of a file simultaneously is not supported in the Live Editor.

```

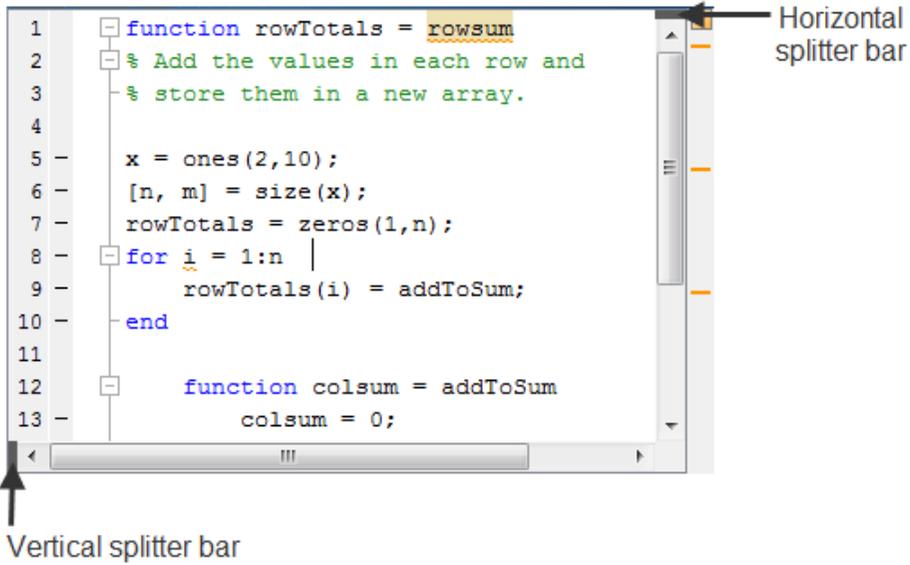
1  function rowTotals = rowsum
2  % Add the values in each row and
3  % store them in a new array.
4
5  x = ones(2,10);
6  [n, m] = size(x);
7  rowTotals = zeros(1,n);
8  for i = 1:n
9      rowTotals(i) = addToSum;
7  rowTotals = zeros(1,n);
8  for i = 1:n
9      rowTotals(i) = addToSum;
10 end
11
12 function colsum = addToSum
13     colsum = 0;
14     thisrow = x(i,:);
15     for i = 1:m

```

The following table describes the various ways you can split the Editor and manipulate the split-screen views. When you open a document, it opens unsplit, regardless of its split status it had when you closed it.

Operation	Instructions
Split the screen horizontally.	<p>Do either of the following:</p> <ul style="list-style-type: none"> Right-click and, select Split Screen > Top/Bottom from the Context Menu. If there is a vertical scroll bar, as shown in the illustration that follows, drag the splitter bar down.

Operation	Instructions
Split the screen vertically.	Do either of the following: <ul style="list-style-type: none">• From the Context Menu, select Split Screen > Left/Right.• If there is a horizontal scroll bar, as shown in the illustration that follows, drag the splitter bar from the left of the scroll bar.
Specify the active view.	Do either of the following: <ul style="list-style-type: none">• From the Context Menu, select Split Screen > Switch Focus.• Click in the view you want to make active. Updates you make to the document in the active view are also visible in the other view.
Remove the splitter	Do one of the following: <ul style="list-style-type: none">• Double-click the splitter.• From the Context Menu, Split Screen > Off.



See Also

More About

- "Document Layout"

Add Reminders to Files

Annotating a file makes it easier to find areas of your code that you intend to improve, complete, or update later. To annotate a file, add comments with the text **TODO**, **FIXME**, or any text of your choosing. After you annotate several files, run the **TODO/FIXME Report**, to identify all the MATLAB code files within a given folder that you have annotated.

This sample **TODO/FIXME Report** shows a file containing the text **TODO**, **FIXME**, and **NOTE**. The search is case insensitive.



Working with TODO/FIXME Reports

- 1 Use the Current Folder browser to navigate to the folder containing the files for which you want to produce a **TODO/FIXME** report.

Note You cannot run reports when the path is a UNC (Universal Naming Convention) path; that is, a path that starts with \\\. Instead, use an actual hard drive on your system, or a mapped network drive.

- 2 On the Current Folder browser, click , and then select **Reports > TODO/FIXME Report**.

The **TODO/FIXME Report** opens in the MATLAB Web Browser.

- 3 In the **TODO/FIXME Report** window, select one or more of the following to specify the lines that you want the report to include:
 - **TODO**
 - **FIXME**

- The text field check box

You can then enter any text in this field, including a regular expression on page 2-41. For example, you can enter NOTE, tbd, or re.*check.

Specifying custom text or which lines to search for is not supported in MATLAB Online.

- 4 Run the report on the files in the current folder, by clicking **Rerun This Report**.

The window refreshes and lists all lines in the MATLAB files within the specified folder that contain the text you selected in step 1. Matches are not case sensitive.

If you want to run the report on a folder other than the one currently specified in the report window, change the current folder. Then, click **Run Report on Current Folder**.

To open a file in the Editor at a specific line, click the line number in the report. Then you can change the file, as needed.

Suppose you have a file, `area.m`, in the current folder. The code for `area.m` appears in the image that follows.

```

1  function [output] = area(flag,radius)
2  % This function calculates the area of the entity
3  % flag = 1 for calculating the area of a
4  % flag = 2 for calculating the surface area of a sphere
5  % radius = radius of the entity
6
7  switch flag
8      % Modify the function to include the area of square
9      % and rectangle. (todo)
10
11     case 1
12         output = pi * radius^2;
13
14     case 2
15         output = 4 * pi * radius^2;
16         % Fixme: is the area of hemisphere as below?
17         % case 3
18         % output = 2 * pi * radius^2;
19         % fixme
20
21     otherwise
22         disp('Incorrect flag');
23         output = NaN;
24         % NOTE: Find out from manager if we need to include
25         % the area of a cone
26
27 end

```

When you run the TODO/FIXME report on the folder containing `area.m`, with the text TODO and FIXME selected and the text NOTE specified and selected, the report lists: 9 and rectangle. (todo)

14 Fixme: Is the area of hemisphere as below?
17 FIXME
21 NOTE: Find out from the manager if we need to include

<u>area</u>	<u>9</u> and rectangle. (todo) <u>14</u> Fixme: Is the area of hemisphere as below? <u>17</u> fixme <u>21</u> NOTE: Find out from the manager if we need to include
-------------	--

Notice the report includes the following:

- Line 9 as a match for the text **TODO**. The report includes lines that have the selected text regardless of its placement within a comment.
- Lines 14 and 17 as a match for the text **FIXME**. The report matches selected text in the file regardless of their casing.
- Line 21 as a match for the text **NOTE**. The report includes lines that have text as specified in the text field, assuming that you select the text field.

MATLAB Code Analyzer Report

In this section...
“Running the Code Analyzer Report” on page 24-44
“Changing Code Based on Code Analyzer Messages” on page 24-46
“Other Ways to Access Code Analyzer Messages” on page 24-47

Running the Code Analyzer Report

The Code Analyzer Report displays potential errors and problems, as well as opportunities for improvement in your code through messages. For example, a common message indicates that a variable `foo` might be unused.

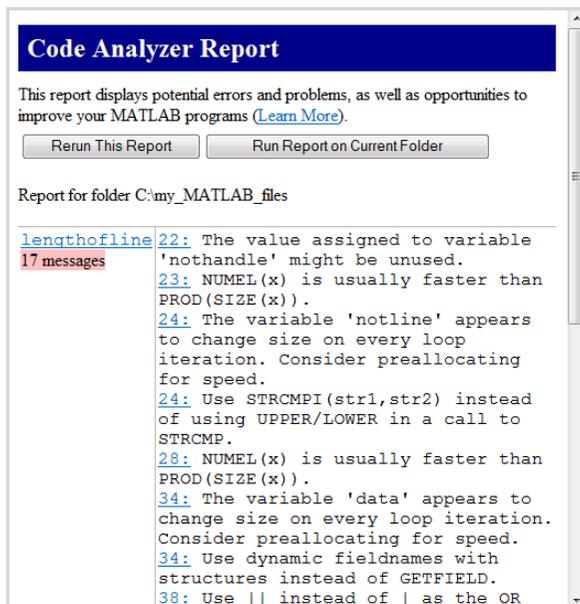
To run the Code Analyzer Report:

- 1 In the Current Folder browser, navigate to the folder that contains the files you want to check.

To use the `lengthofline.m` example shown in this documentation, save the file to the current folder, or to a folder for which you have write access. This example saves the file to the current folder, `C:\my_MATLAB_files`.

- ```
copyfile(fullfile(matlabroot,'help','techdoc','matlab_env','examples','lengthofline.m'))
```
- 2 In the Current Folder browser, click , and then select **Reports > Code Analyzer Report**.

The report displays in the MATLAB Web Browser, showing those files identified as having potential problems or opportunities for improvement.



- 3 For each message in the report, review the suggestion and your code. Click the line number to open the file in the Editor at that line, and change the file based on the message. Use the following general advice:
  - If you are unsure what a message means or what to change in the code, click the link in the message if one appears. For details, see “Check Code for Errors and Warnings” on page 24-6.
  - If the message does not contain a link, and you are unsure what a message means or what to do, search for related topics in the Help browser. For examples of messages and what to do about them, including specific changes to make for the example, `lengthofline.m`, see “Changing Code Based on Code Analyzer Messages” on page 24-46.
  - The messages do not provide perfect information about every situation and in some cases, you might not want to change anything based on the message. For details, see “Understand the Limitations of Code Analysis” on page 24-16.
  - If there are certain messages or types of messages you do not want to see, you can suppress them. For details, see “Adjust Code Analyzer Message Indicators and Messages” on page 24-11.
- 4 After modifying the file, save it. Consider saving the file to a different name if you made significant changes that might introduce errors. Then you can refer to the

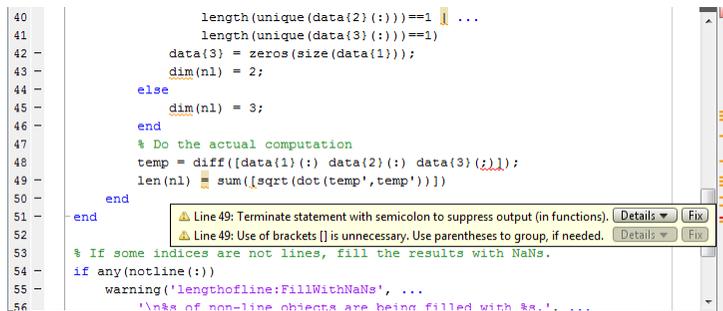
- original file, if needed, to resolve problems with the updated file. Use the  **Compare** button on the **Editor** or **Live Editor** tab to help you identify the changes you made to the file. For more information, see “Compare and Merge Text”.
- 5 Run and debug the file or files again to be sure that you have not introduced any inadvertent errors.
  - 6 If the report is displaying, click **Rerun This Report** to update the report based on the changes you made to the file. Ensure that the messages are gone, based on the changes you made to the files. To rerun the report in MATLAB Online, in the Current Folder browser, click , and then select **Reports > Code Analyzer Report**.

## Changing Code Based on Code Analyzer Messages

For information on how to correct the potential problems presented in Code Analyzer messages, use the following resources:

- Open the file in the Editor and click the **Details** button in the tooltip, as shown in the image following this list. An extended message opens. However, not all messages have extended messages.
- Use the Help browser **Search** pane to find documentation about terms presented in the messages.

The following image shows a tooltip with a **Details** button. The orange *line* under the equals (=) sign indicates a tooltip displays if you hover over the equals sign. The orange *highlighting* indicates that an automatic fix is available.



```

40 length(unique(data{2}(:)))==1 || ...
41 length(unique(data{3}(:)))==1)
42 data{3} = zeros(size(data{1}));
43 dim(n1) = 2;
44 else
45 dim(n1) = 3;
46 end
47 % Do the actual computation
48 temp = diff([data{1}(:) data{2}(:) data{3}(:)]);
49 len(n1) = sum([sqrt(dot(temp', temp))])
50 end
51 end
52
53 % If some indices are not lines, fill the results with NaNs.
54 if any(notline(:))
55 warning('lengthofline:FillWithNaNs', ...
56 '\n%a of non-line objects are being filled with %s.', ...

```

## Other Ways to Access Code Analyzer Messages

You can get Code Analyzer messages using any of the following methods. Each provides the same messages, but in a different format:

- Access the Code Analyzer Report for a file from the Profiler detail report.
- Run the `checkcode` function, which analyzes the specified file and displays messages in the Command Window.
- Run the `mlintrpt` function, which runs `checkcode` and displays the messages in the Web Browser.
- Use automatic code checking while you work on a file in the Editor. See “Automatically Check Code in the Editor — Code Analyzer” on page 24-6.

## MATLAB Code Compatibility Report

### Generate the Code Compatibility Report

The Code Compatibility Report displays potential compatibility issues in your code. This report helps update your code when you upgrade to a newer MATLAB release. For example, the report tells you when you use discouraged or removed functions or invalid syntaxes in your code. After you upgrade to a newer version of MATLAB, you can use this report to identify potential compatibility issues in your existing code.

To run the Code Compatibility Report:

- 1** In the Current Folder browser, navigate to the folder that contains the code files you want to analyze.
- 2** In the Current Folder browser, either click  or right-click the white space. Then select **Reports > Code Compatibility Report**.

The report displays in the MATLAB Web Browser, showing potential compatibility issues. For example:

Web Browser - (1 Error) Code Compatibility Report

(1 Error) Code Compatibility Report

Code Compatibility Report

Top 6 Syntax 1 Error 1 Warning 2 Improvements 380 Checks 3 Files Refresh

Analysis Date: 09-Jul-2018 09:43:40

MATLAB Version: R2018b

▼ Syntax Errors  
Update your code. Syntax errors result in nonrunnable code and impact compatibility analysis. Typically, syntax errors are not introduced with a new version of MATLAB.

| Occurrences | Filename                          |
|-------------|-----------------------------------|
| 3           | <a href="#">exampleScript.m</a>   |
| 3           | <a href="#">exampleFunction.m</a> |

▼ Incompatibility Errors  
Update your code to work in the current release. Incompatibility errors indicate that functionality has been removed.

| Row | Filename          | Line              | Description                                                                                                                     | Details     |
|-----|-------------------|-------------------|---------------------------------------------------------------------------------------------------------------------------------|-------------|
| 1   | anotherFunction.m | <a href="#">9</a> | 'dsp.DigitalFilter' has been removed. Use 'dsp.FIRFilter', 'dsp.BiquadFilter', 'dsp.IIRFilter', or 'dsp.AllpoleFilter' instead. | unavailable |

▼ Incompatibility Warnings  
Update your code now or in a later release. Incompatibility warnings indicate that functionality will be removed or changed in a future release.

| Row | Filename        | Line               | Description                                                                                   | Details                 |
|-----|-----------------|--------------------|-----------------------------------------------------------------------------------------------|-------------------------|
| 1   | exampleScript.m | <a href="#">29</a> | Nested functions will inherit import statements from the parent function in a future release. | <a href="#">Details</a> |

▼ Suggested Improvements  
Consider updating your code. It is expected to continue working but uses functionality that is not recommended.

| Row | Filename        | Line               | Description                                       | Details                 |
|-----|-----------------|--------------------|---------------------------------------------------|-------------------------|
| 1   | exampleScript.m | <a href="#">55</a> | 'strvcat' is not recommended. Use 'char' instead. | <a href="#">Details</a> |
| 2   | exampleScript.m | <a href="#">55</a> | 'strvcat' is not recommended. Use 'char' instead. | <a href="#">Details</a> |

► Checks Performed

► Files

- 3 Update your code to resolve the syntax errors for each file listed in the **Syntax Errors** section. Syntax errors result in code that does not run. While they are not introduced with a new version of MATLAB, syntax errors impact compatibility analysis. For example, *Parse error at '}' : usage might be invalid MATLAB syntax.*
- 4 For each incompatibility error, incompatibility warning, and suggested improvement in the report, review the issue description and your code. Messages include the line numbers to help location the issues in your code. To open the file in the Editor at that line, click the line number. Then change the file based on the message. If you are

unsure what a message means or what to change in the code, click the **Details** link associated with the message. Use the following general advice:

- **Incompatibility Errors**—Update your code to work in the current release. Incompatibility errors indicate that functionality has been removed. For example, *'wavfinfo' has been removed. Use 'AUDIOINFO' instead.*
- **Incompatibility Warnings**—Update your code now or in a later release. Incompatibility warnings indicate that functionality will be removed or changed in a future release. For example, *'aviinfo' will be removed in a future release. Use 'VideoReader' instead.*
- **Suggested Improvements**—Consider updating your code. The code is expected to continue working in future releases but uses functionality that is not recommended. For example, *Programmatic use of DISPLAY is not recommended. Use DISP or FPRINTF instead.*

The Code Compatibility Report also includes information about the checks performed on your code and the list of files that MATLAB analyzed for code compatibility.

## Programmatic Use

When you generate a Code Compatibility Report through the current folder browser, MATLAB analyzes code in the current working folder and subfolders. However, if you generate a report programmatically, you can specify particular files to analyze or to exclude subfolders from analysis. To generate a report programmatically, use one of the following methods.

- To generate a report that opens in the MATLAB® Web Browser programmatically, use the `codeCompatibilityReport` function.
- To return a `CodeCompatibilityAnalysis` object that contains the report information, use the `analyzeCodeCompatibility` function. You can then display a report for the stored object using the `codeCompatibilityReport` function.

## See Also

`CodeCompatibilityAnalysis` | `analyzeCodeCompatibility` | `codeCompatibilityReport`

# Programming Utilities

---

- “Identify Program Dependencies” on page 25-2
- “Protect Your Source Code” on page 25-7
- “Create Hyperlinks that Run Functions” on page 25-10
- “Create and Share Toolboxes” on page 25-13

## Identify Program Dependencies

If you need to know what other functions and scripts your program is dependent upon, use one of the techniques described below.

|                           |
|---------------------------|
| <b>In this section...</b> |
|---------------------------|

|                                                            |
|------------------------------------------------------------|
| “Simple Display of Program File Dependencies” on page 25-2 |
|------------------------------------------------------------|

|                                                              |
|--------------------------------------------------------------|
| “Detailed Display of Program File Dependencies” on page 25-2 |
|--------------------------------------------------------------|

|                                             |
|---------------------------------------------|
| “Dependencies Within a Folder” on page 25-3 |
|---------------------------------------------|

### 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 more detailed display of dependent function information, use the `matlab.codetools.requiredFilesAndProducts` function. In addition to program files, `matlab.codetools.requiredFilesAndProducts` shows which MathWorks products a particular function depends on. If you have a function, `myFun`, that calls to the edge function in the Image Processing Toolbox™:

```
[fList,pList] = matlab.codetools.requiredFilesAndProducts('myFun.m');
fList
```

```
fList =
 'C:\work\myFun.m'
```

The only required program file, is the function file itself, myFun.

```
{pList.Name}'
ans =
 'MATLAB'
 'Image Processing Toolbox'
```

The file, myFun.m, requires both MATLAB and the Image Processing Toolbox.

## Dependencies Within a Folder

The Dependency Report shows dependencies among MATLAB code files in a folder. Use this report to determine:

- Which files in the folder are required by other files in the folder
- If any files in the current folder will fail if you delete a file
- If any called files are missing from the current folder

The report does not list:

- Files in the `toolbox/matlab` folder because every MATLAB user has those files.

Therefore, if you use a function file that shadows a built-in function file, MATLAB excludes both files from the list.

- Files called from anonymous functions.
- The superclass for a class file.
- Files called from `eval`, `evalc`, `run`, `load`, function handles, and callbacks.

MATLAB does not resolve these files until run time, and therefore the Dependency Report cannot discover them.

- Some method files.

The Dependency Report finds class constructors that you call in a MATLAB file. However, any methods you execute on the resulting object are unknown to the report.

These methods can exist in the `classdef` file, as separate method files, or files belonging to superclass or superclasses of a method file.

To provide meaningful results, the Dependency Report requires the following:

- The search path when you run the report is the same as when you run the files in the folder. (That is, the current folder is at the top of the search path.)
- The files in the folder for which you are running the report do not change the search path or otherwise manipulate it.
- The files in the folder do not load variables, or otherwise create name clashes that result in different program elements with the same name.

---

**Note** Do not use the Dependency Report to determine which MATLAB code files someone else needs to run a particular file. Instead use the `matlab.codetools.requiredFilesAndProducts` function.

---

### Creating Dependency Reports

- 1 Use the Current Folder pane to navigate to the folder containing the files for which you want to produce a Dependency Report.

---

**Note** You cannot run reports when the path is a UNC (Universal Naming Convention) path; that is, a path that starts with `\\`. Instead, use an actual hard drive on your system, or a mapped network drive.

---

- 2 On the Current Folder pane, click , and then select **Reports > Dependency Report**.

The Dependency Report opens in the MATLAB Web Browser.

- 3 If you want, select one or more options within the report, as follows:

- To see a list of all MATLAB code files (children) called by each file in the folder (parent), select **Show child functions**.

The report indicates where each child function resides, for example, in a specified toolbox. If the report specifies that the location of a child function is unknown, it can be because:

- The child function is not on the search path.

- The child function is not in the current folder.
- The file was moved or deleted.
- To list the files that call each MATLAB code file, select **Show parent functions**.

The report limits the parent (calling) functions to functions in the current folder.

- To include local functions in the report, select **Show subfunctions**. The report lists local functions directly after the main function and highlights them in gray.

#### 4 Click **Run Report on Current Folder**.

### Reading and Working with Dependency Reports

The following image shows a Dependency Report. It indicates that `chirpy.m` calls two files in Signal Processing Toolbox™ and one in Image Processing Toolbox. It also shows that `go.m` calls `mobius.m`, which is in the current folder.

**Dependency Report**

The Dependency Report shows dependencies among MATLAB files in a folder ([Learn More](#)).

Rerun This Report    Run Report on Current Folder

Show child functions     Show parent functions (current folder only)  
 Show subfunctions

Built-in functions and files in toolbox/matlab are not shown

Report for Folder I:\my\_MATLAB\_files

| MATLAB File List                | Children<br>(called functions)                                                                                                                              |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">chirpy</a>          | toolbox : <a href="#">\images\images\erode.m</a><br>toolbox : <a href="#">\shared\siglib\chirp.m</a><br>toolbox : <a href="#">\signal\signal\specgram.m</a> |
| <a href="#">collatz</a>         |                                                                                                                                                             |
| <a href="#">collatzall</a>      | subfunction : <a href="#">collatzplot new</a>                                                                                                               |
| <a href="#">collatzplot</a>     | current dir : <a href="#">collatz</a>                                                                                                                       |
| <a href="#">collatzplot new</a> | current dir : <a href="#">collatz</a>                                                                                                                       |
| <a href="#">go</a>              | current dir : <a href="#">mobius</a>                                                                                                                        |
| <a href="#">mobius</a>          |                                                                                                                                                             |

The Dependency Report includes the following:

- MATLAB File List

The list of files in the folder on which you ran the Dependency Report. Click a link in this column to open the file in the Editor.

- Children

The function or functions called by the MATLAB file.

Click a link in this column to open the MATLAB file listed in the same row, and go to the first reference to the called function. For instance, suppose your Dependency Report appears as shown in the previous image. Clicking **\images\images\erode.m** opens `chirpy.m` and places the cursor at the first line that references `erode`. In other words, it does not open `erode.m`.

- Multiple class methods

Because the report is a static analysis, it cannot determine run-time data types and, therefore, cannot identify the particular class methods required by a file. If multiple class methods match a referenced method, the Dependency Report inserts a question mark link next to the file name. The question mark appears in the following image.

|                                       |         |   |                                                           |
|---------------------------------------|---------|---|-----------------------------------------------------------|
| <a href="#">rational differential</a> | toolbox | : | <a href="#">\rf\rf\s2sdd.m</a>                            |
|                                       | toolbox | : | <a href="#">\rf\rf\@rfckt\smith.m</a>                     |
|                                       | toolbox | : | <a href="#">\rf\rf\s2tf.m</a>                             |
|                                       | toolbox | : | <a href="#">\rf\rf\rationalfit.m</a>                      |
|                                       | toolbox | : | ? Multiple class methods match <a href="#">freqresp.m</a> |
|                                       | toolbox | : | <a href="#">\rf\rf\@rfmodel\timeresp.m</a>                |
|                                       | toolbox | : | <a href="#">\rf\rf\@rfckt\analyze.m</a>                   |
|                                       | toolbox | : | <a href="#">\signal\signal\fftfilt.m</a>                  |

Click the question mark link to list the class methods with the specified name that MATLAB might use. MATLAB lists *almost all* the method files on the search path that match the specified method file (in this case, `freqresp.m`). Do not be concerned if the list includes methods of classes and MATLAB built-in functions that are unfamiliar to you.

It is not necessary for you to determine which file MATLAB will use. MATLAB determines which method to use depending on the object that the program calls at run time.

## Protect Your Source Code

Although MATLAB source code (.m) 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 options instead:

- Deploy as P-code on page 25-7 — 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. When MATLAB P-codes a file, the file is *obfuscated* not *encrypted*. While the content in a .p file is difficult to understand, it should not be considered secure. It is not recommended that you P-code files to protect your intellectual property.

MATLAB does not support converting live scripts or live functions to P-code files.

- Compile into binary format on page 25-8 — Compile your source code files using the MATLAB Compiler to produce a standalone application. Distribute the latter to end users of your application.

### 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. P-code files are purposely obfuscated. They are not encrypted. While the content in a .p file is difficult to understand, it should not be considered secure. It is not recommended that you P-code files to protect your intellectual property.

---

**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. These older P-code files do not run in MATLAB 8.6 (R2015b) or later. 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 “Create Standalone Application from MATLAB” (MATLAB Compiler).

## Create Hyperlinks that Run Functions

The special keyword `matlab:` lets you embed commands in other functions. Most commonly, the functions that contain it display hyperlinks, which execute the commands when you click the hyperlink text. Functions that support `matlab:` syntax include `disp`, `error`, `fprintf`, `help`, and `warning`.

Use `matlab:` syntax to create a hyperlink in the Command Window that runs one or more functions. For example, you can use `disp` to display the word Hypotenuse as an executable hyperlink as follows:

```
disp(' Hypotenuse')
```

Clicking the hyperlink executes the three commands following `matlab:`, resulting in

```
c =
 5
```

Executing the link creates or redefines the variables `a`, `b`, and `c` in the base workspace.

The argument to `disp` is an `<a href>` HTML hyperlink. Include the full hypertext text, from `'<a href= to </a>'` within a single line, that is, do not continue long text on a new line. No spaces are allowed after the opening `<` and before the closing `>`. A single space is required between `a` and `href`.

You cannot directly execute `matlab:` syntax. That is, if you type

```
matlab:a=3; b=4;c=hypot(a,b)
```

you receive an error, because MATLAB interprets the colon as an array operator in an illegal context:

```
??? matlab:a=3; b=4;c=hypot(a,b)
 |
Error: The expression to the left of the equals sign
 is not a valid target for an assignment.
```

You do not need to use `matlab:` to display a live hyperlink to the Web. For example, if you want to link to an external Web page, you can use `disp`, as follows:

```
disp(' Hypotenuse')
```

The result in the Command Window looks the same as the previous example, but instead opens a page at [en.wikipedia.org](http://en.wikipedia.org):

Hypotenuse

Using `matlab:`, you can:

- “Run a Single Function” on page 25-11
- “Run Multiple Functions” on page 25-11
- “Provide Command Options” on page 25-12
- “Include Special Characters” on page 25-12

## Run a Single Function

Use `matlab:` to run a specified statement when you click a hyperlink in the Command Window. For example, run this command:

```
disp('Generate magic square')
```

It displays this link in the Command Window:

[Generate magic square](#)

When you click the link, MATLAB runs `magic(4)`.

## Run Multiple Functions

You can run multiple functions with a single link. For example, run this command:

```
disp('Plot x,y')
```

It displays this link in the Command Window:

[Plot x,y](#)

When you click the link, MATLAB runs this code:

```
x = 0:1:8;
y = sin(x);
plot(x,y)
```

Redefine `x` in the base workspace:

```
x = -2*pi:pi/16:2*pi;
```

Click the hyperlink, `Plot x,y` again and it changes the current value of `x` back to `0:1:8`. The code that `matlab:` runs when you click the `Plot x,y` defines `x` in the base workspace.

## Provide Command Options

Use multiple `matlab:` statements in a file to present options, such as

```
disp('Disable feature')
disp('Enable feature')
```

The Command Window displays the links that follow. Depending on which link you click, MATLAB sets `state` to `0` or `1`.

[Disable feature](#)  
[Enable feature](#)

## Include Special Characters

MATLAB correctly interprets most text that includes special characters, such as a greater than symbol (`>`). For example, the following statement includes a greater than symbol (`>`).

```
disp(' 0''">Positive')
```

and generates the following hyperlink.

[Positive](#)

Some symbols might not be interpreted correctly and you might need to use the ASCII value for the symbol. For example, an alternative way to run the previous statement is to use ASCII 62 instead of the greater than symbol:

```
disp('Positive')
```

# Create and Share Toolboxes

## In this section...

“Create Toolbox” on page 25-13

“Share Toolbox” on page 25-21

You can package MATLAB files to create a toolbox to share with others. These files can include MATLAB code, data, apps, examples, and documentation. When you create a toolbox, MATLAB generates a single installation file (.mltbx) that enables you or others to install your toolbox.

## Create Toolbox

To create a toolbox installation file:

- 1 In the **Environment** section of the **Home** tab, select  **Package Toolbox** from the **Add-Ons** menu.
- 2 In the Package a Toolbox dialog box, click the  button and select your toolbox folder. It is good practice to create the toolbox package from the folder level above your toolbox folder. The .mltbx toolbox file contains information about the path settings for your toolbox files and folders. By default, any of the included folders and files that are on your path when you create the toolbox appear on their paths after the end users install the toolbox.
- 3 In the dialog box, add the following information about your toolbox.

| Toolbox Information Field | Description                                                                                                                                          |
|---------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| Toolbox Name              | Enter the toolbox name, if necessary. By default, the toolbox name is the name of the toolbox folder. The Toolbox Name becomes the .mltbx file name. |
| Version                   | Enter the toolbox version number in the <i>Major.Minor.Bug.Build</i> format. <i>Bug</i> and <i>Build</i> are optional.                               |

| <b>Toolbox Information Field</b> | <b>Description</b>                                                                                                                                     |
|----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| Author Name, Email, and Company  | Enter contact information for the toolbox author. To save the contact information, click <b>Set as default contact</b> .                               |
| Toolbox Image                    | To select an image that represents your toolbox, click <b>Select toolbox image</b> .                                                                   |
| Summary and Description          | Enter the toolbox summary and description. It is good practice to keep the <b>Summary</b> text brief and to add detail to the <b>Description</b> text. |

- 4 To ensure MATLAB detects the expected components, review the toolbox contents. The following sections of the Package a Toolbox dialog box appear after you select a toolbox folder.

| <b>Package a Toolbox Dialog Box Section</b> | <b>Description</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Toolbox Files and Folders</b>            | <p>List of the folders and files contained in your toolbox. The listed files and folders are only those files that are located in the top level of the toolbox folder. You cannot navigate through the folders in the Toolbox Packaging dialog box.</p> <p>By default, if your toolbox contains a P-code file and a MATLAB code file (.m) with the same name in the same folder, MATLAB excludes the .m file from the toolbox. To include both the .p and .m files, clear the <b>Exclude MATLAB script or function files with matching P-files</b> option.</p> <p>To exclude other files or folders from the toolbox, register them in the text file that is displayed when you click <b>Exclude files and folders</b>. It is good practice to exclude any source control files related to your toolbox.</p> |

| <b>Package a Toolbox Dialog Box Section</b> | <b>Description</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>External Files</b>                       | List of the files required for your toolbox that are located outside the toolbox folder. By default, MATLAB includes the required files. You can choose to omit any files you do not want in your toolbox.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Install Actions</b>                      | <p>MATLAB Path — List of folders that are added to the user's MATLAB path when they install a toolbox. By default, the list includes any of the toolbox folders that are on your path when you create the toolbox. You can exclude folders from being added to the user's path by clearing them from the list. To manage the path for when a toolbox is installed, click <b>Manage the current MATLAB path</b>. To reset the list to the default list, click <b>Reset to the current MATLAB path</b>.</p> <p>Java Class Path — List of Java files that are added to the user's Java class path when they install a toolbox. Upon toolbox installation, the JAR files are added to the dynamic path for the duration of the MATLAB session. When the toolbox user restarts MATLAB, the JAR files are added to the static path.</p> |

| Package a<br>Toolbox<br>Dialog Box<br>Section | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                               | <p>Installation of Additional Software — List of additional software ZIP files that are installed on the user's system when they install a toolbox.</p> <p>Specify these fields:</p> <ul style="list-style-type: none"> <li>• <b>Display Name</b> — The name to display to the user when they install a toolbox.</li> <li>• <b>License URL</b> — The URL of the additional software license agreement to display to the user when they install a toolbox. The user is prompted to review and agree to the license agreement during installation. You must specify a valid URL to the license agreement.</li> <li>• <b>Download URL</b> — The URL to the ZIP file that contains the additional software. To specify different download URLs for different platforms, select a platform name from the drop-down menu to the left of the download URL. Then, click <b>Add Platform</b> to add a download URL for additional platforms.</li> </ul> <p>When the user installs a toolbox, MATLAB installs all additional software in the <code>addons\Toolboxes\AdditionalSoftware</code> folder, where <code>addons</code> is the add-ons default installation folder. For more information about the location of the add-ons default installation folder, see “Manage Your Add-Ons”.</p> <p>If your toolbox contains code that refers to the installation folder of the specified additional software, make these references portable to other computers. Replace the references with calls to the generated function <code>toolboxname\getInstallationLocation.mlx</code>, where <code>toolboxname</code> is the name of your toolbox. For example, if you are creating a toolbox called <code>mytoolbox</code> and want to reference the install location for additional software called <code>mysoftware</code>, replace this code</p> <pre>mysoftwarelocation = 'C:\InstalledSoftware\mysoftware\'</pre> |

| <b>Package a<br/>Toolbox<br/>Dialog Box<br/>Section</b> | <b>Description</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                         | <p>with this code:</p> <pre>mysoftwarelocation = mytoolbox.getInstallationLocation('mysoftware')</pre> <p>To enable testing of the toolbox on your computer before packaging the toolbox, click the <i>toolboxname</i> \getInstallationLocation.mlx link at the bottom of the <b>Installation of Additional Software</b> section and enter the installed location of each additional piece of software on your computer.</p>                                                                                                                                                                                                                                                                                                                                                    |
| <b>Toolbox<br/>Portability</b>                          | <p>MATLAB uses the information in the <b>Toolbox Portability</b> section when the user installs the toolbox. If the compatibility check fails because the user has an unsupported platform or MATLAB version, MATLAB displays a warning. However, the user still can install the toolbox.</p> <p>Platform Compatibility—List of platforms that support the toolbox. Consider if your toolbox has third-party software or hardware requirements that are platform specific. MATLAB Online cannot interact with hardware, including devices used for image acquisition and instrument control.</p> <p>Release Compatibility—List of MATLAB releases that support the toolbox.</p> <p>Products—List of MathWorks products required by your toolbox. Create this list manually.</p> |

| Package a Toolbox Dialog Box Section     | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Examples, Apps, and Documentation</b> | <p>Examples—Published MATLAB examples associated with your toolbox. To include <code>.m</code> and <code>.mlx</code> files as examples, click the <b>Add examples</b> button, select your code file, and click <b>Publish HTML</b>. MATLAB publishes the code to HTML and places the output files in the <code>html</code> folder.</p> <p>Alternatively, you can manually publish code files to HTML in MATLAB and then include the code files and the HTML files in your toolbox folder.</p> <ul style="list-style-type: none"> <li>• For a live script (<code>.mlx</code>) example, export it to HTML. On the <b>Live Editor</b> tab, select <b>Save &gt; Export to HTML</b> and save it in a folder named <code>html</code>.</li> <li>• For a script (<code>.m</code>) example, publish it to HTML with the <code>publish</code> function. Do not specify an output folder when publishing your examples. For the Package a Toolbox tool to recognize the examples, the output folder must be the default folder (<code>html</code>).</li> </ul> <p>To create different categories for your examples, place the examples in different subfolders within your toolbox folder. When you add your toolbox folder to the Package a Toolbox dialog box, MATLAB creates a <code>demos.xml</code> file to describe your examples, and takes the example subfolder name as the example category name. Alternatively, you can create your own <code>demos.xml</code> file. The <code>demos.xml</code> file allows recipients to access your examples through the <b>Supplemental Software</b> link at the bottom of the Help browser home page. For more information, see “Display Custom Examples” on page 30-37.</p> |

| <b>Package a Toolbox Dialog Box Section</b> | <b>Description</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|---------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                             | <p>Apps—Published MATLAB installable apps associated with your toolbox. The Package a Toolbox tool recognizes apps (.mlapp files) and app installer files (.mlappinstall files) and includes them in your toolbox.</p> <ul style="list-style-type: none"> <li>• To specify which apps (.mlapp files) are also installed and registered in the user's MATLAB Apps Gallery, select the apps.</li> <li>• All .mlappinstall files in your toolbox folder are installed and registered in the user's MATLAB Apps Gallery.</li> </ul> <p>Getting Started Guide—Quick start guide for your toolbox. For the Package a Toolbox tool to recognize a Getting Started Guide, include the guide as a live script named GettingStarted.mlx in a doc subfolder within your toolbox folder.</p> <p>Alternatively, you can generate and edit GettingStarted.mlx from the Package a Toolbox dialog box.</p> <p>Users of your toolbox can view the Getting Started Guide through the Options menu for the toolbox in the Add-On Manager. For more information, see “Manage Your Add-Ons”.</p> |

| Package a Toolbox Dialog Box Section | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|--------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                      | <p>Help Browser Integration—Custom documentation associated with your toolbox. For the Package a Toolbox tool to recognize custom documentation, include an <code>info.xml</code> file to identify your documentation files. If you use the <code>builddocsearchdb</code> function to build the documentation database before packaging your toolbox, you can include the generated <code>helpsearch</code> subfolder in your toolbox. The <code>info.xml</code> file and the <code>helpsearch</code> folder allow recipients to access your documentation through the <b>Supplemental Software</b> link at the bottom of the Help browser home page. For more information, see “Display Custom Documentation” on page 30-27.</p> <p>Alternatively, you can generate <code>info.xml</code> and <code>helptoc.xml</code> template files from the Package a Toolbox dialog box. To access your documentation through the Help browser, complete the documentation templates and include <code>info.xml</code> on the MATLAB path.</p> |

## 5 Package your toolbox.

- To save your toolbox, click **Package** at the top of the Package a Toolbox dialog box. Packaging your toolbox generates a `.mltbx` file in your current MATLAB folder.
- To save your toolbox and share it on MATLAB Central File Exchange, select **Package and Share** from the **Package** menu at the top of the Package a Toolbox dialog box. This option generates a `.mltbx` file in your current MATLAB folder and opens a web page for your toolbox submission to File Exchange. MATLAB populates the File Exchange submission form with information about the toolbox. Review and submit the form to share your toolbox on File Exchange.

When you create a toolbox, MATLAB generates a `.prj` file that contains information about the toolbox and saves it frequently. It is good practice to save this associated `.prj` file so that you can quickly create future revisions of your toolbox.

## Share Toolbox

To share your toolbox with others, give them the `.mltbx` file. All files you added when you packaged the toolbox are included in the `.mltbx` file. When the end users install your toolbox, they do not need to be concerned with the MATLAB path or other installation details. The `.mltbx` file manages these details for end users.

For information on installing, uninstalling, and viewing information about toolboxes, see “Get Add-Ons” and “Manage Your Add-Ons”.

You can share your toolbox with others by attaching the `.mltbx` file to an email message, or using any other method you typically use to share files—such as uploading to MATLAB Central File Exchange. If you upload your toolbox to File Exchange, your users can download the toolbox from within MATLAB. For more information, see “Get Add-Ons”.

Alternatively, you can upload your toolbox to File Exchange when you package it. Select **Package and Share** from the **Package** menu at the top of the Package a Toolbox dialog box.

---

**Note** While `.mltbx` files can contain any files you specify, MATLAB Central File Exchange places additional limitations on submissions. If your toolbox contains any of the following, it cannot be submitted to File Exchange:

- MEX-files.
  - Other binary executable files, such as DLLs or ActiveX® controls. (Data and image files are typically acceptable.)
- 

## See Also

`matlab.addons.toolbox.installToolbox` |  
`matlab.addons.toolbox.installedToolboxes` |  
`matlab.addons.toolbox.packageToolbox` |  
`matlab.addons.toolbox.toolboxVersion` |  
`matlab.addons.toolbox.uninstallToolbox` | `publish`

## Related Examples

- “Get Add-Ons”

- “Manage Your Add-Ons”
- “Display Custom Examples” on page 30-37
- “Package Apps From the MATLAB Toolstrip”
- “Display Custom Documentation” on page 30-27

# Software Development



# Error Handling

---

- “Exception Handling in a MATLAB Application” on page 26-2
- “Capture Information About Exceptions” on page 26-5
- “Throw an Exception” on page 26-15
- “Respond to an Exception” on page 26-17
- “Clean Up When Functions Complete” on page 26-22
- “Issue Warnings and Errors” on page 26-28
- “Suppress Warnings” on page 26-32
- “Restore Warnings” on page 26-35
- “Change How Warnings Display” on page 26-38
- “Use try/catch to Handle Errors” on page 26-40

## Exception Handling in a MATLAB Application

### In this section...

“Overview” on page 26-2

“Getting an Exception at the Command Line” on page 26-2

“Getting an Exception in Your Program Code” on page 26-3

“Generating a New Exception” on page 26-4

### 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 “Debug a MATLAB Program” on page 22-2 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 “Capture Information About Exceptions” on page 26-5 describes how to acquire information about what caused the error, and “Respond to an Exception” on page 26-17 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 “Capture Information About Exceptions” on page 26-5 describes how to use an `MException` object to capture information about the error, and “Throw an Exception” on page 26-15 explains how to initiate the exception process.

# Capture Information About Exceptions

## In this section...

“Overview” on page 26-5

“The MException Class” on page 26-5

“Properties of the MException Class” on page 26-7

“Methods of the MException Class” on page 26-13

## 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.

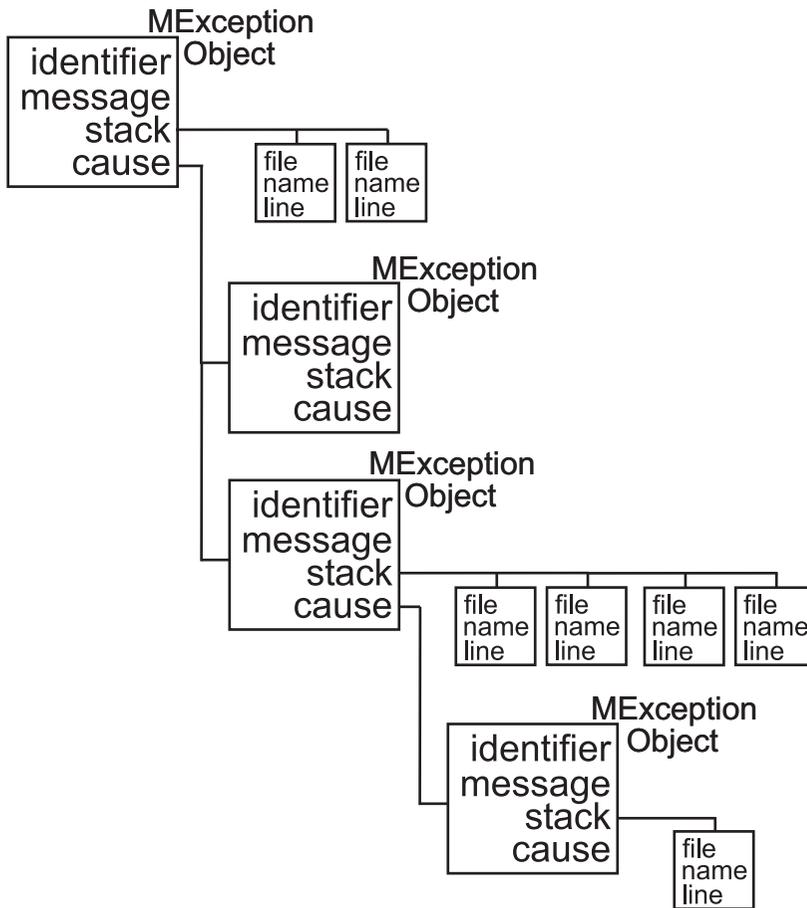
This section describes the `MException` class and objects constructed from that class:

Information on how to use this class is presented in later sections on “Respond to an Exception” on page 26-17 and “Throw an Exception” on page 26-15.

## 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.

See “Properties of the MException Class” on page 26-7 for a full description of these properties.



**Object Constructor**

Any code that detects an error and throws an exception must also construct an MException object in which to record and transfer information about the error. The syntax of the MException constructor is

```
ME = MException(identifier, message)
```

where identifier is a MATLAB message identifier on page 26-8 of the form component:mnemonic

that is enclosed in single quotes, and `message` is text, 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 “Respond to an Exception” on page 26-17 and “Throw an Exception” on page 26-15. All are read-only; their values cannot be changed.

The `MException` properties are:

- `identifier` on page 26-8
- `message` on page 26-10
- `stack` on page 26-11
- `cause` on page 26-12

If you call the `surf` function with no inputs, MATLAB throws an exception. If you catch 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 26-17 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 vector 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 character vector.

### 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 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, text)
```

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 vector 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, text)
```

If your message 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.%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
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.

| <b>Method Name</b>                    | <b>Description</b>                                                                                     |
|---------------------------------------|--------------------------------------------------------------------------------------------------------|
| <code>MException.addCause</code>      | Append an <code>MException</code> to the <code>cause</code> field of another <code>MException</code> . |
| <code>MException.getReport</code>     | Return a formatted message based on the current exception.                                             |
| <code>MException.last</code>          | Return the last uncaught exception. This is a static method.                                           |
| <code>MException.rethrow</code>       | Reissue an exception that has previously been caught.                                                  |
| <code>MException.throw</code>         | Issue an exception.                                                                                    |
| <code>MException.throwAsCaller</code> | Issue an exception, but omit the current stack frame from the <code>stack</code> field.                |

## Throw 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` or `try/catch` statement that checks the output of the current operation.
- 2 Construct an `MException` object to represent the error. Add a message identifier and error message 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:

Create an array, and an index into it with a logical array.

```
A = [13 42; 7 20];
idx = [1 0 1; 0 1 0];
```

Create an exception that provides general information about an error. Test the index array and add exceptions with more detailed information about the source of the failure.

```
% 1) Detect the error.
try
 A(idx);
catch

 % 2) Construct an MException object to represent the error.
 msgID = 'MYFUN:BadIndex';
 msg = 'Unable to index into array.';
 baseException = MException(msgID,msg);

 % 3) Store any information contributing to the error.
 try
 assert(islogical(idx), 'MYFUN:notLogical', ...
 'Indexing array is not logical.')
```

```
catch causeException
 baseException = addCause(baseException,causeException);
end

if any(size(idx) > size(A))
 msgID = 'MYFUN:incorrectSize';
 msg = 'Indexing array is too large.';
 causeException2 = MException(msgID,msg);
 baseException = addCause(baseException,causeException2);
end

% 4) Throw the exception to stop execution and display an error
% message.
throw(baseException)
end

Unable to index into array.

Caused by:
 Indexing array is not logical.
 Indexing array is too large.
```

## Respond to an Exception

### In this section...

“Overview” on page 26-17

“The try/catch Statement” on page 26-17

“Suggestions on How to Handle an Exception” on page 26-19

### Overview

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 26-5 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 “Capture Information About Exceptions” on page 26-5.

## 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.

## Clean Up When Functions Complete

### In this section...

“Overview” on page 26-22

“Examples of Cleaning Up a Program Upon Exit” on page 26-23

“Retrieving Information About the Cleanup Routine” on page 26-25

“Using onCleanup Versus try/catch” on page 26-26

“onCleanup in Scripts” on page 26-27

### Overview

A good programming practice is to 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 function testCleanup
k = 3; k = 3;
myFun obj = onCleanup(@myFun);
 function myFun function myFun
 fprintf('k is %d\n', k) fprintf('k is %d\n', k)
 end end
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(figureHandle, newPath)
 disp ' Closing the figure'
 close(figureHandle);
```

```

 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.

## Issue Warnings and Errors

### In this section...

“Issue Warnings” on page 26-28

“Throw Errors” on page 26-28

“Add Run-Time Parameters to Your Warnings and Errors” on page 26-29

“Add Identifiers to Warnings and Errors” on page 26-30

### Issue Warnings

You can issue a warning to flag unexpected conditions detected when running a program. The `warning` function prints a warning message to the command line. Warnings differ from errors in two significant ways:

- Warnings do not halt the execution of the program.
- You can suppress any unhelpful MATLAB warnings.

Use the `warning` function in your code to generate a warning message during execution. Specify the message as the input argument to the `warning` function:

```
warning('Input must be text')
```

For example, you can insert a warning in your code to verify the software version:

```
function warningExample1
 if ~strncmp(version, '7', 1)
 warning('You are using a version other than v7')
 end
```

### Throw Errors

You can throw an error to flag fatal problems within the program. Use the `error` function to print error messages to the command line. After displaying the message, MATLAB stops the execution of the current program.

For example, suppose you construct a function that returns the number of combinations of  $k$  elements from  $n$  elements. Such a function is nonsensical if  $k > n$ ; you cannot choose 8 elements if you start with just 4. You must incorporate this fact into the function to let anyone using `combinations` know of the problem:

```
function com = combinations(n,k)
 if k > n
 error('Cannot calculate with given values')
 end
 com = factorial(n)/(factorial(k)*factorial(n-k));
end
```

If the `combinations` function receives invalid input, MATLAB stops execution immediately after throwing the error message:

```
combinations(4,8)
```

```
Error using combinations (line 3)
Cannot calculate with given values
```

## Add Run-Time Parameters to Your Warnings and Errors

To make your warning or error messages more specific, insert components of the message at the time of execution. The warning function uses *conversion characters* that are the same as those used by the `sprintf` function. Conversion characters act as placeholders for substrings or values, unknown until the code executes.

For example, this warning uses `%s` and `%d` to mark where to insert the values of variables `arrayname` and `arraydims`:

```
warning('Array %s has %d dimensions.',arrayname,arraydims)
```

If you execute this command with `arrayname = 'A'` and `arraydims = 3`, MATLAB responds:

```
Warning: Array A has 3 dimensions.
```

Adding run-time parameters to your warnings and errors can clarify the problems within a program. Consider the function `combinations` from “Throw Errors” on page 26-28. You can throw a much more informative error using run-time parameters:

```
function com = combinations(n,k)
 if k > n
 error('Cannot choose %i from %i elements',k,n)
 end
 com = factorial(n)/(factorial(k)*factorial(n-k));
end
```

If this function receives invalid arguments, MATLAB throws an error message and stops the program:

```
combinations(6,9)
```

```
Error using combinations (line 3)
Cannot choose 9 from 6 elements
```

## Add Identifiers to Warnings and Errors

A message identifier provides a way to uniquely reference a warning or an error.

Enable or disable warnings with identifiers. Use an identifying text argument with the `warning` function to attach a unique tag to a message:

```
warning(identifier_text,message_text)
```

For example, you can add an identifier tag to the previous MATLAB warning about which version of software is running:

```
minver = '7';
if ~strcmp(version,minver,1)
 warning('MYTEST:VERCHK','Running a version other than v%s',minver)
end
```

Adding an identifier to an error message allows for negative testing. However, adding and recovering more information from errors often requires working with `MException` objects.

## See Also

`MException` | `lastwarn` | `warndlg` | `warning`

## Related Examples

- “Suppress Warnings” on page 26-32
- “Restore Warnings” on page 26-35
- “Capture Information About Exceptions” on page 26-5
- “Exception Handling in a MATLAB Application” on page 26-2

## **More About**

- “Message Identifiers” on page 26-8

## Suppress Warnings

Your program might issue warnings that do not always adversely affect execution. To avoid confusion, you can hide warning messages during execution by changing their states from 'on' to 'off'.

To suppress specific warning messages, you must first find the warning identifier. Each warning message has a unique identifier. To find the identifier associated with a MATLAB warning, reproduce the warning. For example, this code reproduces a warning thrown if MATLAB attempts to remove a nonexistent folder:

```
rmpath('folderthatisnotonpath')
```

```
Warning: "folderthatisnotonpath" not found in path.
```

---

**Note** If this statement does not produce a warning message, use the following code to temporarily enable the display of all warnings, and then restore the original warning state:

```
w = warning('on','all');
rmpath('folderthatisnotonpath')
warning(w)
```

---

To obtain information about the most recently issued warning, use the `warning` or `lastwarn` functions. This code uses the `query` state to return a data structure containing the message identifier and the current state of the last warning:

```
w = warning('query','last')

w =
 identifier: 'MATLAB:rmpath:DirNotFound'
 state: 'on'
```

You can save the identifier field in the variable, `id`:

```
id = w.identifier;
```

---

**Note** `warning('query','last')` returns the last displayed warning. MATLAB only displays warning messages that have `state: 'on'` and a warning identifier.

---

Using the `lastwarn` function, you can retrieve the last warning message, regardless of its display state:

```
lastwarn
ans =
"folderthatisnotonpath" not found in path.
```

## Turn Warnings On and Off

After you obtain the identifier from the query state, use this information to disable or enable the warning associated with that identifier.

Continuing the example from the previous section, turn the warning 'MATLAB:rmpath:DirNotFound' off, and repeat the operation.

```
warning('off',id)
rmpath('folderthatisnotonpath')
```

MATLAB displays no warning.

Turn the warning on, and try to remove a nonexistent path:

```
warning('on',id)
rmpath('folderthatisnotonpath')
```

```
Warning: "folderthatisnotonpath" not found in path.
```

MATLAB now issues a warning.

---

**Tip** Turn off the most recently invoked warning with `warning('off','last')`.

---

## Controlling All Warnings

The term **all** refers *only* to those warnings that have been issued or modified during your current MATLAB session. Modified warning states persist only through the current session. Starting a new session restores the default settings.

Use the identifier 'all' to represent the group of all warnings. View the state of all warnings with either syntax:

```
warning('query', 'all')
```

```
warning
```

To enable all warnings and verify the state:

```
warning('on', 'all')
warning('query', 'all')
```

All warnings have the state 'on'.

To disable all warnings and verify the state, use this syntax:

```
warning('off', 'all')
warning
```

All warnings have the state 'off'.

## See Also

### Related Examples

- “Restore Warnings” on page 26-35
- “Change How Warnings Display” on page 26-38

## Restore Warnings

MATLAB allows you to save the on-off warning states, modify warning states, and restore the original warning states. This is useful if you need to temporarily turn off some warnings and later reinstate the original settings.

The following statement saves the current state of all warnings in the structure array called `orig_state`:

```
orig_state = warning;
```

To restore the original state after any warning modifications, use this syntax:

```
warning(orig_state);
```

You also can save the current state and toggle warnings in a single command. For example, the statement, `orig_state = warning('off', 'all');` is equivalent to the commands:

```
orig_state = warning;
warning('off', 'all')
```

### Disable and Restore a Particular Warning

This example shows you how to restore the state of a particular warning.

- 1 Query the `Control:parameterNotSymmetric` warning:

```
warning('query', 'Control:parameterNotSymmetric')
```

The state of warning 'Control:parameterNotSymmetric' is 'on'.

- 2 Turn off the `Control:parameterNotSymmetric` warning:

```
orig_state = warning('off', 'Control:parameterNotSymmetric')
```

```
orig_state =
```

```
 identifier: 'Control:parameterNotSymmetric'
 state: 'on'
```

`orig_state` contains the warning state before MATLAB turns `Control:parameterNotSymmetric` off.

- 3 Query all warning states:

```
warning
```

The default warning state is 'on'. Warnings not set to the default are

```
State Warning Identifier
```

```
off Control:parameterNotSymmetric
```

MATLAB indicates that `Control:parameterNotSymmetric` is 'off'.

- 4 Restore the original state:

```
warning(orig_state)
warning('query', 'Control:parameterNotSymmetric')
```

The state of warning 'Control:parameterNotSymmetric' is 'on'.

## Disable and Restore Multiple Warnings

This example shows you how to save and restore multiple warning states.

- 1 Disable three warnings, and query all the warnings:

```
w(1) = warning('off', 'MATLAB:rmpath:DirNotFound');
w(2) = warning('off', 'MATLAB:singularMatrix');
w(3) = warning('off', 'Control:parameterNotSymmetric');
warning
```

The default warning state is 'on'. Warnings not set to the default are

```
State Warning Identifier
```

```
off Control:parameterNotSymmetric
off MATLAB:rmpath:DirNotFound
off MATLAB:singularMatrix
```

- 2 Restore the three warnings to their the original state, and query all warnings:

```
warning(w)
warning
```

All warnings have the state 'on'.

You do not need to store information about the previous warning states in an array, but doing so allows you to restore warnings with one command.

---

**Note** When temporarily disabling multiple warnings, using methods related to `onCleanup` might be advantageous.

---

Alternatively, you can save and restore all warnings.

- 1 Enable all warnings, and save the original warning state:

```
orig_state = warning('on', 'all');
```

- 2 Restore your warnings to the previous state:

```
warning(orig_state)
```

## See Also

`onCleanup` | `warning`

## Related Examples

- “Suppress Warnings” on page 26-32
- “Clean Up When Functions Complete” on page 26-22

## Change How Warnings Display

You can control how warnings appear in MATLAB by modifying two warning *modes*, `verbose` and `backtrace`.

| Mode                   | Description                                       | Default                   |
|------------------------|---------------------------------------------------|---------------------------|
| <code>verbose</code>   | Display a message on how to suppress the warning. | <code>off (terse)</code>  |
| <code>backtrace</code> | Display a stack trace after a warning is invoked. | <code>on (enabled)</code> |

**Note** The `verbose` and `backtrace` modes present some limitations:

- `prev_state` does not contain information about the `backtrace` or `verbose` modes in the statement, `prev_state = warning('query','all')`.
- A mode change affects all enabled warnings.

### Enable Verbose Warnings

When you enable verbose warnings, MATLAB displays an extra line of information with each warning that tells you how to suppress it.

For example, you can turn on all warnings, disable `backtrace`, and enable verbose warnings:

```
warning on all
warning off backtrace
warning on verbose
```

Running a command that produces an error displays an extended message:

```
rmpath('folderthatisnotonpath')
```

```
Warning: "folderthatisnotonpath" not found in path.
(Type "warning off MATLAB:rmpath:DirNotFound" to suppress this warning.)
```

### Display 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. When you enable the `backtrace` mode, MATLAB displays

the file name and line number where the warning occurred. For example, you can enable backtrace and disable verbose:

```
warning on backtrace
warning off verbose
```

Running a command that produces an error displays a hyperlink with a line number:

```
Warning: "folderthatisonotpath" not found in path.
> In rmpath at 58
```

Clicking the hyperlink takes you to the location of the warning.

## Use try/catch to Handle Errors

You can use a `try/catch` statement to execute code after your program encounters an error. `try/catch` statements can be useful if you:

- Want to finish the program in another way that avoids errors
- Need to clean up unwanted side effects of the error
- Have many problematic input parameters or commands

Arrange `try/catch` statements into blocks of code, similar to this pseudocode:

```
try
 try block...
catch
 catch block...
end
```

If an error occurs within the *try block*, MATLAB skips any remaining commands in the *try block* and executes the commands in the *catch block*. If no error occurs within *try block*, MATLAB skips the entire *catch block*.

For example, a `try/catch` statement can prevent the need to throw errors. Consider the `combinations` function that returns the number of combinations of  $k$  elements from  $n$  elements:

```
function com = combinations(n,k)
 com = factorial(n)/(factorial(k)*factorial(n-k));
end
```

MATLAB throws an error whenever  $k > n$ . You cannot construct a set with more elements,  $k$ , than elements you possess,  $n$ . Using a `try/catch` statement, you can avoid the error and execute this function regardless of the order of inputs:

```
function com = robust_combine(n,k)
 try
 com = factorial(n)/(factorial(k)*factorial(n-k));
 catch
 com = factorial(k)/(factorial(n)*factorial(k-n));
 end
end
```

`robust_combine` treats any order of integers as valid inputs:

```
C1 = robust_combine(8,4)
C2 = robust_combine(4,8)
```

```
C1 =
 70
```

```
C2 =
 70
```

Optionally, you can capture more information about errors if a variable follows your catch statement:

```
catch MExc
```

MExc is an MException class object that contains more information about the thrown error. To learn more about accessing information from MException objects, see “Exception Handling in a MATLAB Application” on page 26-2.

## See Also

MException | onCleanup



# Program Scheduling

---

- “Schedule Command Execution Using Timer” on page 27-2
- “Timer Callback Functions” on page 27-5
- “Handling Timer Queuing Conflicts” on page 27-10

## Schedule Command Execution Using Timer

|                                              |
|----------------------------------------------|
| <b>In this section...</b>                    |
| “Overview” on page 27-2                      |
| “Example: Displaying a Message” on page 27-3 |

### 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.

- 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 `timer` and `set`. 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.

- 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 `delete` 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 character vector 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 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:

```
.
.
.
.
.
.
.
.
.
.
Timer!
```

```
delete(t) % Always delete timer objects after using them.
```

## See Also

timer

## **More About**

- “Timer Callback Functions” on page 27-5
- “Handling Timer Queuing Conflicts” on page 27-10

## Timer Callback Functions

| In this section...                                                  |
|---------------------------------------------------------------------|
| “Associating Commands with Timer Object Events” on page 27-5        |
| “Creating Callback Functions” on page 27-6                          |
| “Specifying the Value of Callback Function Properties” on page 27-8 |

---

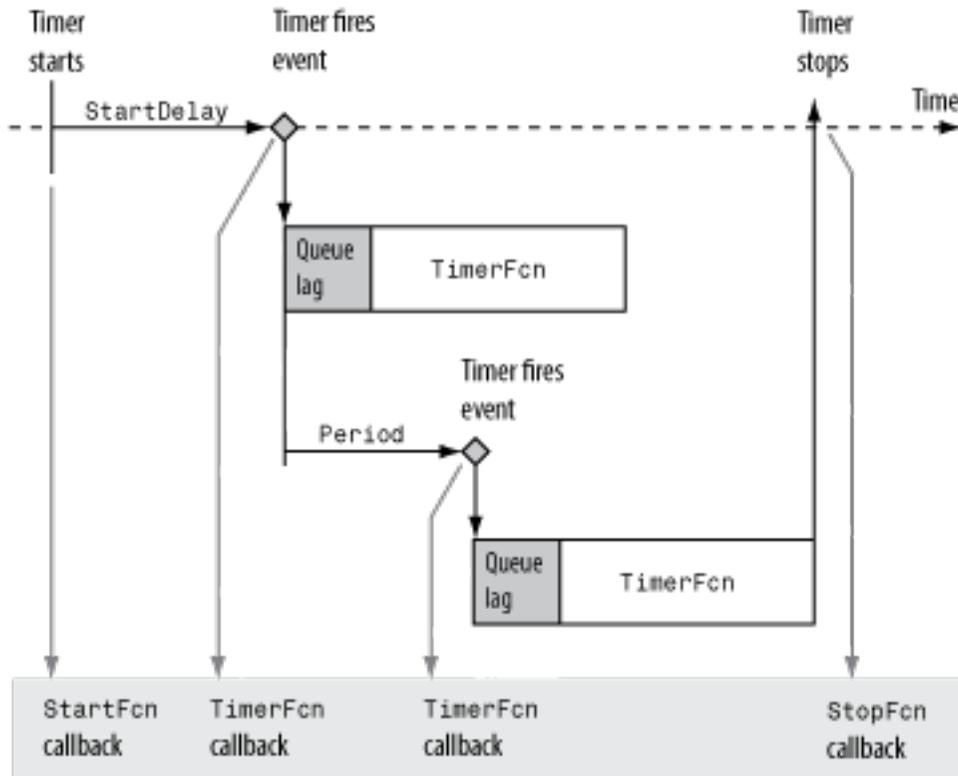
**Note** Callback function execution might be delayed if the callback involves a CPU-intensive task such as updating a figure.

---

### 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 character vector.

```
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 character vector that identifies the type of event that caused the callback. The value of this field can be any of the following: '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 27-8.

### 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 character vector and includes this text in the display output. To see this function used with a callback property, see “Specifying the Value of Callback Function Properties” on page 27-8.

```
function my_callback_fcn(obj, event, text_arg)

txt1 = ' event occurred at ';
txt2 = text_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 <code>t</code> |
|------------------------------------------------------|--------------------------------------------------------------|
| <code>function myfile(obj, event)</code>             | <code>t.StartFcn = @myfile</code>                            |
| <code>function myfile</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 27-7:

- 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 character vector:

- ```
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)
```

See Also

timer

More About

- “Handling Timer Queuing Conflicts” on page 27-10

Handling Timer 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 setting the `BusyMode` property to use one of these modes:

In this section...
"Drop Mode (Default)" on page 27-10
"Error Mode" on page 27-11
"Queue Mode" on page 27-13

Drop Mode (Default)

If you specify 'drop' as the value of the `BusyMode` property, the timer object adds the timer callback function to the execution queue only when the queue is empty. If the execution queue is not empty, the timer object skips the execution of the callback.

For example, suppose you create a timer with a period of 1 second, but a callback that requires at least 1.6 seconds, as shown here for `mytimer.m`.

```
function mytimer()
    t = timer;

    t.Period          = 1;
    t.ExecutionMode   = 'fixedRate';
    t.TimerFcn        = @mytimer_cb;
    t.BusyMode        = 'drop';
    t.TasksToExecute = 5;
    t.UserData        = tic;

    start(t)
end

function mytimer_cb(h,~)
    timeStart = toc(h.UserData)
    pause(1.6);
    timeEnd = toc(h.UserData)
end
```

This table describes how the timer manages the execution queue.

Approximate Elapsed Time (Seconds)	Action
0	Start the first execution of the callback.
1	Attempt to start the second execution of the callback. The first execution is not complete, but the execution queue is empty. The timer adds the callback to the queue.
1.6	Finish the first callback execution, and start the second. This action clears the execution queue.
2	Attempt to start the third callback execution. The second execution is not complete, but the queue is empty. The timer adds the callback to the queue.
3	Attempt to start the fourth callback execution. The third callback is in the execution queue, so the timer drops this execution of the function.
3.2	Finish the second callback and start the third, clearing the execution queue.
4	Attempt to start another callback execution. Because the queue is empty, the timer adds the callback to the queue. This is the fifth attempt, but only the fourth instance that will run.
4.8	Finish the third execution and start the fourth instance, clearing the queue.
5	Attempt to start another callback. An instance is running, but the execution queue is empty, so the timer adds it to the queue. This is the fifth instance that will run.
6	Do nothing: the value of the <code>TasksToExecute</code> property is 5, and the fifth instance to run is in the queue.
6.4	Finish the fourth callback execution and start the fifth.
8	Finish the fifth callback execution.

Error Mode

The 'error' mode for the `BusyMode` property is similar to the 'drop' mode: In both modes, the timer allows only one instance of the callback in the execution queue.

However, in 'error' mode, when the queue is nonempty, the timer calls the function that you specify using the ErrorFcn property, and then stops processing. The currently running callback function completes, but the callback in the queue does not execute.

For example, modify `mytimer.m` (described in the previous section) so that it includes an error handling function and sets `BusyMode` to 'error'.

```
function mytimer()
    t = timer;

    t.Period          = 1;
    t.ExecutionMode   = 'fixedRate';
    t.TimerFcn        = @mytimer_cb;
    t.ErrorFcn         = @myerror;
    t.BusyMode        = 'error';
    t.TasksToExecute  = 5;
    t.UserData         = tic;

    start(t)
end

function mytimer_cb(h,~)
    timeStart = toc(h.UserData)
    pause(1.6);
    timeEnd = toc(h.UserData)
end

function myerror(h,~)
    disp('Reached the error function')
end
```

This table describes how the timer manages the execution queue.

Approximate Elapsed Time (Seconds)	Action
0	Start the first execution of the callback.
1	Attempt to start the second execution of the callback. The first execution is not complete, but the execution queue is empty. The timer adds the callback to the queue.

Approximate Elapsed Time (Seconds)	Action
1.6	Finish the first callback execution, and start the second. This action clears the execution queue.
2	Attempt to start the third callback execution. The second execution is not complete, but the queue is empty. The timer adds the callback to the queue.
3	Attempt to start the fourth callback execution. The third callback is in the execution queue. The timer does not execute the third callback, but instead calls the error handling function.
3.2	Finish the second callback and start the error handling function.

Queue Mode

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.

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.

See Also

timer

More About

- “Timer Callback Functions” on page 27-5

Performance

- “Measure Performance of Your Program” on page 28-2
- “Profile to Improve Performance” on page 28-5
- “Use Profiler to Determine Code Coverage” on page 28-12
- “Techniques to Improve Performance” on page 28-14
- “Preallocation” on page 28-17
- “Vectorization” on page 28-19

Measure Performance of Your Program

In this section...

“Overview of Performance Timing Functions” on page 28-2

“Time Functions” on page 28-2

“Time Portions of Code” on page 28-2

“The `cputime` Function vs. `tic/toc` and `timeit`” on page 28-3

“Tips for Measuring Performance” on page 28-3

Overview of Performance Timing Functions

The `timeit` function and the stopwatch timer functions, `tic` and `toc`, enable you to time how long your code takes to run. Use the `timeit` function for a rigorous measurement of function execution time. Use `tic` and `toc` to estimate time for smaller portions of code that are not complete functions.

For additional details about the performance of your code, such as function call information and execution time of individual lines of code, use the MATLAB Profiler. For more information, see “Profile to Improve Performance” on page 28-5.

Time Functions

To measure the time required to run a function, use the `timeit` function. The `timeit` function calls the specified function multiple times, and returns the median of the measurements. It takes a handle to the function to be measured and returns the typical execution time, in seconds. Suppose that you have defined a function, `computeFunction`, that takes two inputs, `x` and `y`, that are defined in your workspace. You can compute the time to execute the function using `timeit`.

```
f = @() myComputeFunction(x,y); % handle to function
timeit(f)
```

Time Portions of Code

To estimate how long a portion of your program takes to run or to compare the speed of different implementations of portions of your program, use the stopwatch timer functions, `tic` and `toc`. Invoking `tic` starts the timer, and the next `toc` reads the elapsed time.

```
tic
    % The program section to time.
toc
```

Sometimes programs run too fast for `tic` and `toc` to provide useful data. If your code is faster than 1/10 second, consider measuring it running in a loop, and then average to find the time for a single run.

The `cputime` Function vs. `tic/toc` and `timeit`

It is recommended that you use `timeit` or `tic` and `toc` to measure the performance of your code. These functions return wall-clock time. Unlike `tic` and `toc`, the `timeit` function calls your code multiple times, and, therefore, considers first-time costs.

The `cputime` function measures the total CPU time and sums across all threads. This measurement is different from the wall-clock time that `timeit` or `tic/toc` return, and could be misleading. For example:

- The CPU time for the `pause` function is typically small, but the wall-clock time accounts for the actual time that MATLAB execution is paused. Therefore, the wall-clock time might be longer.
- If your function uses four processing cores equally, the CPU time could be approximately four times higher than the wall-clock time.

Tips for Measuring Performance

Consider the following tips when you are measuring the performance of your code:

- Time a significant enough portion of code. Ideally, the code you are timing should take more than 1/10 second to run.
- Put the code you are trying to time into a function instead of timing it at the command line or inside a script.
- Unless you are trying to measure first-time cost, run your code multiple times. Use the `timeit` function.
- Avoid `clear all` when measuring performance. For more information, see the `clear` function.
- Assign your output to a variable instead of letting it default to `ans`.

See Also

`profile` | `tic` | `timeit` | `toc`

Related Examples

- “Profile to Improve Performance” on page 28-5
- “Techniques to Improve Performance” on page 28-14
- MATLAB Performance Measurement White Paper on MATLAB Central File Exchange

Profile to Improve Performance

In this section...

“What Is Profiling?” on page 28-5
“Profiling Process and Guidelines” on page 28-5
“Using the Profiler” on page 28-6
“Profile Summary Report” on page 28-8
“Profile Detail Report” on page 28-9

What Is Profiling?

Profiling is a way to measure where a program spends time. After you identify which functions are consuming the most time, you can evaluate them for possible performance improvements. Also, you can profile your code as a debugging tool. For example, determining which lines of code MATLAB does not run can help you develop test cases that exercise that code. If you get an error in the file when profiling, you can see what ran and what did not to help you isolate the problem.

Tip Code that is prematurely optimized can be unnecessarily complex without providing a significant gain in performance. Make your first implementation as simple as possible. Then, if speed is an issue, use profiling to identify bottlenecks.

You can profile your code using the MATLAB Profiler. The Profiler is a user interface based on the results returned by the `profile` function. If you are profiling code that runs in parallel, for best results use the Parallel Computing Toolbox™ parallel profiler. For details, see “Profiling Parallel Code” (Parallel Computing Toolbox).

Profiling Process and Guidelines

Use this general process to improve performance in your code:

- 1 Run the Profiler on your code.
- 2 In the Profile Summary report, look for functions that use a significant amount of time or that are called most frequently.
- 3 View the Profile Detail report for those functions, and look for the lines of code that take the most time or are called most often.

Consider keeping a copy of your first detail report as a basis for comparison. After you change your code, you can run the Profiler again and compare the reports.

- 4 Determine whether there are changes you can make to those lines of code to improve performance.

For example, if you have a `load` statement within a loop, you might be able to move the `load` statement outside the loop so that it is called only once.

- 5 Implement the potential performance improvements in your code. Save the files, and run `clear all`. Run the Profiler again and compare the results to the original report.

If you profile the identical code twice, you can get slightly different results each time due to inherent time fluctuations that are not dependent on your code.

- 6 To continue improving the performance of your code, repeat these steps.

When your code spends most of its time on calls to a few built-in functions, you have probably optimized the code as much as possible.

Using the Profiler

To profile a MATLAB code file or a line of code:

- 1 Open the Profiler using one of the following methods:
 - In the Command Window, type `profile viewer`.
 - On the **Home** tab, in the **Code** section, click  **Run and Time**.
 - In the Editor, on the **Editor** tab, in the **Run** section, click  **Run and Time**. If you use this method, the Profiler automatically profiles the code in the current Editor tab. If that is the code you want to profile, skip to step 4.
- 2 In the **Run this code** field, type the statement you want to run.

For example, you can run the Lotka-Volterra example, which is provided with MATLAB:

```
[t,y] = ode23('lotka',[0 2],[20;20])
```

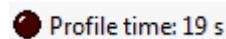
If, in the current MATLAB session, you previously profiled the statement, select it from the **Run this code** list. MATLAB automatically starts profiling the code, and you can skip to step 4.

3 Click **Start Profiling**.

While the Profiler is running, the **Profile time** indicator is green and the number of seconds it reports increases. The **Profile time** indicator appears at the top right of the Profiler window.



When the Profiler finishes, the **Profile time** indicator turns black and shows the length of time the Profiler ran. The statements you profiled display as having been executed in the Command Window.



This time is not the actual time that your statements took to run. It is the time elapsed from when you clicked **Start Profiling** until the profiling stops. If the time reported is very different from what you expected (for example, hundreds of seconds for a simple statement), you might have profiled longer than necessary. This time does not match the time reported in Profile Summary report statistics, which is based on performance clock time by default. To view profile statistics using a different type of clock, use the `profile` function instead of the Profiler.

- 4 When profiling is complete, the Profile Summary report appears in the Profiler window. For more information, see “Profile Summary Report” on page 28-8.

Profile Multiple Statements in Command Window

To profile more than one statement:

- 1 In the Profiler, click **Start Profiling**. Make sure that no code appears in the **Run this code** field.
- 2 In the Command Window, enter and run the statements you want to profile.
- 3 After running all the statements, click **Stop Profiling** in the Profiler, and view the Profile Summary report.

Profile a User Interface

You can run the Profiler for a user interface, such as the Filter Design and Analysis tool included with Signal Processing Toolbox. Or, you can profile an interface you created, such as one built using GUIDE.

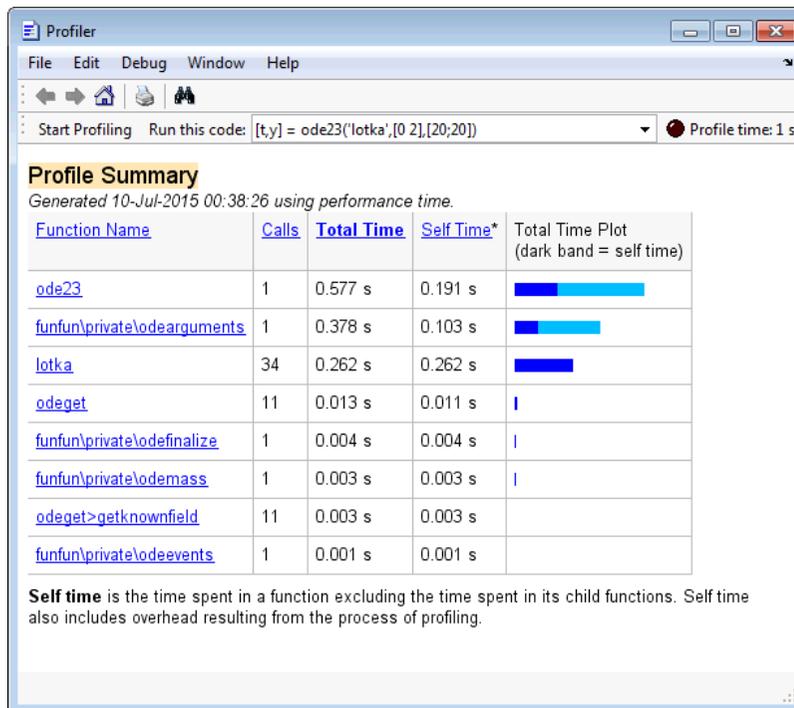
To profile a user interface:

- 1 In the Profiler, click **Start Profiling**. Make sure that no code appears in the **Run this code** field.
- 2 Start the user interface.
- 3 Use the interface. When you finish, click **Stop Profiling** in the Profiler, and view the Profile Summary report.

Note To exclude the user interface startup process in the profile, reverse steps 1 and 2. In other words, start the user interface before you click **Start Profiling**.

Profile Summary Report

The Profile Summary report presents statistics about the overall execution of the function and provides summary statistics for each function called. The following is an image of the Profile Summary report for the Lotka-Volterra model. See “Using the Profiler” on page 28-6.



The Profile Summary report presents this information.

Column	Description
Function Name	List of all the functions called by the profiled code. Initially the functions appear in order of time they took to process.
Calls	Number of times the profiled code called the function.
Total Time	Total time spent in a function, including all accessed child functions, in seconds. The time for a function includes time spent in child functions. The Profiler itself takes some time, which is included in the results. The total time can be zero for files whose run time is inconsequential.
Self Time	Total time in seconds spent in a function, excluding time spent in any child functions. Self time also includes some overhead resulting from the process of profiling.
Total Time Plot	Graphic display showing self time compared to total time.

In the summary report, you can:

- Print the report, by clicking the print button .
- Get more detailed information about a particular function by clicking its name in the **Function Name** column. For more information, see “Profile Detail Report” on page 28-9.
- Sort by a given column by clicking the name of the column. For example, click the **Function Name** link to sort the functions alphabetically. Initially the results appear in order by **Total Time**.

Profile Detail Report

The Profile Detail report shows profiling results for a function that MATLAB called while profiling.

To open the Profile Detail report, click a function name in the Profile Summary report. To return to the Profile Summary report from the Profile Detail report, click  in the toolbar of the Profile window.

The header of the Profile Detail report contains this information.

- Name of the profiled function
- Number of times the parent function called the profiled function
- Time spent in the profiled function
- Link to open the function in your default editor
- Link to copy the report to a separate window. Saving a copy of the report is helpful to compare the impact of changes to your function. when you change the file.

To specify which sections the Profile Detail Report includes, select the check boxes at the top of the report, and click the **Refresh** button. Use the check boxes to select from these options.

Display Option	Details
Show parent functions	Display information about the parent functions, with links to their detail reports. To open a Profile Detail report for a parent function, click the name of the function.
Show busy lines	List the lines in the profiled function that used the greatest amount of processing time.
Show child functions	List all the functions called by the profiled function. To open a Profile Detail report for a child function, click the name of the function.
Show Code Analyzer results	Display information about problems and potential improvements for the profiled function.
Show file coverage	Display statistics about the lines of code in the function that MATLAB executed while profiling.

Display Option	Details
Show function listing	<p>Display the source code for the function, if it is a MATLAB code file.</p> <p>For each line of code, the Function listing includes these columns:</p> <ul style="list-style-type: none"> • Execution time for each line of code • Number of times that MATLAB executed the line of code • The line number • The source code for the function. The color of the text indicates the following: <ul style="list-style-type: none"> • Green — Commented lines • Black — Executed lines of code • Gray — Non-executed lines of code <p>By default, the Profile Detail report highlights lines of code with the longest execution time. The darker the highlighting, the longer the line of code took to execute. To change the highlighting criteria, use the color highlight code drop-down list.</p>

See Also

profile

More About

- “Measure Performance of Your Program” on page 28-2
- “Techniques to Improve Performance” on page 28-14
- “Use Profiler to Determine Code Coverage” on page 28-12

Use Profiler to Determine Code Coverage

When you run the Profiler on a file, some code might not run, such as a block containing an `if` statement.

To determine how much of a file MATLAB executed when you profiled it, run the Coverage Report.

- 1 Profile your MATLAB code file. For more information, see “Profile to Improve Performance” on page 28-5 or the `profile` function.
- 2 Ensure that the Profiler is not currently profiling.
 - In the Profiler, a **Stop Profiling** button displays if the Profiler is running. If the Profiler is running, click the **Stop Profiling** button.
 - At the command prompt, check the Profiler status using `profile status`. If the `ProfilerStatus` is 'on', stop the Profiler by typing `profile off`.
- 3 Use the Current Folder browser to navigate to the folder containing the profiled code file.

Note You cannot run reports when the path is a UNC (Universal Naming Convention) path; that is, a path that starts with `\\`. Instead, use an actual hard drive on your system, or a mapped network drive.

- 4 On the Current Folder browser, click , and then select **Reports > Coverage Report**.

The Profiler Coverage Report opens, providing a summary of coverage for the profiled file. In the following image, the profiled file is `lengthofline2.m`.

Profiler Coverage Report

Run the Coverage Report after you run the Profiler to identify how much of a file ran when it was profiled ([Learn More](#)).

Report for folder I:\my_MATLAB_files

lengthofline2.m	 <p>Coverage: 87.9% Total time: 0.2 seconds Total lines:</p>
api2.m	
area.m	
basic matrix.m	
breakpoint.m	
collat2z.m	

- 5 Click the **Coverage** link to see the Profile Detail Report for the file.

See Also

profile

More About

- “Profile to Improve Performance” on page 28-5
- “Measure Performance of Your Program” on page 28-2
- “Techniques to Improve Performance” on page 28-14

Techniques to Improve Performance

In this section...
“Environment” on page 28-14
“Code Structure” on page 28-14
“Programming Practices for Performance” on page 28-14
“Tips on Specific MATLAB Functions” on page 28-15

To speed up the performance of your code, consider these techniques.

Environment

Be aware of background processes that share computational resources and decrease the performance of your MATLAB code.

Code Structure

While organizing your code:

- Use functions instead of scripts. Functions are generally faster.
- Prefer local functions over nested functions. Use this practice especially if the function does not need to access variables in the main function.
- Use modular programming. To avoid large files and files with infrequently accessed code, split your code into simple and cohesive functions. This practice can decrease first-time run costs.

Programming Practices for Performance

Consider these programming practices to improve the performance of your code.

- Preallocate — Instead of continuously resizing arrays, consider preallocating the maximum amount of space required for an array. For more information, see “Preallocation” on page 28-17.
- Vectorize — Instead of writing loop-based code, consider using MATLAB matrix and vector operations. For more information, see “Vectorization” on page 28-19.

- Place independent operations outside loops — If code does not evaluate differently with each `for` or `while` loop iteration, move it outside of the loop to avoid redundant computations.
- Create new variables if data type changes — Create a new variable rather than assigning data of a different type to an existing variable. Changing the class or array shape of an existing variable takes extra time to process.
- Use short-circuit operators — Use short-circuiting logical operators, `&&` and `||` when possible. Short-circuiting is more efficient because MATLAB evaluates the second operand only when the result is not fully determined by the first operand. For more information, see [Logical Operators: Short Circuit](#).
- Avoid global variables — Minimizing the use of global variables is a good programming practice, and global variables can decrease performance of your MATLAB code.
- Avoid overloading built-ins — Avoid overloading built-in functions on any standard MATLAB data classes.
- Avoid using “data as code” — If you have large portions of code (for example, over 500 lines) that generate variables with constant values, consider constructing the variables and saving them in a MAT-file. Then you can load the variables instead of executing code to generate them.

Tips on Specific MATLAB Functions

Consider the following tips on specific MATLAB functions when writing performance critical code.

- Avoid clearing more code than necessary. Do not use `clear all` programmatically. For more information, see [clear](#).
- Avoid functions that query the state of MATLAB such as `inputname`, `which`, `whos`, `exist(var)`, and `dbstack`. Run-time introspection is computationally expensive.
- Avoid functions such as `eval`, `evalc`, `evalin`, and `feval(fname)`. Use the function `handle input` to `feval` whenever possible. Indirectly evaluating a MATLAB expression from text is computationally expensive.
- Avoid programmatic use of `cd`, `addpath`, and `rmpath`, when possible. Changing the MATLAB path during run time results in code recompilation.

See Also

More About

- “Measure Performance of Your Program” on page 28-2
- “Profile to Improve Performance” on page 28-5
- “Preallocation” on page 28-17
- “Vectorization” on page 28-19
- “Graphics Performance”

Preallocation

for and while loops that incrementally increase the size of a data structure each time through the loop can adversely affect performance and memory use. Repeatedly resizing arrays often requires MATLAB to spend extra time looking for larger contiguous blocks of memory, and then moving the array into those blocks. Often, you can improve code execution time by preallocating the maximum amount of space required for the array.

The following code displays the amount of time needed to create a scalar variable, `x`, and then to gradually increase the size of `x` in a `for` loop.

```
tic
x = 0;
for k = 2:1000000
    x(k) = x(k-1) + 5;
end
toc
```

Elapsed time is 0.301528 seconds.

If you preallocate a 1-by-1,000,000 block of memory for `x` and initialize it to zero, then the code runs much faster because there is no need to repeatedly reallocate memory for the growing data structure.

```
tic
x = zeros(1, 1000000);
for k = 2:1000000
    x(k) = x(k-1) + 5;
end
toc
```

Elapsed time is 0.011938 seconds.

Use the appropriate preallocation function for the kind of array you want to initialize:

- `zeros` for numeric arrays
- `cell` for character arrays

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 double values, and then by converting each element to `int8`. Creating the array as `int8` values saves time and memory. For example:

```
A = zeros(100, 'int8');
```

See Also

Related Examples

- “Reshaping and Rearranging Arrays”
- “Preallocate Memory for Cell Array” on page 12-18
- “Access Data Using Categorical Arrays” on page 8-30
- “Preallocate Arrays of Graphics Objects”
- “Construct Object Arrays”

More About

- “Techniques to Improve Performance” on page 28-14

Vectorization

In this section...

“Using Vectorization” on page 28-19

“Array Operations” on page 28-20

“Logical Array Operations” on page 28-22

“Matrix Operations” on page 28-23

“Ordering, Setting, and Counting Operations” on page 28-24

“Functions Commonly Used in Vectorization” on page 28-26

Using Vectorization

MATLAB is optimized for operations involving matrices and vectors. The process of revising loop-based, scalar-oriented code to use MATLAB matrix and vector operations is called *vectorization*. Vectorizing your code is worthwhile for several reasons:

- *Appearance*: Vectorized mathematical code appears more like the mathematical expressions found in textbooks, making the code easier to understand.
- *Less Error Prone*: Without loops, vectorized code is often shorter. Fewer lines of code mean fewer opportunities to introduce programming errors.
- *Performance*: Vectorized code often runs much faster than the corresponding code containing loops.

Vectorizing Code for General Computing

This code computes the sine of 1,001 values ranging from 0 to 10:

```
i = 0;
for t = 0:.01:10
    i = i + 1;
    y(i) = sin(t);
end
```

This is a vectorized version of the same code:

```
t = 0:.01:10;
y = sin(t);
```

The second code sample usually executes faster than the first and is a more efficient use of MATLAB. Test execution speed on your system by creating scripts that contain the code shown, and then use the `tic` and `toc` functions to measure their execution time.

Vectorizing Code for Specific Tasks

This code computes the cumulative sum of a vector at every fifth element:

```
x = 1:10000;
ylength = (length(x) - mod(length(x),5))/5;
y(1:ylength) = 0;
for n= 5:5:length(x)
    y(n/5) = sum(x(1:n));
end
```

Using vectorization, you can write a much more concise MATLAB process. This code shows one way to accomplish the task:

```
x = 1:10000;
xsums = cumsum(x);
y = xsums(5:5:length(x));
```

Array Operations

Array operators perform the same operation for all elements in the data set. These types of operations are useful for repetitive calculations. For example, suppose you collect the volume (V) of various cones by recording their diameter (D) and height (H). If you collect the information for just one cone, you can calculate the volume for that single cone:

```
V = 1/12*pi*(D^2)*H;
```

Now, collect information on 10,000 cones. The vectors D and H each contain 10,000 elements, and you want to calculate 10,000 volumes. In most programming languages, you need to set up a loop similar to this MATLAB code:

```
for n = 1:10000
    V(n) = 1/12*pi*(D(n)^2)*H(n);
end
```

With MATLAB, you can perform the calculation for each element of a vector with similar syntax as the scalar case:

```
% Vectorized Calculation
V = 1/12*pi*(D.^2).*H;
```

Note Placing a period (.) before the operators *, /, and ^, transforms them into array operators.

Array operators also enable you to combine matrices of different dimensions. This automatic expansion of size-1 dimensions is useful for vectorizing grid creation, matrix and vector operations, and more.

Suppose that matrix A represents test scores, the rows of which denote different classes. You want to calculate the difference between the average score and individual scores for each class. Using a loop, the operation looks like:

```
A = [97 89 84; 95 82 92; 64 80 99; 76 77 67; ...
     88 59 74; 78 66 87; 55 93 85];

mA = mean(A);
B = zeros(size(A));
for n = 1:size(A,2)
    B(:,n) = A(:,n) - mA(n);
end
```

A more direct way to do this is with `A - mean(A)`, which avoids the need of a loop and is significantly faster.

```
devA = A - mean(A)
```

```
devA =
```

```
    18    11     0
    16     4     8
   -15     2    15
    -3    -1   -17
     9   -19   -10
    -1   -12     3
   -24    15     1
```

Even though A is a 7-by-3 matrix and `mean(A)` is a 1-by-3 vector, MATLAB implicitly expands the vector as if it had the same size as the matrix, and the operation executes as a normal element-wise minus operation.

The size requirement for the operands is that for each dimension, the arrays must either have the same size or one of them is 1. If this requirement is met, then dimensions where one of the arrays has size 1 are expanded to be the same size as the corresponding

dimension in the other array. For more information, see “Compatible Array Sizes for Basic Operations” on page 2-19.

Another area where implicit expansion is useful for vectorization is if you are working with multidimensional data. Suppose you want to evaluate a function, F , of two variables, x and y .

$$F(x, y) = x \cdot \exp(-x^2 - y^2)$$

To evaluate this function at every combination of points in the x and y vectors, you need to define a grid of values. For this task you should avoid using loops to iterate through the point combinations. Instead, if one of the vectors is a column and the other is a row, then MATLAB automatically constructs the grid when the vectors are used with an array operator, such as $x+y$ or $x-y$. In this example, x is a 21-by-1 vector and y is a 1-by-16 vector, so the operation produces a 21-by-16 matrix by expanding the second dimension of x and the first dimension of y .

```
x = (-2:0.2:2)'; % 21-by-1
y = -1.5:0.2:1.5; % 1-by-16
F = x.*exp(-x.^2-y.^2); % 21-by-16
```

In cases where you want to explicitly create the grids, you can use the `meshgrid` and `ndgrid` functions.

Logical Array Operations

A logical extension of the bulk processing of arrays is to vectorize comparisons and decision making. MATLAB comparison operators accept vector inputs and return vector outputs.

For example, suppose while collecting data from 10,000 cones, you record several negative values for the diameter. You can determine which values in a vector are valid with the `>=` operator:

```
D = [-0.2 1.0 1.5 3.0 -1.0 4.2 3.14];
D >= 0
```

```
ans =
```

```
    0    1    1    1    0    1    1
```

You can directly exploit the logical indexing power of MATLAB to select the valid cone volumes, `Vgood`, for which the corresponding elements of `D` are nonnegative:

```
Vgood = V(D >= 0);
```

MATLAB allows you to perform a logical AND or OR on the elements of an entire vector with the functions `all` and `any`, respectively. You can throw a warning if all values of `D` are below zero:

```
if all(D < 0)
    warning('All values of diameter are negative.')
    return
end
```

MATLAB can also compare two vectors with compatible sizes, allowing you to impose further restrictions. This code finds all the values where `V` is nonnegative and `D` is greater than `H`:

```
V((V >= 0) & (D > H))
```

The resulting vector is the same size as the inputs.

To aid comparison, MATLAB contains special values to denote overflow, underflow, and undefined operators, such as `Inf` and `NaN`. Logical operators `isinf` and `isnan` exist to help perform logical tests for these special values. For example, it is often useful to exclude `NaN` values from computations:

```
x = [2 -1 0 3 NaN 2 NaN 11 4 Inf];
xvalid = x(~isnan(x))

xvalid =

     2    -1     0     3     2    11     4    Inf
```

Note `Inf == Inf` returns true; however, `NaN == NaN` always returns false.

Matrix Operations

When vectorizing code, you often need to construct a matrix with a particular size or structure. Techniques exist for creating uniform matrices. For instance, you might need a 5-by-5 matrix of equal elements:

```
A = ones(5,5)*10;
```

Or, you might need a matrix of repeating values:

```
v = 1:5;  
A = repmat(v,3,1)
```

```
A =
```

```
    1    2    3    4    5  
    1    2    3    4    5  
    1    2    3    4    5
```

The function `repmat` possesses flexibility in building matrices from smaller matrices or vectors. `repmat` creates matrices by repeating an input matrix:

```
A = repmat(1:3,5,2)  
B = repmat([1 2; 3 4],2,2)
```

```
A =
```

```
    1    2    3    1    2    3  
    1    2    3    1    2    3  
    1    2    3    1    2    3  
    1    2    3    1    2    3  
    1    2    3    1    2    3
```

```
B =
```

```
    1    2    1    2  
    3    4    3    4  
    1    2    1    2  
    3    4    3    4
```

Ordering, Setting, and Counting Operations

In many applications, calculations done on an element of a vector depend on other elements in the same vector. For example, a vector, x , might represent a set. How to iterate through a set without a `for` or `while` loop is not obvious. The process becomes much clearer and the syntax less cumbersome when you use vectorized code.

Eliminating Redundant Elements

A number of different ways exist for finding the redundant elements of a vector. One way involves the function `diff`. After sorting the vector elements, equal adjacent elements produce a zero entry when you use the `diff` function on that vector. Because `diff(x)`

produces a vector that has one fewer element than `x`, you must add an element that is not equal to any other element in the set. `NaN` always satisfies this condition. Finally, you can use logical indexing to choose the unique elements in the set:

```
x = [2 1 2 2 3 1 3 2 1 3];
x = sort(x);
difference = diff([x,NaN]);
y = x(difference~=0)
```

```
y =
```

```
     1     2     3
```

Alternatively, you could accomplish the same operation by using the `unique` function:

```
y=unique(x);
```

However, the `unique` function might provide more functionality than is needed and slow down the execution of your code. Use the `tic` and `toc` functions if you want to measure the performance of each code snippet.

Counting Elements in a Vector

Rather than merely returning the set, or subset, of `x`, you can count the occurrences of an element in a vector. After the vector sorts, you can use the `find` function to determine the indices of zero values in `diff(x)` and to show where the elements change value. The difference between subsequent indices from the `find` function indicates the number of occurrences for a particular element:

```
x = [2 1 2 2 3 1 3 2 1 3];
x = sort(x);
difference = diff([x,max(x)+1]);
count = diff(find([1,difference]))
y = x(find(difference))
```

```
count =
```

```
     3     4     3
```

```
y =
```

```
     1     2     3
```

The `find` function does not return indices for NaN elements. You can count the number of NaN and Inf values using the `isnan` and `isinf` functions.

```
count_nans = sum(isnan(x(:)));  
count_infs = sum(isinf(x(:)));
```

Functions Commonly Used in Vectorization

Function	Description
<code>all</code>	Determine if all array elements are nonzero or true
<code>any</code>	Determine if any array elements are nonzero
<code>cumsum</code>	Cumulative sum
<code>diff</code>	Differences and Approximate Derivatives
<code>find</code>	Find indices and values of nonzero elements
<code>ind2sub</code>	Subscripts from linear index
<code>ipermute</code>	Inverse permute dimensions of N-D array
<code>logical</code>	Convert numeric values to logicals
<code>meshgrid</code>	Rectangular grid in 2-D and 3-D space
<code>ndgrid</code>	Rectangular grid in N-D space
<code>permute</code>	Rearrange dimensions of N-D array
<code>prod</code>	Product of array elements
<code>repmat</code>	Repeat copies of array
<code>reshape</code>	Reshape array
<code>shiftdim</code>	Shift dimensions
<code>sort</code>	Sort array elements
<code>squeeze</code>	Remove singleton dimensions
<code>sub2ind</code>	Convert subscripts to linear indices
<code>sum</code>	Sum of array elements

See Also

More About

- “Array Indexing”
- “Techniques to Improve Performance” on page 28-14
- “Array vs. Matrix Operations” on page 2-13

External Websites

- MathWorks Newsletter: Matrix Indexing in MATLAB

Memory Usage

- “Strategies for Efficient Use of Memory” on page 29-2
- “Resolve “Out of Memory” Errors” on page 29-9
- “How MATLAB Allocates Memory” on page 29-12
- “Avoid Unnecessary Copies of Data” on page 29-21

Strategies for Efficient Use of Memory

In this section...

"Ways to Reduce the Amount of Memory Required" on page 29-2

"Using Appropriate Data Storage" on page 29-4

"How to Avoid Fragmenting Memory" on page 29-7

"Reclaiming Used Memory" on page 29-8

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.

File Type	Partial Loading
MAT-file	Load part of a variable by indexing into an object that you create with the <code>matfile</code> function.
Text	Use the <code>textscan</code> 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 <code>textscan</code> , MATLAB calculates the exact amount of memory required beforehand.

File Type	Partial Loading
Binary	You can use low-level binary file I/O functions, such as <code>fread</code> , to access parts of any file that has a known format. For binary files of an unknown format, try using memory mapping with the <code>mmapfile</code> function.
Image, HDF, Audio, and Video	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 for Import and Export”.

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 the `single` class, 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
<code>single</code>	4	Most math
<code>double</code>	8	All math
<code>logical</code>	1	Logical/conditional operations
<code>int8, uint8</code>	1	Arithmetic and some simple functions
<code>int16, uint16</code>	2	Arithmetic and some simple functions
<code>int32, uint32</code>	4	Arithmetic and some simple functions
<code>int64, int64</code>	8	Arithmetic and some simple functions

Reduce the Amount of Overhead When Storing Data

MATLAB arrays (implemented internally as `mxAArrays`) 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 `mxAArrays` (e.g., scalars). The `whos` command lists the memory used by variables, but does not include this overhead.

Because simple numeric arrays (comprising one `mxAArray`) 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 `mxAArrays` 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 “Array Headers” on page 29-14). 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 following example compares the space required for storage of an array of mainly zeros:

```
A = eye(1000);           % Full matrix with ones on the diagonal
As = sparse(A);         % Sparse matrix with only nonzero elements
whos
  Name      Size      Bytes  Class  Attributes
  A         1000x1000  8000000  double
  As        1000x1000   24008   double  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 might 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 “Preallocation” on page 28-17.

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 might 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
```

Resolve “Out of Memory” Errors

In this section...

“General Suggestions for Reclaiming Memory” on page 29-9

“Increase System Swap Space” on page 29-10

“Set the Process Limit on Linux Systems” on page 29-10

“Disable Java VM on Linux Systems” on page 29-10

“Free System Resources on Windows Systems” on page 29-11

General Suggestions for Reclaiming Memory

The MATLAB software is a 64-bit application that runs on 64-bit operating systems. It generates an **Out of Memory** message whenever it requests a segment of memory from the operating system that is larger than what is available. When you see the **Out of Memory** message, use any of the techniques discussed under “Strategies for Efficient Use of Memory” on page 29-2 to help optimize the available memory including:

- Reducing required memory
- Selecting appropriate data storage
- Using contiguous memory
- Reclaiming used memory

If the **Out of Memory** message still appears, you can try any of the following:

- If possible, reduce the size of your data. For example, break large matrices into several smaller matrices so that less memory is used at any one time.
- If you have large files and data sets, see “Large Files and Big Data”.
- Make sure that there are no external constraints on the memory accessible to MATLAB. On Linux® systems, use the `limit` command to investigate.
- Increase the size of the swap file. We recommend that you configure your system with twice as much swap space as you have RAM. For more information, see “Increase System Swap Space” on page 29-10.
- Add more memory to the system.

Increase System Swap Space

The total memory available to applications on your computer is composed of physical memory (RAM), plus a page file, or swap file, on disk. The swap file can be very large (for example, 512 terabytes on 64-bit Windows). The operating system allocates the virtual memory for each process to physical memory or to the swap file, depending on the needs of the system and other processes.

Most systems enable you to control the size of your swap file. The steps involved depend on your operating system.

- 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 — Change your swap space by using the `mkswap` and `swapon` commands. For more information, at the Linux prompt type `man` followed by the command name.

There is no interface for directly controlling the swap space on macOS systems.

Set the Process Limit on Linux Systems

The process limit is the maximum amount of virtual memory a single process (or application) can address. The process limit must be large enough to accommodate:

- All the data to process
- MATLAB program files
- The MATLAB executable itself
- Additional state information

The 64-bit operating systems support a process limit of 8 terabytes. On Linux systems, see the `ulimit` command to view and set user limits including virtual memory.

Disable Java VM on Linux Systems

On Linux systems, if you start MATLAB without the Java JVM™, you can increase the available workspace memory by approximately 400 megabytes. To start MATLAB without Java JVM, use the command-line option `-nojvm`. This option also increases the size of the largest contiguous memory block by about the same. By increasing the largest contiguous memory block, you increase the largest possible matrix size.

Using `-nojvm` comes with a penalty in that you 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.

Free System Resources on Windows Systems

There are no MATLAB functions to manipulate the way MATLAB handles Microsoft Windows system resources. Windows systems use these resources to track fonts, windows, and screen objects. For example, using multiple figure windows, multiple fonts, or several UI controls can deplete resources. One way to free up system resources is to close all inactive windows. Windows system icons still use resources.

If total system memory is the limiting factor, shutting down other applications and services can help (for example, using `msconfig` on Windows systems). However, the process limit is usually the main limiting factor.

See Also

memory

Related Examples

- “Strategies for Efficient Use of Memory” on page 29-2
- “Large Files and Big Data”
- “Java Heap Memory Preferences”

How MATLAB Allocates Memory

In this section...
“Memory Allocation for Arrays” on page 29-12
“Data Structures and Memory” on page 29-16

Memory Allocation for Arrays

The following topics 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 29-12
- “Copying Arrays” on page 29-13
- “Array Headers” on page 29-14
- “Function Arguments” on page 29-15

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 “Preallocation” on page 28-17.

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.

This example uses the `memory` function to demonstrate how MATLAB handles copying arrays. `memory` is available only on Windows systems.

Start by creating a simple script `memUsed.m` to display how much memory is 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 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 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 (that is, 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 29-16.

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 might 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 contains your data without overflowing. For more information, see “Numeric Types”.

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 an 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 might 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 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:

```
(header_size x number_of_cells) + 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 x 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 vector to `Address` and a 12-character vector 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-500 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:

```
structure = fields x ((60 x array elements) + 64) =  
            4 x ((60 x 30) + 64) = 7,456 bytes
```

```
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
```

Avoid Unnecessary Copies of Data

In this section...

“Passing Values to Functions” on page 29-21

“Why Pass-by-Value Semantics” on page 29-23

“Handle Objects” on page 29-24

Passing Values to Functions

When calling a function with input arguments, MATLAB copies the values from the calling function’s workspace into the parameter variables in the function being called. However, MATLAB applies various techniques to avoid making copies of these values when it is not necessary.

MATLAB does not provide a way to define a reference to a value, as in languages like C++. Instead, MATLAB allows multiple output as well as multiple input parameters so that you know what values are going into a function and what values are coming out of the function.

Copy-on-Write

If a function does not modify an input argument, MATLAB does not make a copy of the values contained in the input variable.

For example, suppose that you pass a large array to a function.

```
A = rand(1e7,1);  
B = f1(A);
```

The function `f1` multiplies each element in the input array `X` by `1.1` and assigns the result to the variable `Y`.

```
function Y = f1(X)  
    Y = X.*1.1; % X is a shared copy of A  
end
```

Because the function does not modify the input values, the local variable `X` and the variable `A` in the caller’s workspace share the data. After `f1` executes, the values assigned to `A` have not changed. The variable `B` in the caller’s workspace contains the result of the

element-wise multiplication. The input is passed by value. However, no copy is made when calling `f1`.

The function `f2` does modify its local copy of the input variable, causing the local copy to be unshared with input `A`. The value of `X` in the function is now an independent copy of the input variable `A` in the caller's workspace. When `f2` returns the result to the caller's workspace, the local variable `X` is destroyed.

```
A = rand(1e7,1);
B = f2(A);

function Y = f2(X)
    X = X.*1.1; % X is an independent copy of A
    Y = X;     % Y is a shared copy of X
end
```

Passing Inputs as MATLAB Expressions

You can use the value returned from a function as an input argument to another function. For example, use the `rand` function to create the input for the function `f2` directly.

```
B = f2(rand(1e7,1));
```

The only variable holding the value returned by `rand` is the temporary variable `X` in the workspace of the function `f2`. There is no shared or independent copy of these values in the caller's workspace. Directly passing function outputs saves the time and memory required to create a copy of the input values in the called function. This approach makes sense when the input values are not used again.

Assigning In-Place

You can assign the output of a function to the same variable as the input when you do not need to preserve the original input values.

```
A = f2(A);
```

Reassignment to the same variable name follows the copy-on-write behavior described previously: modifying the input variable values results in a temporary copy of those values. However, MATLAB can apply memory optimizations under certain conditions.

Consider the following example. The `memoryOptimization` function creates a large array of random numbers in the variable `A`. Then it calls the local function `fLocal`, passing `A` as the input, and assigning the output of the local function to the same variable name.

Because the call to the local function, `A = fLocal(A)`, assigns the output to the variable `A`, MATLAB does not need to preserve the original value of `A` during execution of the function. Within `fLocal`, the input `X` has the only copy of the value originally held by `A`.

Therefore, modifications made to `X` inside `fLocal` do not result in a copy of the data. The assignment, `X = X.*1.1`, modifies `X` in place, without allocating a new array for the result of the multiplication.

```
function memoryOptimization
    A = rand(1e7,1);
    A = fLocal(A);
end
function X = fLocal(X)
    X = X.*1.1;
end
```

Eliminating the copy in the local function saves memory and improves execution speed for large arrays. However, MATLAB cannot apply memory optimization when it is possible to use the variable after the function throws an error. Therefore, this optimization is not applied in scripts, on the command line, in calls to `eval`, or to code inside `try/catch` blocks.

Also, MATLAB does not apply memory optimization when the original variable can be accessed directly during execution of the called function. For example, if `fLocal` was a nested function, MATLAB could not apply the optimization because variables can be shared with the parent function.

Finally, MATLAB does not apply memory optimization when the assigned variable is declared as global or persistent.

Why Pass-by-Value Semantics

MATLAB uses pass-by-value semantics when passing arguments to functions and returning values from functions. In some cases, pass-by-value results in copies of the original values being made in the called function. However, pass-by-value semantics provides certain advantages.

When calling functions, you know that the input variables are not modified in the caller's workspace. Therefore, you do not need to make copies of inputs inside a function or at a call site just to guard against the possibility that these values might be modified. Only the variables assigned to returned values are modified.

Also, you avoid the possibility of corrupting workspace variables if an error occurs within a function that has been passed a variable by reference.

Handle Objects

There are special kinds of objects called handles. All variables that hold copies of the same handle can access and modify the same underlying object. Handle objects are useful in specialized circumstances where an object represents a physical object such as a window, plot, device, or person rather than a mathematical object like a number or matrix.

Handle objects derive from the `handle` class, which provides functionality such as events and listeners, destructor methods, and support for dynamic properties.

For more information about values and handles, see “Comparison of Handle and Value Classes” and “Which Kind of Class to Use”.

See Also

`handle`

Related Examples

- “Handle Object Behavior”
- “Avoid Copies of Arrays in MEX Functions”
- “Strategies for Efficient Use of Memory” on page 29-2

Custom Help and Documentation

- “Create Help for Classes” on page 30-2
- “Check Which Programs Have Help” on page 30-9
- “Create Help Summary Files — Contents.m” on page 30-12
- “Customize Code Suggestions and Completions” on page 30-15
- “Display Custom Documentation” on page 30-27
- “Display Custom Examples” on page 30-37

Create Help for Classes

In this section...

“Help Text from the doc Command” on page 30-2

“Custom Help Text” on page 30-3

Help Text from the doc Command

When you use the `doc` command to display help for a class, MATLAB automatically displays information that it derives from the class definition.

For example, create a class definition file named `someClass.m` with several properties and methods, as shown.

```
classdef someClass
    % someClass Summary of this class goes here
    % Detailed explanation goes here

    properties
        One    % First public property
        Two    % Second public property
    end
    properties (Access=private)
        Three % Do not show this property
    end

    methods
        function obj = someClass
            % Summary of constructor
        end
        function myMethod(obj)
            % Summary of myMethod
            disp(obj)
        end
    end
    methods (Static)
        function myStaticMethod
            % Summary of myStaticMethod
        end
    end
end
```

View the help text and the details from the class definition using the `doc` command.

```
doc someClass
```

The screenshot shows the MATLAB File Help interface for a class named 'someClass'. At the top, there is a navigation bar with 'MATLAB File Help: someClass', a link to 'View code for someClass', and a link to 'Default Topics'. Below this, the class name 'someClass' is displayed in a large, bold, red font. Underneath, there is a summary of the class followed by a detailed explanation. The 'Class Details' section lists 'Sealed' as false and 'Construct on load' as false. The 'Constructor Summary' section includes a link to 'someClass' and a summary of the constructor. The 'Property Summary' section lists two public properties, 'One' and 'Two'. The 'Method Summary' section lists two methods, 'myMethod' and 'myStaticMethod', with the latter being static.

Custom Help Text

You can add information about your classes that both the `doc` command and the `help` command include in their displays. The `doc` command displays the help text at the top of the generated HTML pages, above the information derived from the class definition. The `help` command displays the help text in the Command Window. For details, see:

- “Classes” on page 30-4
- “Methods” on page 30-5
- “Properties” on page 30-5
- “Enumerations” on page 30-6

- “Events” on page 30-7

Classes

Create help text for classes by including comments on lines immediately after the `classdef` statement in a file. For example, create a file named `myClass.m`, as shown.

```
classdef myClass
    % myClass Summary of myClass
    % This is the first line of the description of myClass.
    % Descriptions can include multiple lines of text.
    %
    % myClass Properties:
    %     a - Description of a
    %     b - Description of b
    %
    % myClass Methods:
    %     doThis - Description of doThis
    %     doThat - Description of doThat

    properties
        a
        b
    end

    methods
        function obj = myClass
        end
        function doThis(obj)
        end
        function doThat(obj)
        end
    end
end
```

Lists and descriptions of the properties and methods in the initial comment block are optional. If you include comment lines containing the class name followed by `Properties` or `Methods` and a colon (:), then MATLAB creates hyperlinks to the help for the properties or methods.

View the help text for the class in the Command Window using the `help` command.

```
help myClass
```

```
myClass Summary of myClass
This is the first line of the description of myClass.
Descriptions can include multiple lines of text.
```

```
myClass Properties:
  a - Description of a
  b - Description of b
```

```
myClass Methods:
  doThis - Description of doThis
  doThat - Description of doThat
```

Methods

Create help for a method by inserting comments immediately after the function definition statement. For example, modify the class definition file `myClass.m` to include help for the `doThis` method.

```
function doThis(obj)
% doThis Do this thing
% Here is some help text for the doThis method.
%
% See also DOTHAT.

disp(obj)
end
```

View the help text for the method in the Command Window using the `help` command. Specify both the class name and method name, separated by a dot.

```
help myClass.doThis
```

```
doThis Do this thing
Here is some help text for the doThis method.

See also doThat.
```

Properties

There are two ways to create help for properties:

- Insert comment lines above the property definition. Use this approach for multiline help text.
- Add a single-line comment next to the property definition.

Comments above the definition have precedence over a comment next to the definition.

For example, modify the property definitions in the class definition file `myClass.m`.

```
properties
    a          % First property of myClass

    % b - Second property of myClass
    % The description for b has several
    % lines of text.
    b          % Other comment
end
```

View the help for properties in the Command Window using the `help` command. Specify both the class name and property name, separated by a dot.

```
help myClass.a
```

```
a - First property of myClass
```

```
help myClass.b
```

```
b - Second property of myClass
The description for b has several
lines of text.
```

Enumerations

Like properties, there are two ways to create help for enumerations:

- Insert comment lines above the enumeration definition. Use this approach for multiline help text.
- Add a single-line comment next to the enumeration definition.

Comments above the definition have precedence over a comment next to the definition.

For example, create an enumeration class in a file named `myEnumeration.m`.

```
classdef myEnumeration
    enumeration
        uno,          % First enumeration

        % DOS - Second enumeration
        % The description for DOS has several
        % lines of text.
```

```

        dos          % A comment (not help text)
    end
end

```

View the help in the Command Window using the `help` command. Specify both the class name and enumeration member, separated by a dot.

```
help myEnumeration.uno
```

```
uno - First enumeration
```

```
help myEnumeration.dos
```

```

dos - Second enumeration
The description for dos has several
lines of text.

```

Events

Like properties and enumerations, there are two ways to create help for events:

- Insert comment lines above the event definition. Use this approach for multiline help text.
- Add a single-line comment next to the event definition.

Comments above the definition have precedence over a comment next to the definition.

For example, create a class in a file named `hasEvents.m`.

```

classdef hasEvents < handle
    events
        Alpha      % First event

        % Beta - Second event
        % Additional text about second event.
        Beta      % (not help text)
    end

    methods
        function fireEventAlpha(h)
            notify(h, 'Alpha')
        end

        function fireEventBeta(h)
            notify(h, 'Beta')
        end
    end
end

```

```
        end
    end
end
```

View the help in the Command Window using the `help` command. Specify both the class name and event, separated by a dot.

```
help hasEvents.Alpha
```

```
Alpha - First event
```

```
help hasEvents.Beta
```

```
Beta - Second event
Additional text about second event.
```

See Also

`doc | help`

More About

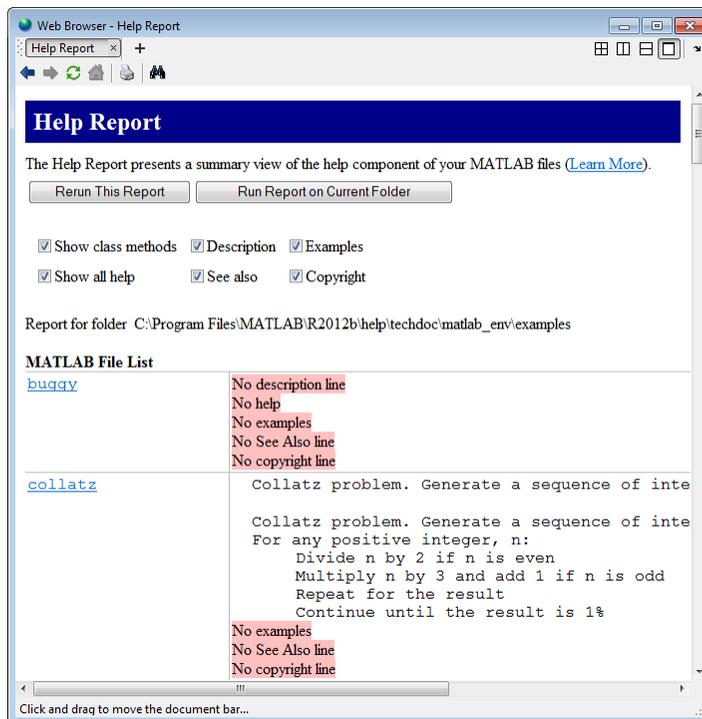
- “Role of Classes in MATLAB”
- “User-Defined Classes”

Check Which Programs Have Help

To determine which of your programs files have help text, you can use the Help Report.

In the Help Report, you specify a set of help components for which you want to search, such as examples or See Also lines. For each file searched, MATLAB displays the help text for the components it finds. Otherwise, MATLAB displays a highlighted message to indicate that the component is missing.

To generate a Help Report, in the Current Folder browser, navigate to the folder you want to check, click , and then select **Reports > Help Report**. The Help Report displays in the MATLAB web browser.



Note You cannot run reports when the path is a UNC (Universal Naming Convention) path; that is, a path that starts with `\\`. Instead, use an actual hard drive on your system, or a mapped network drive.

This table describes the available options for Help Reports.

Help Report Option	Description
Show class methods	Include methods in the report. If you do not select this option, then the report includes results for classes, but not for methods within a class definition file.
Show all help	<p>Display all help text found in each file. If you also select individual help components, such as Description, then help text appears twice in the report for each file: once for the overall help text, and once for the component.</p> <p>If your program has the same name as other programs on the MATLAB search path, then the <code>help</code> command generates a list of those overloaded items. MATLAB automatically adds links to the help for those items.</p>
Description	Check for an initial, nonempty comment line in the file. This line is sometimes called the H1 line.
Examples	Check for examples in the help text. The Help Report performs a case-insensitive search for a help line with a single-word variant of <code>example</code> . The report displays that line and subsequent nonblank comment lines, along with the initial line number.
See Also	<p>Check for a line in the help that begins with the words <code>See also</code>. The report displays the text and the initial line number.</p> <p>If the programs listed after <code>See also</code> are on the search path, then the <code>help</code> command generates hyperlinks to the help for those programs. The Help Report indicates when a program in the <code>See also</code> line is not on the path.</p>
Copyright	<p>Check for a comment line in the file that begins with the word <code>Copyright</code>. When there is a copyright line, the report also checks whether the end year is current. The date check requires that the copyright line includes either a single year (such as 2012) or a range of years with no spaces (such as 2001-2012).</p> <p>The recommended practice is to include a range of years from the year you created the file to the current year.</p>

See Also

Related Examples

- “Add Help for Your Program” on page 20-6
- “Create Help Summary Files — Contents.m” on page 30-12

Create Help Summary Files – Contents.m

In this section...

“What Is a Contents.m File?” on page 30-12

“Create a Contents.m File” on page 30-12

“Check an Existing Contents.m File” on page 30-13

What Is a Contents.m File?

A `Contents.m` file provides a summary of the programs in a particular folder. The `help`, `doc`, and `ver` functions refer to `Contents.m` files to display information about folders.

`Contents.m` files contain only comment lines. The first two lines are headers that describe the folder. Subsequent lines list the program files in the folder, along with their descriptions. Optionally, you can group files and include category descriptions. For example, view the functions available in the `codetools` folder:

`help codetools`

```
Commands for creating and debugging code
MATLAB Version 9.3 (R2017b) 24-Jul-2017
```

```
Editing and publishing
```

```
edit           - Edit or create a file
grabcode      - Copy MATLAB code from published HTML
mlint         - Check files for possible problems
publish       - Publish file containing cells to output file
snapnow      - Force snapshot of image for published document
```

```
Directory tools
```

```
mlintrpt      - Run mlint for file or folder, reporting results in browser
visdiff       - Compare two files (text, MAT, or binary) or folders
```

```
...
```

If you do *not* want others to see a summary of your program files, place an empty `Contents.m` file in the folder. An empty `Contents.m` file causes `help foldername` to report `No help found for foldername`. Without a `Contents.m` file, the `help` and `doc` commands display a generated list of all program files in the folder.

Create a Contents.m File

When you have a set of existing program files in a folder, the easiest way to create a `Contents.m` file is to use the Contents Report. The primary purpose of the Contents

Report is to check that an existing `Contents.m` file is up-to-date. However, it also checks whether `Contents.m` exists, and can generate a new file based on the contents of the folder. Follow these steps to create a file:

- 1 In the Current Folder browser, navigate to the folder that contains your program files.
- 2 Click , and then select **Reports > Contents Report**.
- 3 In the report, where prompted to make a `Contents.m` file, click **yes**. The new file includes the names of all program files in the folder, using the description line (the first nonempty comment line) whenever it is available.
- 4 Open the generated file in the Editor, and modify the file so that the second comment line is in this form:

```
% Version xxx dd-mmm-yyyy
```

Do not include any spaces in the date. This comment line enables the `ver` function to detect the version information.

Note MATLAB does not include live scripts or functions when creating a Contents Report.

Check an Existing Contents.m File

Verify whether your `Contents.m` file reflects the current contents of the folder using the Contents Report, as follows:

- 1 In the Current Folder browser, navigate to the folder that contains the `Contents.m` file.
- 2 Click , and then select **Reports > Contents Report**.

Note You cannot run reports when the path is a UNC (Universal Naming Convention) path; that is, a path that starts with `\\`. Instead, use an actual hard drive on your system, or a mapped network drive.

The Contents Report performs the following checks.

Check Whether the Contents.m File...	Details
Exists	If there is no Contents.m file in the folder, you can create one from the report.
Includes all programs in the folder	Missing programs appear in gray highlights. You do not need to add programs that you do not want to expose to end users.
Incorrectly lists nonexistent files	Listed programs that are not in the folder appear in pink highlights.
Matches the program file descriptions	The report compares file descriptions in Contents.m with the first nonempty comment line in the corresponding file. Discrepancies appear in pink highlights. You can update either the program file or the Contents.m file.
Uses consistent spacing between file names and descriptions	Fix the alignment by clicking fix spacing at the top of the report.

You can make all the suggested changes by clicking **fix all**, or open the file in the Editor by clicking **edit Contents.m**.

See Also

doc | help | ver

Customize Code Suggestions and Completions

To customize code suggestions and completions for your functions, provide MATLAB with information about your function signatures. Function signatures describes the acceptable syntaxes and allowable data types for a function. MATLAB uses this information to improve interactive features such as tab completion and function hints. Define this function information in a JSON-formatted file called `functionSignatures.json`.

For MATLAB to detect the function signature information, you must place `functionSignatures.json` in the folder that contains the function code. You can define signatures for multiple functions in the same file.

The `functionSignatures.json` file contains a single JSON object. JSON uses braces to define objects, and refers to objects as collections of name and value pairs. Since these terms are overloaded in context of function signatures, "property" is used instead of "name." The JSON object in `functionSignatures.json` contains an optional schema version and a list of function objects. Each function object contains a list of signature objects, and each signature object contains an array of argument objects. JSON uses brackets to define arrays.

```
{
  "_schemaVersion": "<major#>.<minor#>.<patch#>",
  "functionName1": {
    {
      "inputs": [
        { "name": "A", "kind": "required", "type": ["numeric"] },
        { "name": "dim", "kind": "ordered", "type": ["numeric", "integer", "scalar", ">0"] },
        { "name": "nonflag", "kind": "flag", "type": ["char", "choices-{'includenan','excludenan'}"] }
      ]
    }
  },
  "functionName2": {
    {
      "inputs": [
        { "name": "A", "kind": "required", "type": ["numeric"] },
        { "name": "B", "kind": "required", "type": ["numeric", "scalar"] }
      ]
    }
  },
  ...
}
```

To specify the optional schema version use `_schemaVersion` as the first property and the version number as its value. Specify the version number as a JSON string in the

format *major#.minor#.patch#*, with each number specified as a nonnegative integer. The current schema version is 1.0.0. If the file does not specify a schema version, MATLAB assumes version 1.0.0.

If `functionSignatures.json` contains syntax errors, MATLAB displays an error message in the Command Window when it reads the file. Use the `validateFunctionSignaturesJSON` function to validate the `functionSignatures.json` file against the JSON schema and the MATLAB function signature schema.

Function Objects

To define information for a function, create a property that is the same as the function name. Its value is a signature object on page 30-16.

```
{
  "functionName1": { signatureObj1 },
  "functionName2": { signatureObj2 }
}
```

To define information for a class method or package function, use the full name of the function or method. Example properties are "MyClass.myMethod" or "myPackage.myFunction". You can define multiple function signatures for the same function by defining multiple function objects with the same property (function name). For more information, see "Multiple Signatures" on page 30-24.

Signature Objects

A signature object defines the input and output arguments and supported platforms for the function. The value of each property, except for the `platforms` property, is an array of argument objects on page 30-17.

```
{
  "functionName1":
  {
    "inputs": [ argumentObj1, argumentObj2 ]
  }
}
```

Each signature can include the following properties.

Property	Description	JSON Data Type of Value
inputs	List of function input arguments. MATLAB uses this property for code suggestions and completions.	Array of argument objects
outputs	List of function output arguments. MATLAB does not currently use this property for code suggestions and completions.	Array of argument objects
platforms	List of platforms that support the function. MATLAB does not present custom code suggestions and completions if the platform does not support the function. The default is all platforms. Elements of the list must match an <code>archstr</code> returned from the <code>computer</code> function. The list can be inclusive or exclusive, but not both. Example values are "win64,maci64" or "-win64,-maci64".	String of comma-separated values

Argument Objects

Argument objects define the information for each of the input and output arguments.

```
{
  "functionName1":
  {
    "inputs":
    [
      {"name":"in1", "kind":"required", "type":["numeric"]},
      {"name":"in2", "kind":"required", "type":["numeric","integer","scalar"]}
    ]
  }
}
```

The order that the inputs appear in the JSON file is significant. For example, in a call to the `functionName1` function, `in1` must appear before `in2`.

Each argument object can include the following properties.

name - Name of argument

The name of the input or output argument, specified as a JSON string. This property and value is required. The name property does not need to match the argument name in the source code, but it is a best practice for it to match any help or documentation.

Example: "name": "myArgumentName"

kind - Kind of argument

The kind of argument, specified as a JSON string with one of the following values. MATLAB uses the value of the kind property to determine if and when to display the arguments within the function signature.

Value	Description
required	Argument is required, and its location is relative to other required arguments in the signature object.
ordered	Argument is optional, and its location is relative to the required and preceding optional arguments in the signature object.
namevalue	Argument is an optional name-value pair. Name-value pair arguments occur at the end of a function signature, but the pairs can be specified in any order.

Arguments that are `required` and `ordered` appear first in the function signature, and are followed by any `namevalue` arguments.

`required`, `ordered`, and `namevalue` arguments are most common. You can also specify the following values for `kind`.

- `positional` - Argument is optional if it occurs at the end of the argument list, but becomes required to specify a subsequent positional argument. Any `positional` arguments must appear with the `required` and `ordered` arguments, before any `namevalue` arguments.
- `flag` - Argument is an optional, constant string, typically used as a switch. For example, `'ascend'` or `'descend'`. Flags occur at the end of a function signature. All `flag` arguments must appear before any `namevalue` arguments.
- `properties` - Argument is optional and is used to specify public, settable properties of a different MATLAB class. Indicate the class using the argument object `type` property. In code suggestions, these properties appear as name-value pairs. Any `properties` arguments must be the last argument in the signature.

Example: "kind": "required" or "kind": "namevalue"

type - Class and/or attributes of argument

Class or attributes of the argument, specified as a JSON string, list, or list of lists.

The `type` property can define what class the argument is and what attributes the argument must have.

- To match one class or attribute, use a single JSON string. For example, if an argument must be numeric, then specify `"type": "numeric"`.
- To match all classes or attributes, use a list of JSON strings. For example, if an argument must be both numeric and positive, then specify `"type": ["numeric", ">=0"]`.
- To match any of multiple classes or attributes, use a list of lists of JSON strings. For the inner lists, MATLAB uses a logical AND of the values. For the outer list, MATLAB uses a logical OR of the values. For example, if an argument must be either a positive number or a `containers.Map` object, then specify `"type": [{"numeric", ">=0"}, ["containers.Map"]]`.

Value	Argument Description
" <i>classname</i> "	Must be an object of class <i>classname</i> , where <i>classname</i> is the name of the class returned by the <code>class</code> function. For example, "double" or "function_handle".
"choices= <i>expression</i> "	Must be a case-insensitive match to one of the specified choices. <i>expression</i> is any valid MATLAB expression that returns a cell of character vectors, string array, or cell of integer values. For example, "choices={'on', 'off'}" or "choices={8, 16, 24}". <i>expression</i> can refer by name to other input arguments that appear in the argument list. Since <i>expression</i> is evaluated at run time, allowable choices can vary dynamically with the value of other input arguments.
"file= <i>*.ext, ...</i> "	Must be a string or character vector that names an existing file with the specified extension. Filenames are relative to the current working folder. For example, to allow all <code>.m</code> and <code>.mlx</code> files in the current folder, use "file= <code>*.m, *.mlx</code> ". To match all files in the current folder, use "file".

Value	Argument Description
"folder"	Must be a string or character vector that is the name of an existing folder relative to the current working folder.
"matlabpathfile= *.ext,..."	Must be a string or character vector that names an existing file on the MATLAB path. This value requires at least one file extension. For example, to allow all .mat files on the path, use "matlabpathfile=*.mat".
"size=size1, size 2,..., sizeN"	Must match size constraints. This value requires two or more dimensions. Each size dimension must be either a positive integer that indicates the allowable size of the dimension, or a colon to allow any size. For example, "size=2, :, 2" constrains the argument to have a size of 2 in the 1st and 3rd dimensions.
"numel=integerVa lue"	Must have a specified number of elements.
"nrows=integerVa lue"	Must have a specified number of rows.
"ncols=integerVa lue"	Must have a specified number of columns.
"numeric"	Must be numeric. A numeric value is one for which the isa function with the 'numeric' class category returns true.
"logical"	Must be numeric or logical.
"real"	Must be a real-valued numeric or a character or logical value.
"scalar"	Must be scalar.
"integer"	Must be an integer of type double.
"square"	Must be a square matrix.
"vector"	Must be a column or row vector.
"column"	Must be a column vector.
"row"	Must be a row vector.
"2d"	Must be 2-dimensional.
"3d"	Must have no more than three dimensions.
"sparse"	Must be sparse.
"positive"	Must be greater than zero.

Value	Argument Description
"> <i>expression</i> "	Must be numeric and satisfy the inequality. <i>expression</i> must return a full scalar double.
">= <i>expression</i> "	
"< <i>expression</i> "	
"<= <i>expression</i> "	
@(<i>args</i>) <i>expression</i>	Must satisfy the function handle. For a value to satisfy the function handle, the handle must evaluate to <code>true</code> .

repeating - Specify argument multiple times

Indicator that an argument can be specified multiple times, specified as a JSON `true` or `false` (without quotes). The default is `false`. If specified as `true`, the argument or set of arguments (tuple) can be specified multiple times. A required repeating argument must appear one or more times, and an optional repeating argument can appear zero or more times.

Example: `"repeating":true`

purpose - Description of argument

Description of argument, specified as a JSON string. Use this property to communicate the purpose of the argument.

Example: `"purpose":"Product ID"`

For more complicated function signatures, the following properties are available for each argument object.

platforms - List of supported platforms

List of platforms that support the argument, specified as a JSON string. The default is all platforms. Elements of the list must match an `archstr` returned from the `computer` function. The list can be inclusive or exclusive, but not both.

Example: `"platforms":"win64,maci64"` or `"platforms": "-maci64"`

tuple - Definition of set of arguments

Definition of a set of arguments that must always appear together, specified as a list of argument objects. This property is only used to define sets of multiple repeating

arguments. For these function signatures, define tuples and set the `repeating` property to `true`.

mutuallyExclusiveGroup - Definition of set of exclusive arguments

Definition of a set of sets of arguments that cannot be used together, specified as a list of argument objects. This property is used to provide information about functions with multiple function signatures. However, typically it is easier to define multiple function signatures using multiple function objects. For more information, see “Multiple Signatures” on page 30-24.

Create Function Signature File

This example describes how to create custom code suggestions and completions for a function.

Create a function whose signature you will describe in a JSON file in later steps. The following function accepts:

- Two required arguments
- One optional positional argument via `varargin`
- Two optional name-value pair arguments via `varargin`

`myFunc` is presented to demonstrate code suggestions and does not include argument checking.

```
% myFunc Example function
% This function is called with any of these syntaxes:
%
% myFunc(in1, in2) accepts 2 required arguments.
% myFunc(in1, in2, in3) also accepts an optional 3rd argument.
% myFunc(__, NAME, VALUE) accepts one or more of the following name-value pair
% arguments. This syntax can be used in any of the previous syntaxes.
%     * 'NAME1' with logical value
%     * 'NAME2' with 'Default', 'Choice1', or 'Choice2'
function myFunc(reqA, reqB, varargin)
    % Initialize default values
    NV1 = true;
    NV2 = 'Default';
    posA = [];

    if nargin > 3
        if rem(nargin,2)
            posA = varargin{1};
            V = varargin(2:end);
        else
            V = varargin;
        end
        for n = 1:2:size(V,2)
```

```

switch V{n}
    case 'Name1'
        NV1 = V{n+1};
    case 'Name2'
        NV2 = V{n+1}
    otherwise
        error('Error.')
end
end
end
end
end

```

In the same folder as `myFunc`, create the following function signature description in a file called `functionSignatures.json`. The input names do not match the names in the body of `myFunc`, but are consistent with the help text.

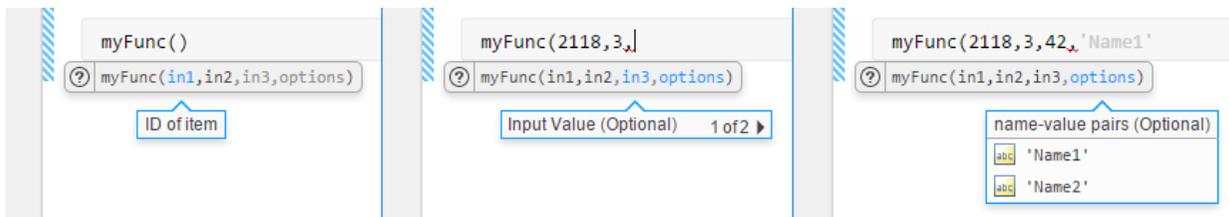
```

{
  "_schemaVersion": "1.0.0",
  "myFunc":
  {
    "inputs":
    [
      {"name": "in1", "kind": "required", "type": ["numeric"], "purpose": "ID of item"},
      {"name": "in2", "kind": "required", "type": ["numeric"], "purpose": "# Items"},
      {"name": "in3", "kind": "ordered", "type": ["numeric"], "purpose": "Input Value"},
      {"name": "Name1", "kind": "namevalue", "type": ["logical", "scalar"], "purpose": "Option"},
      {"name": "Name2", "kind": "namevalue", "type": ["char", "choices={ 'Default', 'Choice1', 'Choice2' }"]}
    ]
  }
}

```

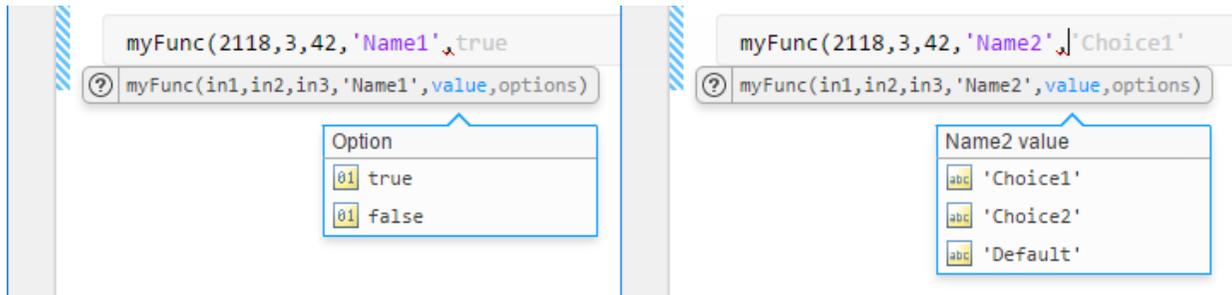
MATLAB uses this function signature description to inform code suggestions and completion.

To experiment with code suggestions, start to call the function from a live script and observe the suggestions. For example, the names and purposes from the JSON file appear. MATLAB indicates when arguments are optional and if there are multiple suggestions available (such as the third positional argument or a name-value pair). Name-value pairs options are listed.



When adding a name-value pair argument to the function call, MATLAB presents the choices from the JSON file. Since `'Name1'` is defined as a logical scalar, MATLAB

populates the choices automatically (true or false). MATLAB takes the three values for the 'Name2' argument from the JSON file.



Multiple Signatures

If a function has many syntaxes, it can be helpful for code suggestions to group syntaxes as multiple function signatures (regardless of the implementation of the function). To provide code suggestions and completions for multiple signatures, create multiple function objects with the same property in the JSON file.

Consider the following function that follows different code paths depending on the class of the second input. This function is presented as an example for code suggestions, and, therefore, does not perform any computations or error checking.

```
function anotherFunc(arg1,arg2,arg3)
    switch class(arg2)
        case 'double'
            % Follow code path 1
        case {'char','string'}
            % Follow code path 2
        otherwise
            error('Invalid syntax.')
    end
end
```

From a code suggestions perspective, consider the function as having two function signatures. The first signature accepts two required numeric values. The second signature accepts a required numeric, followed by a character or string, and finally a required numeric. To define multiple function signatures, define multiple function objects in the JSON file with the same property (function name).

```
{
    "_schemaVersion": "1.0.0",
```

```

"anotherFunc":
{
  "inputs":
  [
    {"name":"input1", "kind":"required", "type":["numeric"]},
    {"name":"input2", "kind":"required", "type":["numeric"]}
  ]
},
"anotherFunc":
{
  "inputs":
  [
    {"name":"input1", "kind":"required", "type":["numeric"]},
    {"name":"input2", "kind":"required", "type":[["char"],["string"]]},
    {"name":"input3", "kind":"required", "type":["numeric"]}
  ]
}
}

```

Alternatively, you can define multiple function signatures using the `mutuallyExclusiveGroup` property of the argument object. Typically, it is easier and more readable to implement multiple function objects, but using mutually exclusive groups enables reuse of common argument objects, such as `input1`.

```

{
  "_schemaVersion": "1.0.0",
  "anotherFunc":
  {
    "inputs":
    [
      {"name":"input1", "kind":"required", "type":["numeric"]},
      {"mutuallyExclusiveGroup":
        [
          [
            {"name":"input2", "kind":"required", "type":["numeric"]}
          ],
          [
            {"name":"input2", "kind":"required", "type":[["char"],["string"]]},
            {"name":"input3", "kind":"required", "type":["numeric"]}
          ]
        ]
      }
    ]
  }
}

```

See Also

`validateFunctionSignaturesJSON`

More About

- “Check Syntax as You Type”

External Websites

- <https://www.json.org/>

Display Custom Documentation

In this section...

“Overview” on page 30-27

“Create HTML Help Files” on page 30-28

“Create info.xml File” on page 30-29

“Create helptoc.xml File” on page 30-31

“Build a Search Database” on page 30-33

“Address Validation Errors for info.xml Files” on page 30-34

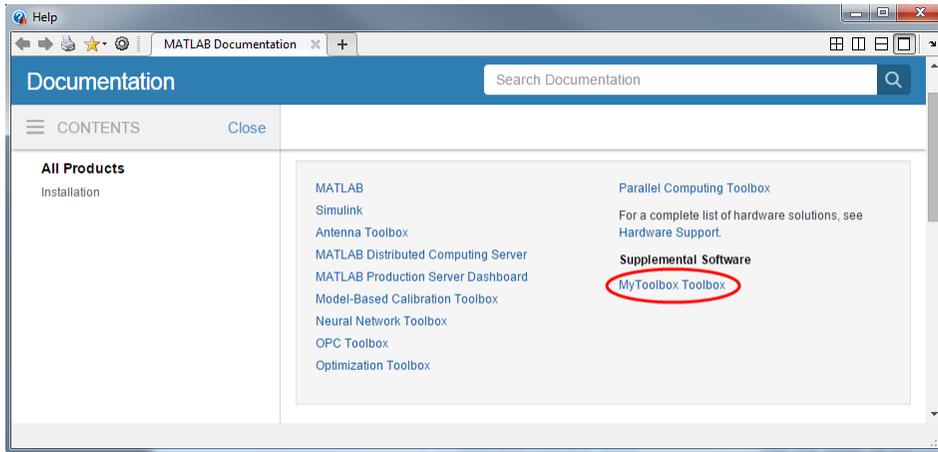
Overview

If you create a toolbox that works with MathWorks products, even if it only contains a few functions, you can include custom documentation in the form of HTML help files. Custom documentation for your toolbox can include figures, diagrams, screen captures, equations, and formatting to make your toolbox help more usable.

To display properly, your custom documentation must contain these files:

- **HTML help files** — These files contain your custom documentation information.
- **info.xml file** — This file enables MATLAB to find and identify your HTML help files.
- **helptoc.xml file** — This file contain the Table of Contents for your documentation that displays in the **Contents** pane of the Help browser. This file must be stored in the folder that contains your HTML help files.
- **Search database (optional)** — These files enable searching in your HTML help files.

To view your custom documentation, open the Help browser and navigate to the home page. At the bottom right of the home page, under **Supplemental Software**, click the name of your toolbox. Your help opens in the current window.



Create HTML Help Files

You can create HTML help files in any text editor or web publishing software. To create help files in MATLAB, use either of these two methods:

- Create a live script (*.mlx) and export it to HTML. For more information, see “Share Live Scripts and Functions” on page 19-67.
- Create a script (*.m), and publish it to HTML. For more information, see “Publishing MATLAB Code” on page 23-4.

Store all your HTML help files in one folder, such as an `html` subfolder in your toolbox folder. This folder must be:

- On the MATLAB search path
- Outside the `matlabroot` folder
- Outside any installed hardware support package help folder

Documentation sets often contain:

- A roadmap page (that is, an initial landing page for the documentation)
- Examples and topics that explain how to use the toolbox
- Function or block reference pages

Create info.xml File

The `info.xml` file describes your custom documentation, including the name to display for your documentation. It also identifies where to find your HTML help files and the `helptoc.xml` file. Create a file named `info.xml` for each toolbox you document.

To create `info.xml` to describe your toolbox, you can adapt this template:

```
<productinfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="optional">
  <?xml-stylesheet type="text/xsl"href="optional"?>

  <matlabrelease>R2016b</matlabrelease>
  <name>MyToolbox</name>
  <type>toolbox</type>
  <icon></icon>
  <help_location>html</help_location>

</productinfo>
```

You can also create `info.xml` by using the template `info_template.xml` included with the MATLAB documentation. To create and edit a copy of the template file in your current folder, run this code in the command window:

```
copyfile(fullfile(matlabroot,'help','techdoc','matlab_env',...
  'examples','templates','info_template.xml'),pwd)
fileattrib('info_template.xml','+w')
edit('info_template.xml')
```

The following table describes the required elements of the `info.xml` file.

XML Tag	Description	Value in Template	Notes
<code><matlabrelease></code>	Release of MATLAB	R2016b	Indicates when you added help files. Not displayed in the browser.
<code><name></code>	Title of toolbox	MyToolbox	The name to display for your custom documentation in the browser Contents pane.
<code><type></code>	Label for the toolbox	toolbox	Allowable values: <code>matlab</code> , <code>toolbox</code> , <code>simulink</code> , <code>blockset</code> , <code>links_targets</code> , <code>other</code> .

XML Tag	Description	Value in Template	Notes
<icon>	Icon for the Start button (not used)	none	No longer used, but the <icon> element is still required for MATLAB to parse the <code>info.xml</code> file.
<help_location>	Location of help files	html	Name of the subfolder containing <code>helptoc.xml</code> and the HTML help files for your toolbox. If the help location is not a subfolder of the <code>info.xml</code> file location, specify the path to <code>help_location</code> relative to the <code>info.xml</code> file. If you provide HTML help files for multiple toolboxes, the <code>help_location</code> in each <code>info.xml</code> file must be a different folder.
<help_contents_icon>	Icon to display in Contents pane	none	Ignored in MATLAB R2015a and later. Does not cause error if it appears in the <code>info.xml</code> file, but is not required.

You also can include comments in your `info.xml` file, such as copyright and contact information. Create comments by enclosing the text on a line between `<!--` and `-->`.

When you create the `info.xml` file, make sure that:

- You include all required elements.
- The entries are in the same order as in the preceding table.
- File and folder names in the XML exactly match the names of your files and folders and are capitalized identically.
- The `info.xml` file is in a folder on the MATLAB search path.

Note MATLAB parses the `info.xml` file and displays your documentation when you add the folder that contains `info.xml` to the path. If you created an `info.xml` file in a folder already on the path, remove the folder from the path. Then add the folder

again, so that MATLAB parses the file. Make sure that the folder you are adding is *not* your current folder.

Create helptoc.xml File

The `helptoc.xml` file defines the hierarchy of help files displayed in the **Contents** pane of the Supplemental Software browser.

You can create a `helptoc.xml` file by using the template included with the MATLAB documentation. To create and edit a copy of the template file `helptoc_template.xml` in your current folder, run this code in the Command Window:

```
copyfile(fullfile(matlabroot,'help','techdoc','matlab_env',...
'examples','templates','helptoc_template.xml'),pwd)
fileattrib('helptoc_template.xml','+w')
edit('helptoc_template.xml')
```

Place the `helptoc.xml` file in the folder that contains your HTML documentation files. This folder must be referenced as the `<help_location>` in your `info.xml` file.

Each `<tocitem>` entry in the `helptoc.xml` file references one of your HTML help files. The first `<tocitem>` entry in the `helptoc.xml` file serves as the initial landing page for your documentation.

Within the top-level `<toc>` element, the nested `<tocitem>` elements define the structure of your table of contents. Each `<tocitem>` element has a `target` attribute that provides the file name. File and path names are case-sensitive.

When you create the `helptoc.xml` file, make sure that:

- The location of the `helptoc.xml` files is listed as the `<help_location>` in your `info.xml` file.
- All file and path names exactly match the names of the files and folders, including capitalization.
- All path names use URL file path separators (`/`). Windows style file path separators (`\`) can cause the table of contents to display incorrectly. For example, if you have an HTML help page `firstfx.html` located in a subfolder called `refpages` within the main documentation folder, the `<tocitem>` `target` attribute value for that page would be `refpages/firstfx.html`.

Example helptoc.xml File

Suppose that you have created the following HTML files:

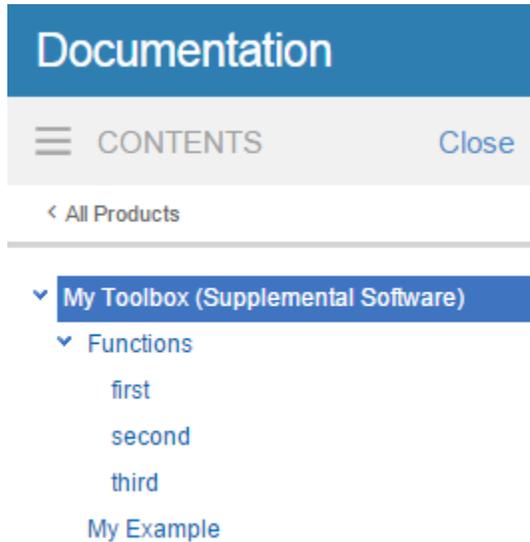
- A roadmap or starting page for your toolbox, `mytoolbox.html`.
- A page that lists your functions, `funclist.html`.
- Three function reference pages: `firstfx.html`, `secondfx.html`, and `thirdfx.html`.
- An example, `myexample.html`.

Include file names and descriptions in a `helptoc.xml` file as follows:

```
<?xml version='1.0' encoding="utf-8"?>
<toc version="2.0">

  <tocitem target="mytoolbox.html">My Toolbox
    <tocitem target="funclist.html">Functions
      <tocitem target="firstfx.html">first</tocitem>
      <tocitem target="secondfx.html">second</tocitem>
      <tocitem target="thirdfx.html">third</tocitem>
    </tocitem>
    <tocitem target="myexample.html">My Example
  </tocitem>
</toc>
```

This `helptoc.xml` file, paired with a properly formulated `info.xml` file, produced this display in the Help browser.



Build a Search Database

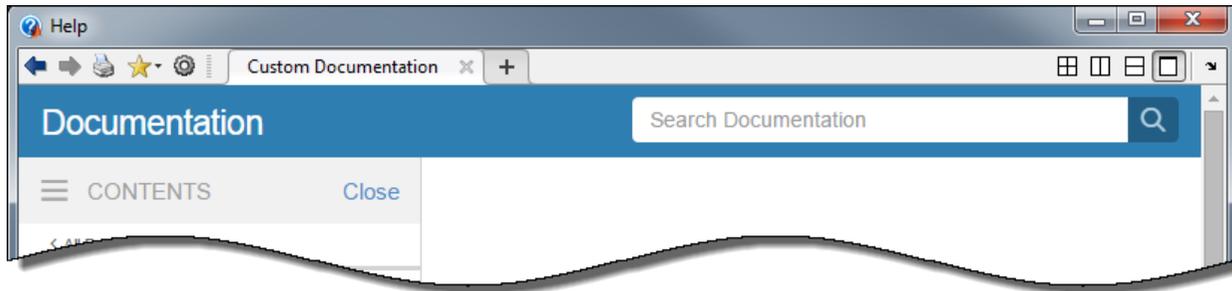
To make your documentation searchable, create a search database, also referred to as a search index, using the `builddocsearchdb` command. When using this command, specify the complete path to the folder that contains your HTML files.

For example, suppose that your HTML files are in `C:\MATLAB\MyToolbox\html`. This command creates a searchable database for those files:

```
builddocsearchdb('C:\MATLAB\MyToolbox\html')
```

`builddocsearchdb` creates a subfolder of `C:\MATLAB\MyToolbox\html` named `helpsearch-v3`, which contains the database files.

You can search for terms in your toolbox from the **Search Documentation** field in the Help browser.



Beginning with MATLAB R2014b, you can maintain search indexes side by side. For instance, if you already have a search index for MATLAB R2014a or earlier, run `builddocsearchdb` against your help files using MATLAB R2014b. Then, when you run any MATLAB release, the help browser automatically uses the appropriate index for searching your documentation database.

Address Validation Errors for `info.xml` Files

What Are XML Validation Errors?

When MATLAB finds an `info.xml` file on the search path or in the current folder, it 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 or add folders to the search path.

The primary causes of an XML file validation error are:

- Entities are missing or out of order in the `info.xml` file.
- An unrelated `info.xml` file exists.
- Syntax errors in the `info.xml` file.
- MATLAB is trying to access an outdated `info.xml` file for a MathWorks product.

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.

For a description of the elements you need in an `info.xml` file and their required ordering, see “Create info.xml File” on page 30-29.

Unrelated info.xml File

Suppose that you have a file named `info.xml` that has nothing to do with custom documentation. Because this `info.xml` file is an unrelated file, if it causes an error, you can safely ignore it. To prevent the error message from reoccurring, rename the unrelated `info.xml` file. Alternatively, ensure that the file is not on the search path or in the current folder.

Syntax Errors in the info.xml File.

Use the error message to isolate the problem or use any XML schema validator. For more information about the structure of the `info.xml` file, consult its schema 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. Alternatively, remove the outdated `info.xml` file from the search path and out of the current folder.

See Also

Related Examples

- “Display Custom Examples” on page 30-37
- “Create and Share Toolboxes” on page 25-13

- “Add Help for Your Program” on page 20-6

Display Custom Examples

In this section...

“How to Display Examples” on page 30-37

“Elements of the demos.xml File” on page 30-38

How to Display Examples

To display examples such as videos, published program scripts, or other files that illustrate the use of your programs in the MATLAB help browser, follow these steps:

- 1 Create your example files. Store the files in a folder that is on the MATLAB search path, but outside the *matlabroot* folder.

Tip MATLAB includes a feature that converts scripts or functions to formatted HTML files, which you can display as examples. To create these HTML files in MATLAB, use either of these two methods:

- Create a live script (*.mlx) and export it to HTML. For more information, see “Share Live Scripts and Functions” on page 19-67.
- Create a script (*.m), and publish it to HTML. For more information, see “Publishing MATLAB Code” on page 23-4.

- 2 Create a `demos.xml` file that describes the name, type, and display information for your examples.

For example, suppose that you have a toolbox named `My Sample`, which contains a script named `my_example` that you published to HTML. This `demos.xml` file allows you to display `my_example`:

```
<?xml version="1.0" encoding="utf-8"?>
<demos>
  <name>My Sample</name>
  <type>toolbox</type>
  <icon>HelpIcon.DEMOS</icon>
  <description>This text appears on the main page for your examples.</description>
  <website><a href="https://www.mathworks.com">Link to your Web site</a></website>

  <demosection>
    <label>First Section</label>
    <demoitem>
```

```

        <label>My Example Title</label>
        <type>M-file</type>
        <source>my_example</source>
    </demoitem>
</demosession>

</demos>

```

Note <demosession> elements are optional.

3 View your examples.

- a In the Help browser, navigate to the home page.
- b At the bottom of the page, under **Supplemental Software** click the link for your example.

Your example opens in the main help window.

Elements of the demos.xml File

- “General Information in <demos>” on page 30-38
- “Categories Using <demosession>” on page 30-39
- “Information About Each Example in <demoitem>” on page 30-39

General Information in <demos>

Within the `demos.xml` file, the root tag is `<demos>`. This tag includes elements that determine the contents of the main page for your examples.

XML Tag	Notes
<code><name></code>	Name of your toolbox or collection of examples.
<code><type></code>	Possible values are <code>matlab</code> , <code>simulink</code> , <code>toolbox</code> , or <code>blockset</code> .
<code><icon></code>	Ignored in MATLAB R2015a and later. In previous releases, this icon was the icon for your example. In those releases, you can use a standard icon, <code>HelpIcon.DEMOS</code> . Or, you can provide a custom icon by specifying a path to the icon relative to the location of the <code>demos.xml</code> file.

XML Tag	Notes
<description>	The description that appears on the main page for your examples.
<website>	(Optional) Link to a website. For example, MathWorks examples include a link to the product page at https://www.mathworks.com .

Categories Using <demosession>

Optionally, define categories for your examples by including a <demosession> for each category. If you include *any* categories, then *all* examples must be in categories.

Each <demosession> element contains a <label> that provides the category name, and the associated <demosessionitem> elements.

Information About Each Example in <demosessionitem>

XML Tag	Notes
<label>	Defines the title to display in the browser.
<type>	Possible values are M-file, model, M-GUI, video, or other. Typically, if you published your example using the publish function, the appropriate <type> is M-file.
<source>	If <type> is M-file, model, M-GUI, then <source> is the name of the associated .m file or model file, with no extension. Otherwise, do not include a <source> element, but include a <callback> element.
<file>	Use this element only for examples with a <type> value other than M-file when you want to display an HTML file that describes the example. Specify a relative path from the location of demos.xml.
<callback>	Use this element only for examples with a <type> value of video or other to specify an executable file or a MATLAB command to run the example.

XML Tag	Notes
<dependency>	(Optional) Specifies other products required to run the example, 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.

Source Control Interface

The source control interface provides access to your source control system from the MATLAB desktop.

- “About MathWorks Source Control Integration” on page 31-3
- “Select or Disable Source Control System” on page 31-6
- “Create New Repository” on page 31-7
- “Review Changes in Source Control” on page 31-9
- “Mark Files for Addition to Source Control” on page 31-10
- “Resolve Source Control Conflicts” on page 31-11
- “Commit Modified Files to Source Control” on page 31-15
- “Revert Changes in Source Control” on page 31-16
- “Set Up SVN Source Control” on page 31-17
- “Check Out from SVN Repository” on page 31-24
- “Update SVN File Status and Revision” on page 31-28
- “Get SVN File Locks” on page 31-29
- “Set Up Git Source Control” on page 31-30
- “Clone from Git Repository” on page 31-36
- “Update Git File Status and Revision” on page 31-38
- “Branch and Merge with Git” on page 31-40
- “Pull, Push and Fetch Files with Git” on page 31-45
- “Move, Rename, or Delete Files Under Source Control” on page 31-49
- “Customize External Source Control to Use MATLAB for Diff and Merge” on page 31-50
- “MSSCCI Source Control Interface” on page 31-53
- “Set Up MSSCCI Source Control” on page 31-54
- “Check Files In and Out from MSSCCI Source Control” on page 31-61
- “Additional MSSCCI Source Control Actions” on page 31-64

- “Access MSSCCI Source Control from Editors” on page 31-71
- “Troubleshoot MSSCCI Source Control Problems” on page 31-72

About MathWorks Source Control Integration

You can use MATLAB to work with files under source control. You can perform operations such as update, commit, merge changes, and view revision history directly from the Current Folder browser.

MATLAB integrates with:

- Subversion® (SVN)
- Git™

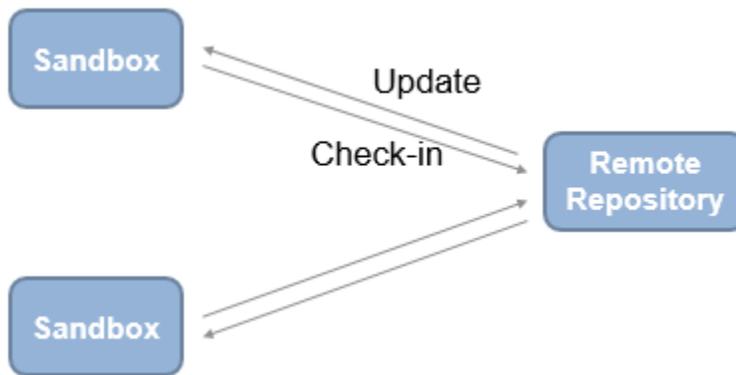
To use source control in your project, use any of these workflows:

- Retrieve files from an existing repository. See “Check Out from SVN Repository” on page 31-24 or “Clone from Git Repository” on page 31-36.
- Add source control to a folder. See “Create New Repository” on page 31-7.
- Add new files in a folder already under source control. See “Mark Files for Addition to Source Control” on page 31-10.

Additional source control integrations, such as Microsoft Source-Code Control Interface (MSSCCI), are available for download from the Add-On Explorer. For more information, see “Get Add-Ons”.

Classic and Distributed Source Control

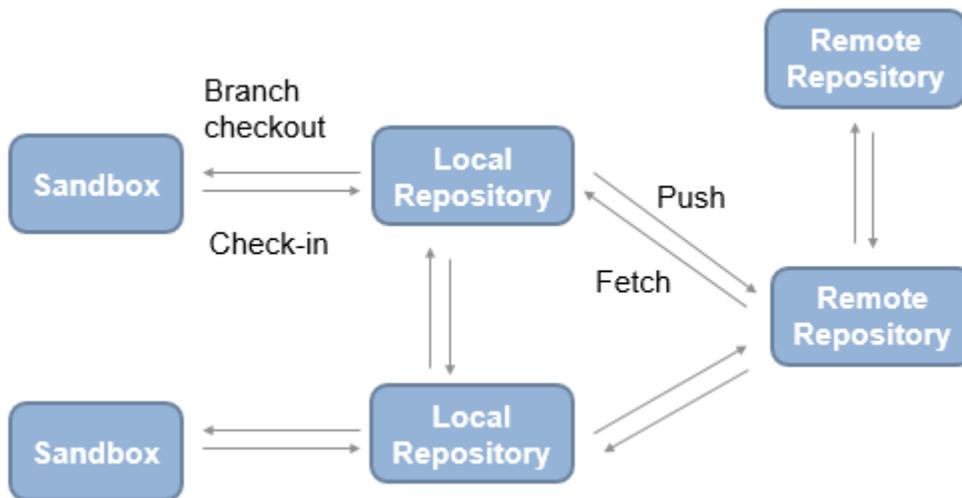
This diagram represents the classic source control workflow (for example, using SVN).



Benefits of classic source control:

- Locking and user permissions on a per-file basis (e.g., you can enforce locking of model files)
- Central server, reducing local storage needs
- Simple and easy to learn

This diagram represents the distributed source control workflow (for example, using Git).



Benefits of distributed source control:

- Offline working
- Local repository, which provides full history
- Branching
- Multiple remote repositories, enabling large-scale hierarchical access control

To choose classic or distributed source control, consider these tips.

Classic source control can be helpful if:

- You need file locks.
- You are new to source control.

Distributed source control can be helpful if:

- You need to work offline, commit regularly, and need access to the full repository history.
- You need to branch locally.

Select or Disable Source Control System

Select Source Control System

If you are just starting to use source control in MATLAB, select a source control system that is part of the MathWorks source control integration with the Current Folder browser, such as Subversion or Git. Doing so enables you to take advantage of the built-in nature of the integration. MathWorks source control integration is on by default.

- 1 On the **Home** tab, in the **Environment** section, click **Preferences**.
- 2 In the preferences dialog box, navigate to the **MATLAB > General > Source Control** pane.
- 3 To use the MathWorks source control integration, which is accessible through the Current Folder browser, select **Enable MathWorks source control integration**. Use this option for source control systems such as Subversion and Git. This is the default option, unless you previously set up source control with MATLAB.

Disable Source Control

When you disable source control, MATLAB does not destroy repository information. For example, it does not remove the `.svn` folder. You can put the folder back under source control by enabling the source control integration again.

- 1 On the **Home** tab, in the **Environment** section, click **Preferences**.
- 2 In the Preferences dialog box, in the **MATLAB > General > Source Control** pane, select **None**.

Create New Repository

You can use MATLAB to add source control to files in a folder. If you want to add version control to your files without sharing with another user, it is quickest to create a local Git repository in your sandbox.

To use a Git server for your remote repository, you can set up your own Apache™ Git server or use a Git server hosting solution. If you cannot set up a server and must use a remote repository via the file:/// protocol, make sure that it is a bare repository with no checked out working copy.

For SVN, check that your sandbox folder is on a local hard disk. Using a network folder with SVN is slow and unreliable.

Before using source control, you must register binary files with your source control tools to avoid corruption. See “Register Binary Files with SVN” on page 31-18 or “Register Binary Files with Git” on page 31-32.

Tip To check out an existing SVN repository, see “Check Out from SVN Repository” on page 31-24. To clone an existing remote Git repository, see “Clone from Git Repository” on page 31-36.

- 1 Right-click in the white space (any blank area) of the MATLAB Current Folder browser. Select **Source Control > Manage Files**.
- 2 In the Manage Files Using Source Control dialog box, in the **Source control integration** list:
 - For an SVN repository, select **Built-In SVN Integration**.
 - For a Git repository, select **Git**.
- 3 Click the **Change** button to open the Specify SVN Repository URL dialog box if you are using SVN or the Select a Repository dialog box if you are using Git.

If you are using Git, you can change the repository URL after the repository is created. In the Current Folder browser, in a folder under source control, right-click and select **Source Control > Remote** and specify a new URL.

- 4 Click the **Create a repository** button  to create a repository on disk.
- 5 Select an empty folder or create a new folder in which you want to create the repository and click **Select Folder** to create the repository.

For SVN, the URL of the new repository is in the **Repository URL** box, and the trunk folder is selected. Specify file:// URLs and create new repositories for single users only. For multiple users, see “Share a Subversion Repository” on page 31-22.

- 6 In the Specify SVN Repository URL (SVN) or Select a Repository (Git), click **Validate** to check the path to the selected repository, and then click **OK**.

If your SVN repository has a file URL, a warning appears that file URLs are for single users. Click **OK** to continue.

- 7 In the Manage Files Using Source Control dialog box, choose the location for your sandbox, and then click **Retrieve**.

For an SVN sandbox, the selected folder can contain files. However, for a Git sandbox, the selected folder must be empty. You cannot clone a remote repository into a folder that contains files.

You need some additional setup steps if you want to merge branches with Git. See “Install Command-Line Git Client” on page 31-31.

After integrity checks are complete, you can commit the first version of your files to the new repository.

See Also

Related Examples

- “Set Up SVN Source Control” on page 31-17
- “Set Up Git Source Control” on page 31-30
- “Check Out from SVN Repository” on page 31-24
- “Clone from Git Repository” on page 31-36
- “Commit Modified Files to Source Control” on page 31-15

Review Changes in Source Control

The files under source control that you have changed display the Modified File symbol  in the Current Folder browser. Right-click the file in the Current Folder browser, select **Source Control**, and select:

- **Show Revisions** to open the File Revisions dialog box and browse the history of a file. You can view information about who previously committed the file, when they committed it, the log messages, and the list of files in each change set. You can select multiple files and view revision history for each file.

With SVN, you can select a revision and browse the lower list of files in the change set. Right-click files to view changes or save revisions.

- **Compare to Revision** to open a dialog box where you can select the revisions you want to compare and view a comparison report. You can either:
 - Select a revision and click **Compare to Local**.
 - Select two revisions and click **Compare Selected**.

With SVN, you can select a revision and browse the lower list of files in the change set. Right-click files to view changes or save revisions.

- **Compare to Ancestor** to run a comparison with the last checked-out version in the sandbox (SVN) or against the local repository (Git). The Comparison Tool displays a report.

If you need to update the status of the modified files, see “Update SVN File Status and Revision” on page 31-28 or “Update Git File Status and Revision” on page 31-38.

See Also

Related Examples

- “Resolve Source Control Conflicts” on page 31-11
- “Commit Modified Files to Source Control” on page 31-15
- “Revert Changes in Source Control” on page 31-16

Mark Files for Addition to Source Control

When you create a new file in a folder under source control, the Not Under Source Control symbol  appears in the status column of the Current Folder browser. To add a file to source control, right-click the file in the Current Folder browser, and select **Source Control** and then the Add option appropriate to your source control system. For example, select **Add to Git** or **Add to SVN**.

When the file is marked for addition to source control, the symbol changes to Added  .

Resolve Source Control Conflicts

Examining and Resolving Conflicts

If you and another user change the same file in different sandboxes or on different branches, a conflict message appears when you try to commit your modified files. Follow the procedure “Resolve Conflicts” on page 31-11 to extract conflict markers if necessary, compare the differences causing the conflict, and resolve the conflict.

To resolve conflicts you can:

- Use the Comparison Tool to merge changes between revisions.
- Decide to overwrite one set of changes with the other.
- Make changes manually by editing files.

For details on using the Comparison Tool to merge changes, see “Merge Text Files” on page 31-12.

After you are satisfied with the file that is marked conflicted, you can mark the conflict resolved and commit the file.

Resolve Conflicts

- 1 Look for conflicted files in the Current Folder browser.
- 2 Check the source control status column (**SVN** or **Git**) for files with a red warning symbol , which indicates a conflict.
- 3 Right-click the conflicted file and select **Source Control > View Conflicts** to compare versions.
- 4 Examine the conflict. A comparison report opens that shows the differences between the conflicted files.

With SVN, the comparison shows the differences between the file and the version of the file in conflict.

With Git, the comparison shows the differences between the file on your branch and the branch you want to merge into.

- 5 Use the Comparison Tool report to determine how to resolve the conflict.

You can use the Comparison Tool to merge changes between revisions, as described in “Merge Text Files” on page 31-12.

- 6 When you have resolved the changes and want to commit the version in your sandbox, in the Current Folder browser, right-click the file and select **Source Control > Mark Conflict Resolved**.

With Git, the Branch status in the Source Control Details dialog box changes from MERGING to SAFE.

- 7 Commit the modified files.

Merge Text Files

When comparing text files, you can merge changes from one file to the other. Merging changes is useful when resolving conflicts between different versions of files.

If you see conflict markers in a text comparison report like this:

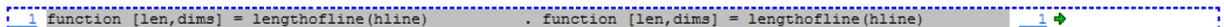
```
<<<<<<< .mine
```

then extract the conflict markers before merging, as described in “Extract Conflict Markers” on page 31-13.

Tip When comparing a file to another version in source control, by default the right file is the version in your sandbox and the left file is either a temporary copy of the previous version or another version causing a conflict (e.g., *filename_theirs*). You can swap the position of the files, so be sure to observe the file paths of the left and right file at the top of the comparison report. Merge differences from the temporary copy to the version in your sandbox to resolve conflicts.

- 1 In the Comparison Tool report, select a difference in the report and click **Merge**. The selected difference is copied from the left file to the right file.

Merged differences display gray row highlighting and a green merge arrow.



The screenshot shows a code editor with a line of code: `1 function [len,dims] = lengthofline(hline)`. The line is highlighted in gray, and a green arrow points to the right, indicating a merge operation. The line is enclosed in a dashed blue border.

The merged file name at the top of the report displays with an asterisk (*filename.m**) to show you that the file contains unsaved changes.

- 2 Click **Save Merged File** to save the file in your sandbox. To resolve conflicts, save the merged file over the conflicted file.
- 3 If you want to inspect the files in the editor, click the line number links in the report.

Note If you make any further changes in the editor, the comparison report does not update to reflect changes and report links can become incorrect.

- 4 When you have resolved the changes mark them as conflict resolved. Right-click the file in the Current Folder browser and select **Source Control > Mark Conflict Resolved**.

Extract Conflict Markers

- “What Are Conflict Markers?” on page 31-13
- “Extract Conflict Markers” on page 31-14

What Are Conflict Markers?

Source control tools can insert conflict markers in files that you have not registered as binary (e.g., text files). You can use MATLAB to extract the conflict markers and compare the files causing the conflict. This process helps you to decide how to resolve the conflict.

Caution Register files with source control tools to prevent them from inserting conflict markers and corrupting files. See “Register Binary Files with SVN” on page 31-18 or “Register Binary Files with Git” on page 31-32. If your files already contains conflict markers, the MATLAB tools can help you to resolve the conflict.

Conflict markers have the following form:

```
<<<<<<["mine" file descriptor]
["mine" file content]
=====
["theirs" file content]
<<<<<<["theirs" file descriptor]
```

If you try to open a file containing conflict markers, the Conflict Markers Found dialog box opens. Follow the prompts to fix the file by extracting the conflict markers. After you extract the conflict markers, resolve the conflicts as described in “Examining and Resolving Conflicts” on page 31-11.

To view the conflict markers, in the Conflict Markers Found dialog box, click **Load File**. Do not try to load files, because MATLAB does not recognize conflict markers. Instead, click **Fix File** to extract the conflict markers.

MATLAB checks only conflicted files for conflict markers.

Extract Conflict Markers

When you open a conflicted file or select **View Conflicts**, MATLAB checks files for conflict markers and offers to extract the conflict markers. MATLAB checks only conflicted files for conflict markers.

However, some files that are not marked as conflicted can still contain conflict markers. This can happen if you or another user marked a conflict resolved without removing the conflict markers and then committed the file. If you see conflict markers in a file that is not marked conflicted, you can extract the conflict markers.

- 1** In the Current Folder browser, right-click the file, and select **Source Control > Extract Conflict Markers to File**.
- 2** In the Extract Conflict Markers to File dialog box, leave the default option to copy “mine” file version over the conflicted file. Leave the **Compare extracted files** check box selected. Click **Extract**.
- 3** Use the Comparison Tool report as usual to continue to resolve the conflict.

Commit Modified Files to Source Control

Before you commit modified files, review changes and mark any new files for addition into source control. The files under source control that you can commit to a repository display the Added to Source Control symbol **+** or the Modified File symbol **■** in the Current Folder browser.

- 1** Right-click in the Current Folder browser and select **Source Control > View and Commit Changes**. In the View and Commit Changes dialog box, select the files to commit to the repository.
- 2** Enter comments in the dialog box, and click **Commit**.
- 3** A message appears if you cannot commit because the repository has moved ahead. Before you can commit the file, you must update the revision up to the current HEAD revision.
 - If you are using SVN source control, right-click in the Current Folder browser. Select **Source Control > Update All from SVN**.
 - If you are using Git source control, right-click in the Current Folder browser. Select **Source Control > Pull**.

Resolve any conflicts before you commit.

See Also

Related Examples

- “Mark Files for Addition to Source Control” on page 31-10
- “Review Changes in Source Control” on page 31-9
- “Resolve Source Control Conflicts” on page 31-11
- “Update SVN File Status and Revision” on page 31-28
- “Update Git File Status and Revision” on page 31-38
- “Pull, Push and Fetch Files with Git” on page 31-45

Revert Changes in Source Control

Revert Local Changes

With SVN, if you want to roll back local changes in a file, right-click the file and select **Source Control > Revert Local Changes and Release Locks**. This command releases locks and reverts to the version in the last sandbox update (that is, the last version you synchronized or retrieved from the repository). If your file is not locked, the menu option is **Source Control > Revert Local Changes**. To abandon all local changes, select all the files in the Current Folder browser before you select the command.

With Git, right-click a file and select **Source Control > Revert Local Changes**. Git does not have locks. To remove all local changes, right-click a blank space in the Current Folder browser and select **Source Control > Branches**. In the Branches dialog box, click **Revert to Head**.

Revert a File to a Specified Revision

- 1 Right-click a file in the Current Folder browser and select **Source Control > Revert using SVN** or **Revert using Git**.
- 2 In the Revert Files dialog box, choose a revision to revert to. Select a revision to view information about the change such as the author, date, and log message.
- 3 Click **Revert**.

If you revert a file to an earlier revision and then make changes, you cannot commit the file until you resolve the conflict with the repository history.

See Also

Related Examples

- “Resolve Source Control Conflicts” on page 31-11

Set Up SVN Source Control

MATLAB provides built-in SVN integration for use with Subversion (SVN) sandboxes and repositories. Because the implementation is built in to MATLAB, you do not need to install SVN. The built-in SVN integration supports secure logins. This integration ignores any existing SVN installation.

In this section...

“SVN Source Control Options” on page 31-17

“Register Binary Files with SVN” on page 31-18

“Standard Repository Structure” on page 31-21

“Tag Versions of Files” on page 31-21

“Enforce Locking Files Before Editing” on page 31-21

“Share a Subversion Repository” on page 31-22

SVN Source Control Options

To use the version of SVN provided with MATLAB, when you retrieve a file from source control, select **Built-In SVN Integration** in the **Source control integration** list. For instructions, see “Check Out from SVN Repository” on page 31-24. When you create a new sandbox using the MATLAB built-in SVN integration, the new sandbox uses the latest version of SVN provided by MATLAB.

Caution Before using source control, you must register binary files with the source control tools to avoid corruption. See “Register Binary Files with SVN” on page 31-18.

If you need to use a version of SVN other than the built-in version, you can create a repository using the **Command-Line SVN Integration (compatibility mode)** **Source control integration** option, but you must also install a command-line SVN client.

Command-line SVN integration communicates with any Subversion (SVN) client that supports the command-line interface. With **Command-Line SVN Integration (compatibility mode)**, if you try to rename a file or folder to a name that contains an @ character, an error occurs because command-line SVN treats all characters after the @ symbol as a peg revision value.

Register Binary Files with SVN

If you use third-party source control tools, you must register your MATLAB and Simulink file extensions such as `.mlx`, `.mat`, `.fig`, `.mdl`, and `.slx` as binary formats. If you do not register the extensions, these tools can corrupt your files when you submit them by changing end-of-line characters, expanding tokens, substituting keywords, or attempting to automerge. Corruption can occur whether you use the source control tools outside of MATLAB or if you try submitting files from MATLAB without first registering your file formats.

Also check that other file extensions are registered as binary to avoid corruption at check-in. Check and register files such as `.mdl`, `.slxp`, `.sldd`, `.p`, MEX-files, `.xlsx`, `.jpg`, `.pdf`, `.docx`, etc.

You must register binary files if you use any version of SVN, including the built-in SVN integration provided by MATLAB. If you do not register your extensions as binary, SVN might add annotations to conflicted MATLAB files and attempt automerge. To avoid this problem when using SVN, register file extensions.

- 1 Locate your SVN config file. Look for the file in these locations:
 - `C:\Users\myusername\AppData\Roaming\Subversion\config` or `C:\Documents and Settings\myusername\Application Data\Subversion\config` on Windows
 - `~/.subversion` on Linux or macOS
- 2 If you do not find a config file, create a new one. See “Create SVN Config File” on page 31-18.
- 3 If you find an existing config file, you have previously installed SVN. Edit the config file. See “Update Existing SVN Config File” on page 31-19.

Create SVN Config File

- 1 If you do not find an SVN config file, create a text file containing these lines:

```
[miscellany]
enable-auto-props = yes
[auto-props]
*.mlx = svn:mime-type=application/octet-stream
*.mat = svn:mime-type=application/octet-stream
*.fig = svn:mime-type=application/octet-stream
*.mdl = svn:mime-type=application/octet-stream
*.slx = svn:mime-type= application/octet-stream
```

- 2 Check for other file types you use that you also need to register as binary to avoid corruption at check-in. Check for files such as `.mat`, `.mdl`, `.slxp`, `.p`, MEX-files (`.mexa64`, `.mexmaci64`, `.mexw64`), `.xlsx`, `.jpg`, `.pdf`, `.docx`, etc. Add a line to the config file for each file type you need. Examples:

```
*.mdl = svn:mime-type=application/octet-stream
*.slxp = svn:mime-type=application/octet-stream
*.sldd = svn:mime-type=application/octet-stream
*.p = svn:mime-type=application/octet-stream
*.mexa64 = svn:mime-type=application/octet-stream
*.mexw64 = svn:mime-type=application/octet-stream
*.mexmaci64 = svn:mime-type=application/octet-stream
*.xlsx = svn:mime-type=application/octet-stream
*.docx = svn:mime-type=application/octet-stream
*.pdf = svn:mime-type=application/octet-stream
*.jpg = svn:mime-type=application/octet-stream
*.png = svn:mime-type=application/octet-stream
```

- 3 Name the file config and save it in the appropriate location:

- `C:\Users\myusername\AppData\Roaming\Subversion\config` or `C:\Documents and Settings\myusername\Application Data\Subversion\config` on Windows
- `~/.subversion` on Linux or macOS.

After you create the SVN config file, SVN treats new files with these extensions as binary. If you already have binary files in repositories, see “Register Files Already in Repositories” on page 31-20.

Update Existing SVN Config File

If you find an existing config file, you have previously installed SVN. Edit the config file to register files as binary.

- 1 Edit the config file in a text editor.
- 2 Locate the [miscellany] section, and verify the following line enables auto-props with yes:

```
enable-auto-props = yes
```

Ensure that this line is not commented (that is, that it does not start with #). Config files can contain example lines that are commented out. If there is a # character at the beginning of the line, delete it.

- 3 Locate the [auto-props] section. Ensure that [auto-props] is not commented. If there is a # character at the beginning, delete it.
- 4 Add the following lines at the end of the [auto-props] section:

```
*.mlx = svn:mime-type=application/octet-stream  
*.mat = svn:mime-type=application/octet-stream  
*.fig = svn:mime-type=application/octet-stream  
*.mdl = svn:mime-type=application/octet-stream  
*.slx = svn:mime-type= application/octet-stream
```

These lines prevent SVN from adding annotations to MATLAB and Simulink files on conflict and from automerging.

- 5 Check for other file types you use that you also need to register as binary to avoid corruption at check-in. Check for files such as .mdl, .slx, .p, MEX-files (.mexa64, .mexmaci64, .mexw64), .xlsx, .jpg, .pdf, .docx, etc. Add a line to the config file for each file type you use. Examples:

```
*.mdl = svn:mime-type=application/octet-stream  
*.slx = svn:mime-type=application/octet-stream  
*.sldd = svn:mime-type=application/octet-stream  
*.p = svn:mime-type=application/octet-stream  
*.mexa64 = svn:mime-type=application/octet-stream  
*.mexw64 = svn:mime-type=application/octet-stream  
*.mexmaci64 = svn:mime-type=application/octet-stream  
*.xlsx = svn:mime-type=application/octet-stream  
*.docx = svn:mime-type=application/octet-stream  
*.pdf = svn:mime-type=application/octet-stream  
*.jpg = svn:mime-type=application/octet-stream  
*.png = svn:mime-type=application/octet-stream
```

- 6 Save the config file.

After you create or update the SVN config file, SVN treats new files as binary. If you already have files in repositories, register them as described in “Register Files Already in Repositories” on page 31-20.

Register Files Already in Repositories

Caution Changing your SVN config file does not affect files already committed to an SVN repository. If a file is not registered as binary, use `svn propset` to manually register the files as binary.

To manually register a file in a repository as binary, use the following command with command-line SVN:

```
svn propset svn:mime-type application/octet-stream binaryfilename
```

Standard Repository Structure

Create your repository with the standard `tags`, `trunk`, and `branches` folders, and check out files from `trunk`. The Subversion project recommends this structure. See the Web page:

<https://svn.apache.org/repos/asf/subversion/trunk/doc/user/svn-best-practices.html>

If you use MATLAB to create an SVN repository, it creates the standard repository structure. To enable tagging, the repository must have the standard `trunk/` and `tags/` folders. After you create a repository with this structure, you can click **Tag** in the **Source Control** context menu to add tags to all of your files. For more information, see “Tag Versions of Files” on page 31-21.

Tag Versions of Files

With SVN, you can use tags to identify specific revisions of all files. To use tags with SVN, you need the standard folder structure in your repository and you need to check out your files from `trunk`. See “Standard Repository Structure” on page 31-21.

- 1 Right-click in the Current Folder browser, and select **Source Control > Tag**.
- 2 Specify the tag text and click **Submit**. The tag is added to every file in the folder. Errors appear if you do not have a `tags` folder in your repository.

Note You can retrieve a tagged version of your files from source control, but you cannot tag them again with a new tag. You must check out from `trunk` to create new tags.

Enforce Locking Files Before Editing

To require that users remember to get a lock on files before editing, configure SVN to make files with specified extensions read only. When your files are read only, you need to select Right-click in the Current Folder browser, and select **Source Control > Get File Lock** before you can edit them. This setting prevents editing of files without getting the

file lock. When the file has a lock, other users know the file is being edited, and you can avoid merge issues.

To enforce locking files, modify entries in the SVN `config` file. To locate your SVN `config` file, see “Register Binary Files with SVN” on page 31-18.

- 1 To make files with a `.m` extension read only, add a property to your SVN `config` file in the `[auto-props]` section. If there is no entry for files with a `.m` extension, add one with the `needs-lock` property.

```
*.m = svn:needs-lock=yes
```

If an entry exists, you can combine properties in any order, but multiple entries must be on a single line separated by semicolons.

- 2 To make files with a `.mlx` extension read only, add a property to your SVN `config` file in the `[auto-props]` section. Since you must register files with a `.mlx` extension as binary, there is an entry for the file type. Add the `needs-lock` property to the entry in any order, but on the same line and separated by a semicolon.

```
*.mlx = svn:mime-type=application/octet-stream;svn:needs-lock=yes
```

- 3 Re-create the sandbox for the configuration to take effect.

With this setting, you need to select **Get File Lock** before you can edit files with a `.m` extension. See “Get SVN File Locks” on page 31-29.

Share a Subversion Repository

You can specify a repository location using the `file://` protocol. However, Subversion documentation strongly recommends that only a single user access a repository directly via `file://` URLs. See the Web page:

<http://svnbook.red-bean.com/en/1.7/svn-book.html#svn.serverconfig.choosing.recommendations>

Caution Do not allow multiple users to access a repository directly via `file://` URLs or you risk corrupting the repository. Use `file://` URLs only for single-user repositories.

Be aware of this caution if you use MATLAB to create a repository. MATLAB uses the `file://` protocol. Creating new repositories is provided for local, single-user access only, for testing and debugging. Accessing a repository via `file://` URLs is slower than using a server.

When you want to share a repository, you need to set up a server. You can use `svnserve` or the Apache SVN module. See the Web page references:

<http://svnbook.red-bean.com/en/1.7/svn-book.html#svn.serverconfig.svnserve>
<http://svnbook.red-bean.com/en/1.7/svn-book.html#svn.serverconfig.httpd>

See Also

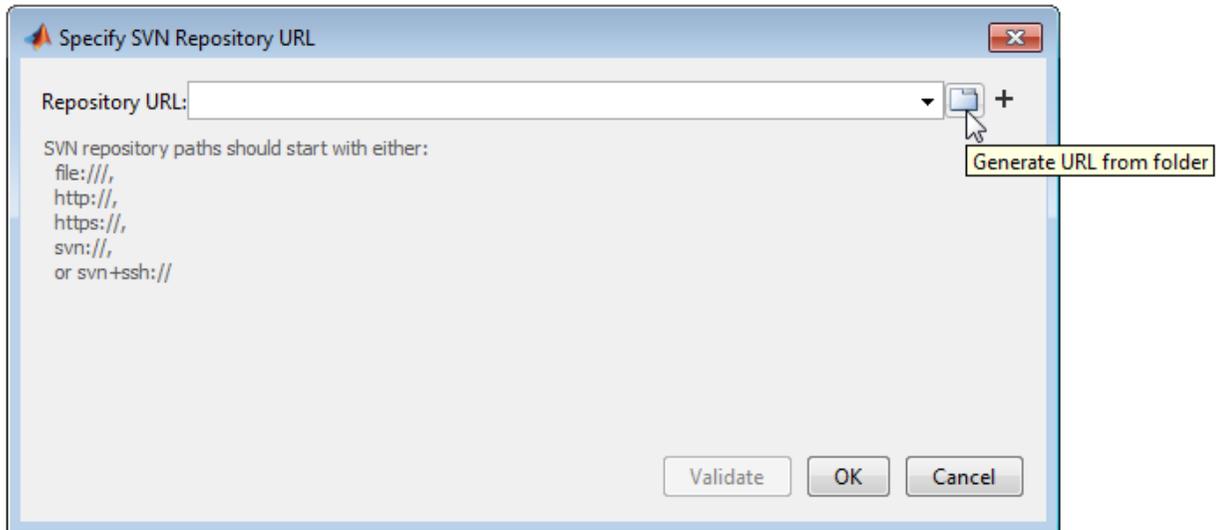
Related Examples

- “Check Out from SVN Repository” on page 31-24

Check Out from SVN Repository

Create a new local copy of a repository by retrieving files from source control.

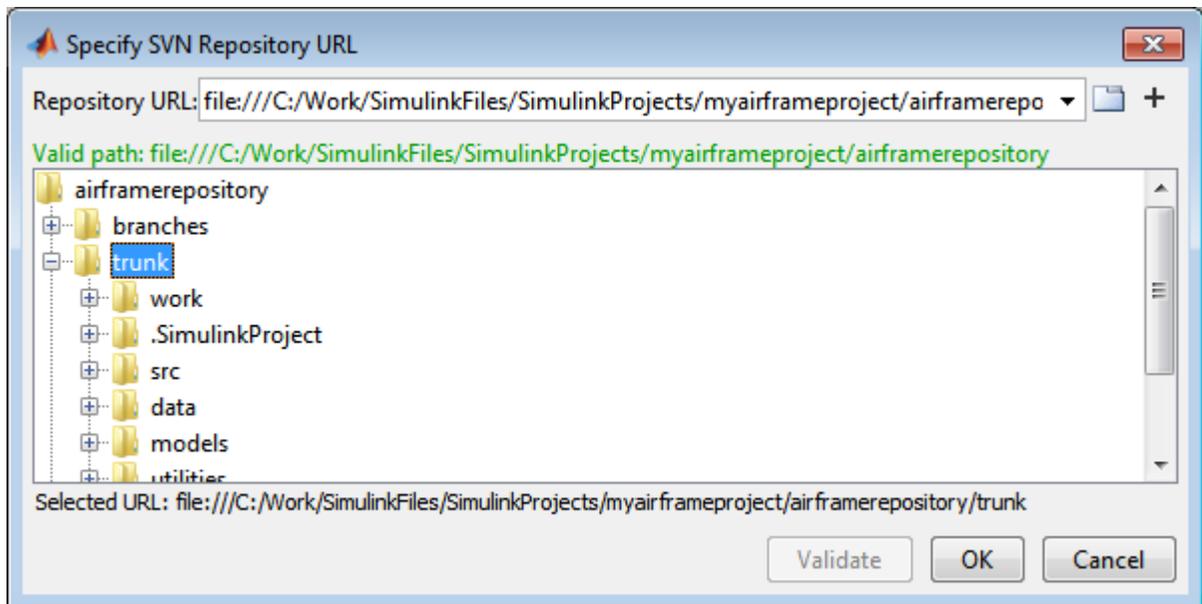
- 1 Right-click in the white space (any blank area) in the Current Folder browser and select **Source Control > Manage Files**.
- 2 In the Manage Files Using Source Control dialog box, select the source control interface from the **Source control integration** list. To use SVN, leave the default Built-In SVN Integration.
- 3 Click **Change** to browse for and validate the repository path. (If you know your repository location, you can paste it into the **Repository Path** box and proceed to step 8.)
- 4 In the Specify Repository URL dialog box, specify the repository URL by entering a URL in the box, using the list of recent repositories, or by using the **Generate URL from folder** button .



Caution Use `file://` URLs only for single-user repositories. For more information, see “Share a Subversion Repository” on page 31-22.

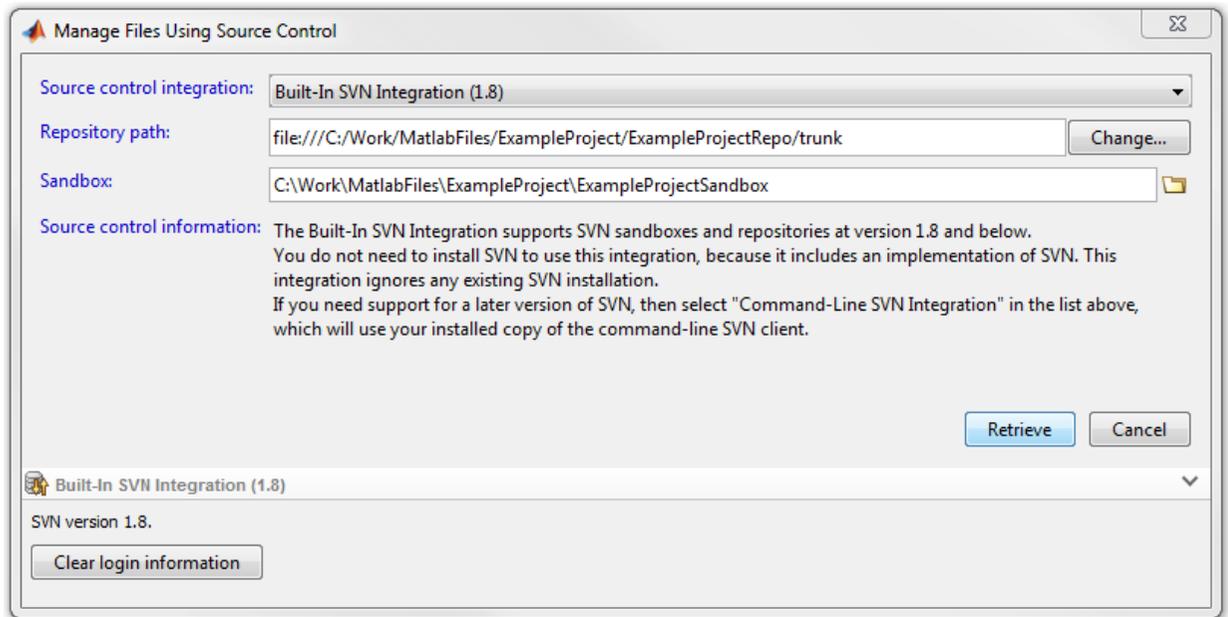
- 5 Click **Validate** to check the repository path.

- If the path is invalid, check the URL against your source control repository browser.
- 6 If you see an authentication dialog box for your repository, enter login information to continue.
 - 7 If necessary, select a deeper folder in the repository tree. You might want to check out from `trunk` or from a branch folder under `tags`, if your repository contains tagged versions of files. You can check out from a branch, but the built-in SVN integration does not support branch merging. Use an external tool such as TortoiseSVN to perform branch merging. The example shows `trunk` selected, and the Selected URL displays at the bottom of the dialog box. The retriever uses this URL when you click **OK**.



- 8 When you have finished specifying the URL path you want to retrieve, click **OK**.
- 9 In the Manage Files Using Source Control dialog box, select the sandbox folder where you want to put the retrieved files, and click **Retrieve**.

Caution Use local sandbox folders. Using a network folder with SVN slows source control operations.



The Manage Files Using Source Control dialog box displays messages as it retrieves the files from source control.

Note To update an existing sandbox from source control, see “Update SVN File Status and Revision” on page 31-28.

Retrieve Tagged Version of Repository

To use tags with SVN, you need the standard folder structure in your repository. For more information, see “Standard Repository Structure” on page 31-21.

- 1 Right-click in the white space in the Current Folder browser, and select **Source Control > Manage Files**.
- 2 In the Manage Files Using Source Control dialog box, select the source control interface from the **Source control integration** list. To use SVN, leave the default **Built-In SVN Integration**.
- 3 Click **Change** to select the Repository Path that you want to retrieve files from.

- 4 In the Specify Repository URL dialog box:
 - a Select a recent repository from the **Repository URL** list, or click the **Generate URL from folder** button  to browse for the repository location.
 - b Click **Validate** to show the repository browser.
 - c Expand the **tags** folder in the repository tree, and select the tag version you want. Navigate up a level in the repository if the URL contains the **trunk**.
 - d Click **OK** to continue and return to the Manage Files Using Source Control dialog box.
- 5 Select the sandbox folder to receive the tagged files. You must use an empty sandbox folder or specify a new folder.
- 6 Click **Retrieve**.

See Also

Related Examples

- “Set Up SVN Source Control” on page 31-17
- “Update SVN File Status and Revision” on page 31-28

Update SVN File Status and Revision

In this section...
“Refresh Status of Files” on page 31-28
“Update Revisions of Files” on page 31-28

Refresh Status of Files

To refresh the source control status of files, select one or more files in the Current Folder browser, right-click and select **Source Control > Refresh SVN status**.

To refresh the status of all files in a folder, right-click the white space of the Current Folder browser and select **Source Control > Refresh SVN status**.

Note For SVN, refreshing the source control status does not contact the repository. To get the latest revisions, see “Update Revisions of Files” on page 31-28.

Update Revisions of Files

To update the local copies of selected files, select one or more files in the Current Folder browser, right-click and select **Source Control > Update Selection from SVN**.

To update all files in a folder, right-click the Current Folder browser and select **Source Control > Update All from SVN**.

See Also

Related Examples

- “Check Out from SVN Repository” on page 31-24
- “Review Changes in Source Control” on page 31-9

Get SVN File Locks

It is good practice to get a file lock before editing a file. The lock tells other users that the file is being edited, and you can avoid merge issues. When you set up source control, you can configure SVN to make files with certain extensions read only. Users must get a lock on these read-only files before editing.

In the Current Folder browser, select the files you want to check out. Right-click the selected files and select **Source Control > Get File Lock**. A lock symbol  appears in the source control status column. Other users cannot see the lock symbol in their sandboxes, but they cannot get a file lock or check in a change when you have the lock. To view or break locks, right-click in the Current Folder browser and select **Source Control > Locks**.

If you see an SVN message reporting a working copy locked error, remove stale locks. In the Current Folder browser, right-click and select **Source Control > SVN Cleanup**. SVN uses working copy locks internally and they are not the file locks you control using **Source Control > Get File Lock**.

Manage SVN Repository Locks

To manage global SVN locks for a repository, from the top-level repository folder, right-click the white space (any blank area) in the Current Folder browser and select **Source Control > Locks**.

In the **SVN Repository Locks** dialog box, you can:

- View which users have locks on files.
- Right-click to break locks.
- Group locks by user or file.

See Also

Related Examples

- “Enforce Locking Files Before Editing” on page 31-21

Set Up Git Source Control

In this section...
“About Git Source Control” on page 31-30
“Install Command-Line Git Client” on page 31-31
“Register Binary Files with Git” on page 31-32
“Add Git Submodules” on page 31-34

About Git Source Control

Git integration with MATLAB provides distributed source control with support for creating and merging branches. Git is a distributed source control tool, so you can commit changes to a local repository and later synchronize with other remote repositories.

Git supports distributed development because every sandbox contains a complete repository. The full revision history of every file is saved locally. This enables working offline, because you do not need to contact remote repositories for every local edit and commit, only when pushing batches of changes. In addition, you can create your own branches and commit local edits. Doing so is fast, and you do not need to merge with other changes on each commit.

Capabilities of Git source control:

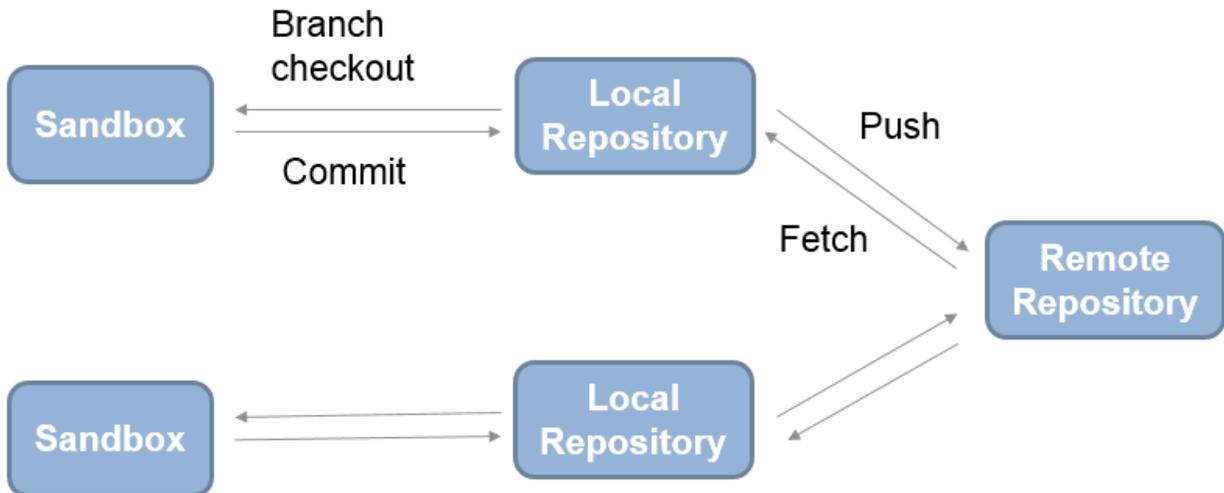
- Branch management
- Local full revision history
- Local access that is quicker than remote access
- Offline working
- Tracking of file names and contents separately
- Enforcing of change logs for tracing accountability
- Integration of batches of changes when ready

These capabilities do not suit every situation. If your project is not appropriate for offline working or your repository is too large for a full local revision history, for example, Git is not the ideal source control. In addition, if you need to enforce locking of files before editing, Git does not have this ability. In that situation, SVN is the better choice.

When you use Git in MATLAB, you can:

- Create local Git repositories.
- Pull and fetch files from remote Git repositories.
- Create and switch branches.
- Merge branches locally.
- Commit locally.
- Push files to remote Git repositories.

This diagram represents the distributed Git workflow.



Install Command-Line Git Client

If you want to use Git to merge branches in MATLAB, you must also install a command-line Git client that is available systemwide. You can use other Git functionality without any additional installation.

Some clients are not available systemwide, including the mingw32 environment provided by GitHub® (**Git Shell** on the **Start** menu). Installing command-line Git makes it available systemwide, and then MATLAB can locate standard ssh keys.

Check if Git is available by using the command `!git` in MATLAB. If Git is not available, install it. After you have installed a command-line Git client and registered your files as binary, you can use the merging features of Git in MATLAB.

On Windows:

- 1 Download the Git installer and run it. You can find command-line Git at:
<https://msysgit.github.io/>
- 2 In the section on adjusting your PATH, choose the install option to **Use Git from the Windows Command Prompt**. This option adds Git to your PATH variable, so that MATLAB can communicate with Git.
- 3 In the section on configuring the line-ending conversions, choose the option **Checkout as-is, commit as-is** to avoid converting any line endings in files.
- 4 To avoid corrupting binary files, before using Git to merge branches, register the binary files.

On Linux, Git is available for most distributions. Install Git for your distribution. For example, on Debian®, install Git by entering:

```
sudo apt-get install git
```

On Mac, on Mavericks (10.9) or above, try to run `git` from the Terminal. If you do not have Git installed already, it will prompt you to install Xcode Command Line Tools. For more options, see <https://git-scm.com/doc>.

Register Binary Files with Git

If you use third-party source control tools, you must register your MATLAB and Simulink file extensions such as `.mlx`, `.mat`, `.fig`, `.mdl`, and `.slx` as binary formats. If you do not register the extensions, these tools can corrupt your files when you submit them by changing end-of-line characters, expanding tokens, substituting keywords, or attempting to automerger. Corruption can occur whether you use the source control tools outside of MATLAB or if you try submitting files from MATLAB without first registering your file formats.

Also check that other file extensions are registered as binary to avoid corruption at check-in. Check and register files such as `.mdl`, `.slx`, `.sldd`, `.p`, MEX-files, `.xlsx`, `.jpg`, `.pdf`, `.docx`, etc.

After you install a command-line Git client, you can prevent Git from corrupting your files by inserting conflict markers. To do so, edit your `.gitattributes` file to register binary files. For details, see:

<https://git-scm.com/docs/gitattributes>

- 1 If you do not already have a `.gitattributes` file in your sandbox folder, create one at the MATLAB command prompt:

```
edit .gitattributes
```

- 2 Add these lines to the `.gitattributes` file:

```
*.mlx -crlf -diff -merge  
*.mat -crlf -diff -merge  
*.fig -crlf -diff -merge  
*.p -crlf -diff -merge  
*.slx -crlf -diff -merge  
*.mdl -crlf -diff -merge
```

These lines specify not to try automatic line feed, diff, and merge attempts for these types of files.

- 3 Check for other file types you use that you also need to register as binary to avoid corruption at check-in. Check for files such as `.mdl`, `.slxp`, MEX-files (`.mexa64`, `.mexmaci64`, `.mexw64`), `.xlsx`, `.jpg`, `.pdf`, `.docx`, etc. Add a line to the attributes file for each file type you need.

Examples:

```
*.mdl -crlf -diff -merge  
*.slxp -crlf -diff -merge  
*.sldd -crlf -diff -merge  
*.mexa64 -crlf -diff -merge  
*.mexw64 -crlf -diff -merge  
*.mexmaci64 -crlf -diff -merge  
*.xlsx -crlf -diff -merge  
*.docx -crlf -diff -merge  
*.pdf -crlf -diff -merge  
*.jpg -crlf -diff -merge  
*.png -crlf -diff -merge
```

- 4 Restart MATLAB so you can start using the Git client.

Add Git Submodules

To reuse code from another repository, you can specify Git submodules.

To clone an external Git repository as a submodule:

- 1 Right-click in the MATLAB Current Folder browser, and select **Source Control > Submodules**.
- 2 In the Submodules dialog box, click the **+** button.
- 3 In the Add Submodule dialog box, in the **Remote** box, specify a repository location. Optionally, click **Validate**.
- 4 In the **Path** box, specify a location for the submodule and click **OK**. The Submodules dialog box displays the status and details of the submodule.
- 5 Check the status message, and click **Close**.

Use Fetch to Get Latest Submodule Version

When you want to manage the added submodule, open the Submodules dialog box.

- 1 To get the latest version of a submodule, in the Submodules dialog box, click **Fetch**.
- 2 After fetching, you must merge. Check the **Status** message in the Submodules dialog box for information about your current branch relative to the remote tracking branch in the repository. When you see the message **Behind**, you need to merge in changes from the repository to your local branch.
- 3 Click **Branches** and merge in the origin changes to your local branch using the Branches dialog box. See “Fetch and Merge” on page 31-46.

Use Push to Send Changes to the Submodule Repository

If you make changes in your submodule and want to send changes back to the repository:

- 1 Perform a local commit in the parent folder.
- 2 Open the Submodules dialog box and click **Push**.

If you want other users to obtain your changes in the submodule when they clone the parent folder, make sure the index and head match.

- 1 In the Submodules dialog box, check the index and head values. The index points to the head commit at the time you first cloned the submodule, or when you last

committed the parent folder. If the index and head do not match, you must update the index.

- 2 To update the index, commit your changes in the parent folder, and then click **Push** in the Submodules dialog box. This action makes the index and head the same.

See Also

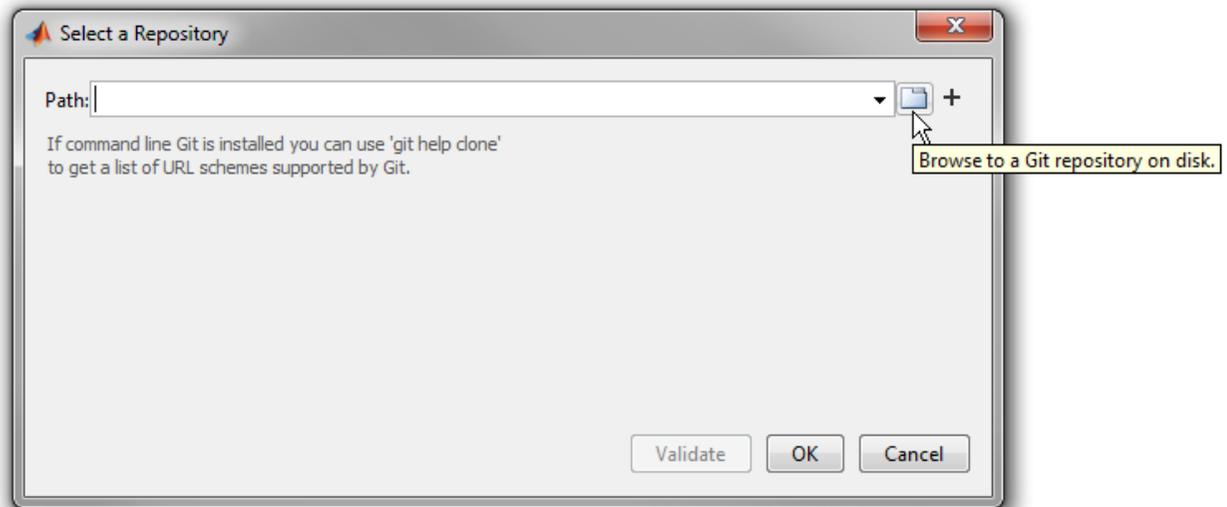
Related Examples

- “Clone from Git Repository” on page 31-36

Clone from Git Repository

Clone a remote Git repository to retrieve repository files.

- 1 Right-click in the white space (any blank area) in the Current Folder browser, and select **Source Control > Manage Files**.
- 2 In the Manage Files Using Source Control dialog box, select **Git** from the **Source control integration** list.
- 3 Click **Change** to browse for and validate the repository path. (If you know your repository location, you can paste it into the **Repository Path** box and proceed to step 7.)
- 4 In the Select a Repository dialog box, specify the repository path by entering the path in the box, using the list of recent repositories, or by using the **Browse to a Git repository on disk** repository on disk button .



- 5 Click **Validate** to check the repository path.

If the path is invalid, check it against your source control repository browser.

- 6 If you see an authentication dialog box for your repository, enter login information to continue.

- 7 When you have finished specifying the path you want to retrieve, click **OK**.
- 8 In the Manage Files Using Source Control dialog box, select the sandbox folder where you want to put the retrieved files, and click **Retrieve**.

Troubleshooting

If you encounter errors like `OutOfMemoryError: Java heap space`, for example when cloning big Git repositories, then edit your MATLAB preferences to increase the heap size.

- 1 On the **Home** tab, in the **Environment** section, click **Preferences**.
- 2 Select **MATLAB > General > Java Heap Memory**.
- 3 Move the slider to increase the heap size, and then click **OK**.
- 4 Restart MATLAB.

See Also

Related Examples

- “Set Up Git Source Control” on page 31-30
- “Update Git File Status and Revision” on page 31-38
- “Branch and Merge with Git” on page 31-40

Update Git File Status and Revision

In this section...
“Refresh Status of Files” on page 31-38
“Update Revisions of Files” on page 31-38

Refresh Status of Files

To refresh the source control status of files, select one or more files in the Current Folder browser, right-click and select **Source Control > Refresh Git status**.

To refresh the status of all files in the repository, right-click the white space of the Current Folder browser and select **Source Control > Refresh Git status**.

Update Revisions of Files

To update all files in a repository, right-click in the Current Folder browser and select **Source Control > Pull**.

Caution Ensure you have registered binary files with Git before using **Pull**. If you do not, conflict markers can corrupt your files. For more information, see “Register Binary Files with Git” on page 31-32.

Pull fetches the latest changes and merges them into your current branch. If you are not sure what is going to come in from the repository, use fetch to examine the changes first and then merge the changes manually. For more information, see “Pull, Push and Fetch Files with Git” on page 31-45.

Pull might fail if you have conflicts. With a complicated change you might want to create a branch from the origin, make some compatibility changes, then merge that branch into the main tracking branch.

See Also

Related Examples

- “Clone from Git Repository” on page 31-36
- “Review Changes in Source Control” on page 31-9

Branch and Merge with Git

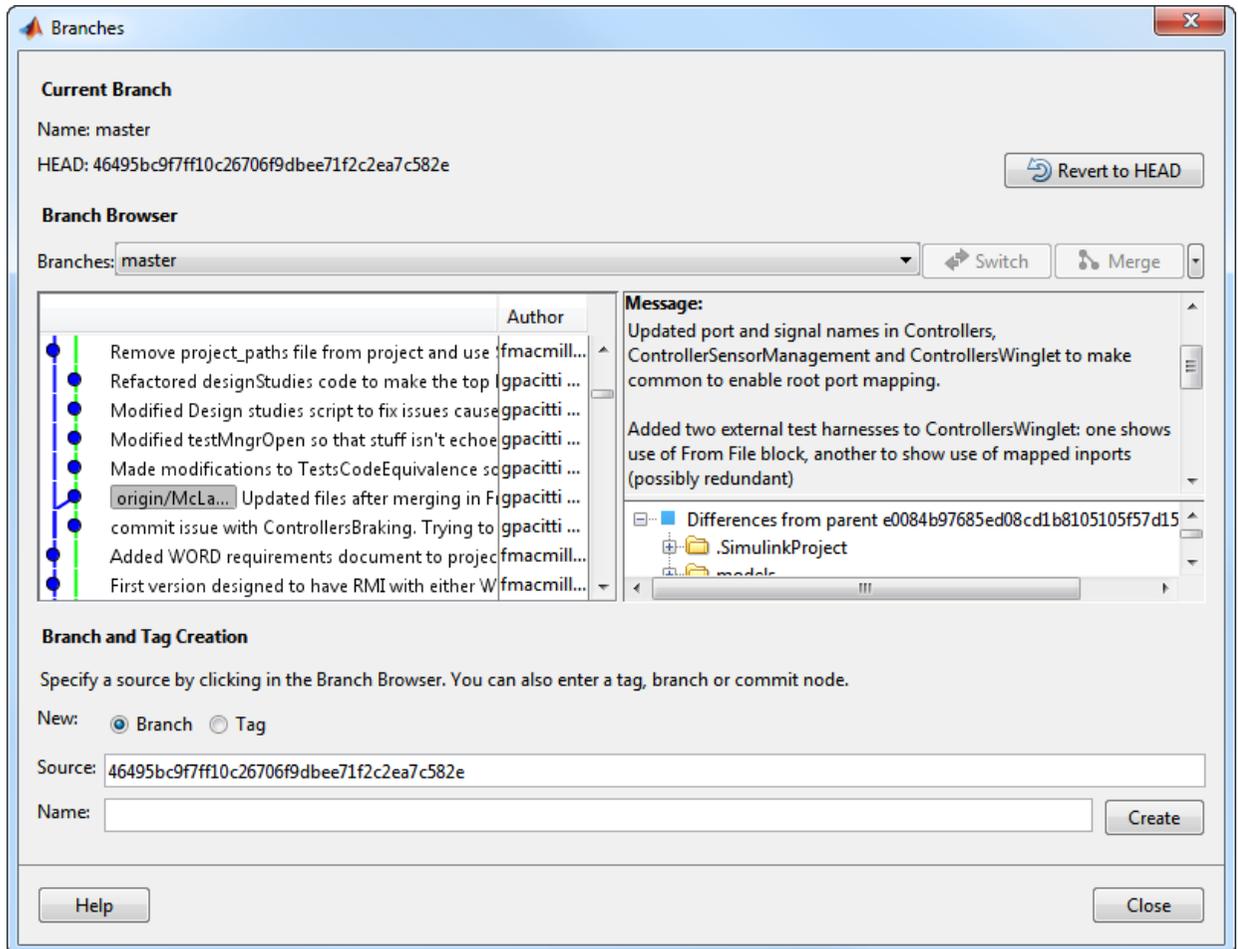
In this section...
“Create Branch” on page 31-40
“Switch Branch” on page 31-42
“Compare Branches” on page 31-42
“Merge Branches” on page 31-42
“Revert to Head” on page 31-43
“Delete Branches” on page 31-43

Create Branch

- 1 From within your Git repository folder, right-click the white space of the Current Folder browser and select **Source Control > Branches**. In the Branches dialog box, you can view, switch, create, and merge branches.

Tip You can inspect information about each commit node. Select a node in the **Branch Browser** diagram to view the author, date, commit message, and changed files.

The **Branch Browser** in this figure shows an example branch history.



- 2 Select a source for the new branch. Click a node in the **Branch Browser** diagram, or enter a unique identifier in the **Source** text box. You can enter a tag, branch name, or a unique prefix of the SHA1 hash (for example, 73c637 to identify a specific commit). Leave the default to create a branch from the head of the current branch.
- 3 Enter a name in the **Branch name** text box and click **Create**.
- 4 To work on the files on your new branch, switch your project to the branch.

In the **Branches** drop-down list, select the branch you want to switch to and click **Switch**.

- 5 Close the Branches dialog box and work on the files on your branch.

For next steps, see “Pull, Push and Fetch Files with Git” on page 31-45.

Switch Branch

- 1 From within your Git repository folder, right-click the white space of the Current Folder browser and select **Source Control > Branches**.
- 2 In the Branches dialog box, in the **Branches** drop-down list, select the branch you want to and click **Switch**.
- 3 Close the Branches dialog box and work on the files on your branch.

Compare Branches

From within your Git repository folder, right-click the white space of the Current Folder browser and select **Source Control > Branches**. In the Branches dialog box, to examine differences from the parent, right-click a file in the tree under **Differences from parent** and select **Show Difference**. MATLAB opens a comparison report.

If you want to examine added or deleted files, or to test how the code ran in previous versions, you can save a copy of the selected or parent files. Right-click a file and select **Save As** or **Save Parent As**.

Merge Branches

Before you can merge branches, you must install command-line Git on your system path and register binary files to prevent Git from inserting conflict markers. See “Install Command-Line Git Client” on page 31-31.

Tip After you fetch changes, you must merge. For more information, see “Fetch and Merge” on page 31-46.

To merge any branches:

- 1 From within your Git repository folder, right-click the white space of the Current Folder browser and select **Source Control** and **Branches**.
- 2 In the Branches dialog box, from the **Branches** drop-down list, select a branch you want to merge into the current branch, and click **Merge**.

- 3 Close the Branches dialog box and work on the files on your branch.

If the branch merge causes a conflict that Git cannot resolve automatically, an error dialog box reports that automatic merge failed. Resolve the conflicts before proceeding.

Caution Do not move or delete files outside of MATLAB because this can cause errors on merge.

Keep Your Version

- 1 To keep your version of the file, right-click the file and select **Mark Conflict Resolved**.
- 2 Click **Commit Modified Files** to commit your change that marks the conflict resolved.

View Conflicts in Branch Versions

If you merge a branch and there is a conflict in a file, Git marks the file as conflicted and does not modify the contents. Right-click the file and select **Source Control > View Conflicts**. A comparison report opens that shows the differences between the file on your branch and the branch you want to merge into. Decide how to resolve the conflict. See “Resolve Source Control Conflicts” on page 31-11.

Revert to Head

- 1 From within your Git repository folder, right-click the white space of the Current Folder browser and select **Source Control > Branches**.
- 2 In the Branches dialog box, click **Revert to Head** to remove all local changes.

Delete Branches

- 1 In the Branches dialog box under **Branch Browser**, expand the **Branches** drop-down list, and select the branch you want to delete.
- 2 On the far right, click the down arrow and select **Delete Branch**.

Caution You cannot undo branch deletion.

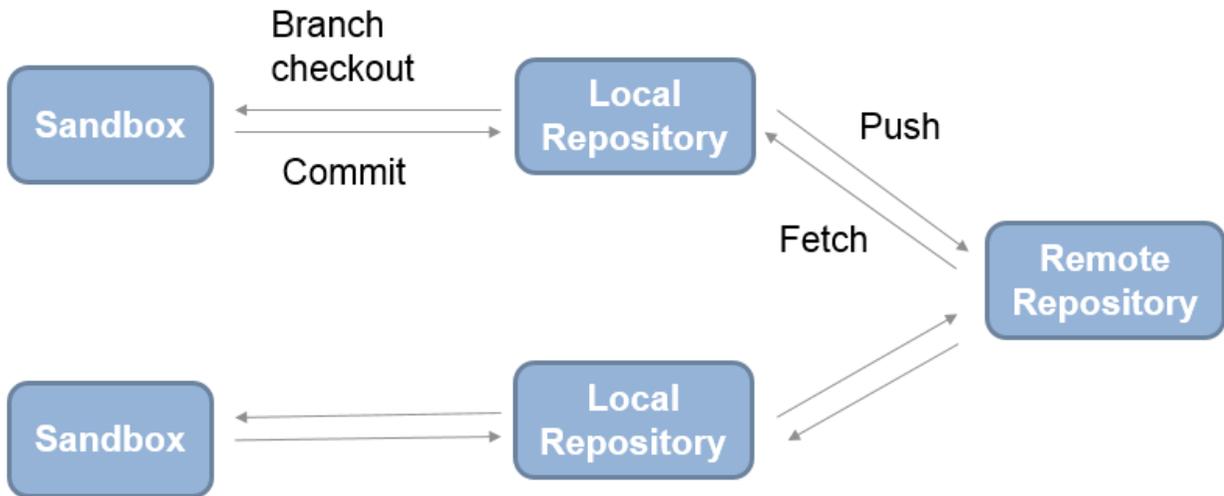
See Also

Related Examples

- “Set Up Git Source Control” on page 31-30
- “Pull, Push and Fetch Files with Git” on page 31-45
- “Resolve Source Control Conflicts” on page 31-11

Pull, Push and Fetch Files with Git

Use this workflow to work with a remote repository. With Git, there is a two-step workflow: commit local changes, and then push to the remote repository. In MATLAB, the only access to the remote repository is through the **Pull**, **Push** and **Fetch** menu options. All other actions, such as **Compare to Ancestor** and **Commit**, use the local repository. This diagram represents the Git workflow.



Pull and Push

To get the latest changes, right-click the Current Folder browser and select **Source Control > Pull**. Pull fetches the latest changes and merges them into your current branch. If you are not sure what is going to come in from the repository, use fetch to examine the changes first and then merge the changes manually.

Note Before you can merge, you must install command-line Git and register binary files to prevent Git from inserting conflict markers. See “Install Command-Line Git Client” on page 31-31.

Pull might fail if you have conflicts. With a complicated change you might want to create a branch from the origin, make some compatibility changes, then merge that branch into the main tracking branch.

To commit changes to the local repository, right-click the Current Folder browser and select **Source Control > View and Commit Changes**.

To see if your local changes have moved ahead of the remote tracking branch, right-click the file or white space of the Current Folder browser and select **Source Control > View Details**. The **Git information** field indicates whether your committed local changes are ahead of, behind, or coincident with the remote tracking branch.

To send local commits to the remote repository, right-click in the Current Folder browser and select **Source Control > Push**. A message appears if you cannot push your changes directly because the repository has moved on. Right-click in the Current Folder browser and select **Source Control > Fetch** to fetch all changes from the remote repository. Merge branches and resolve conflicts, and then you can push your changes.

Using Git, you cannot add empty folders to source control, so you cannot select **Push** and then clone an empty folder. You can create an empty folder in MATLAB, but if you push changes and then sync a new sandbox, then the empty folder does not appear in the new sandbox. To push empty folders to the repository for other users to sync, create a gitignore file in the folder and then push your changes.

Fetch and Merge

Use **Fetch** to get changes and merge manually. Use **Pull** instead to fetch the latest changes and merge them into your current branch.

Note After fetching, you must merge. Before you can merge branches, you must install command-line Git and register binary files to prevent Git from inserting conflict markers. See “Install Command-Line Git Client” on page 31-31.

To fetch changes from the remote repository, right-click in the Current Folder browser and select **Source Control > Fetch**. Fetch updates all of the origin branches in the local repository. Your sandbox files do not change. To see others’ changes, you need to merge in the origin changes to your local branches.

For information about your current branch relative to the remote tracking branch in the repository, right-click the file or white space of the Current Folder browser and select

Source Control > View Details. The **Git information** field indicates whether your committed local changes are ahead of, behind, or coincident with the remote tracking branch. When you see the message **Behind**, you need to merge in changes from the repository to your local branch.

For example, if you are on the master branch, get all changes from the master branch in the remote repository.

- 1 Right-click in the Current Folder browser and select **Source Control > Fetch**
- 2 Right-click in the Current Folder browser and select **Source Control > Branches**.
- 3 In the Branches dialog box, select **origin/master** in the **Branches** list.
- 4 Click **Merge**. The origin branch changes merge into the master branch in your sandbox.

If you right-click the Current Folder browser and select **Source Control > View Details**, the **Git information** field indicates **Coincident with /origin/master**. You can now view the changes that you fetched and merged from the remote repository in your local sandbox.

Use Git Stashes

Store uncommitted changes for later use by creating a Git stash. Use stashes to:

- Store modified files without committing them.
- Move changes easily to a new branch.
- Browse and examine the changes within a stash.

To create and manage stashes, in the Current Folder browser, right-click the white space in a folder managed by Git and select **Source Control > Stashes**.

In the Stashes dialog box:

- To create a stash containing your currently modified files, click **New Stash**.
- To view modified files in a stash, select the stash under **Available Stashes**. Right-click modified files to view changes or save a copy.
- To apply the stash to your current branch and then delete the stash, click **Pop**.
- To apply the stash and keep it, click **Apply**.
- To delete the stash, click **Drop**.

See Also

Related Examples

- “Branch and Merge with Git” on page 31-40
- “Resolve Source Control Conflicts” on page 31-11

Move, Rename, or Delete Files Under Source Control

Move, rename, or delete files using the MATLAB Source Control context menu options or another source control client application.

To move a file under source control, right-click the file in the Current Folder browser, select **Source Control > Move**, and enter a new file location.

To rename a file under source control, right-click the file in the Current Folder browser, select **Source Control > Rename**, and enter a new file name.

To delete a file from the repository, mark the file for deletion.

- To mark a file for deletion from the repository and retain a local copy, right-click the file in the Current Folder browser. Select **Source Control** and then **Delete from SVN** or **Delete from Git**. When the file is marked for deletion from source control, the symbol changes to Deleted . The file is removed from the repository at the next commit.
- To mark a file for deletion from the repository and from your disk, right-click the file in the Current Folder browser. Select **Source Control** and then **Delete from SVN and disk** or **Delete from Git and disk**. The file disappears from the Current Folder browser and is immediately deleted from your disk. The file is removed from the repository at the next commit.

See Also

Related Examples

- “Mark Files for Addition to Source Control” on page 31-10
- “Commit Modified Files to Source Control” on page 31-15

Customize External Source Control to Use MATLAB for Diff and Merge

You can customize external source control tools to use the MATLAB Comparison Tool for diff and merge. If you want to compare MATLAB files such as live scripts, MAT, SLX, or MDL files from your source control tool, then you can configure your source control tool to open the MATLAB Comparison Tool.

MATLAB Comparison Tool provides useful merge tools for MathWorks files and is compatible with all popular software configuration management and version control systems.

Set up your source control tool to use MATLAB as the application for diff and merge for the file extensions you want, for example, `.mlx`, `.mat`, `.slx`, or `.mdl`, by following these steps:

- 1 To get the required file paths and set the preference to reuse open MATLAB sessions, run this command in MATLAB:

```
comparisons.ExternalSCMLink.setup()
```

This command sets the MATLAB preference, under **Comparison**, called **Allow external source control tools to use open MATLAB sessions for diffs and merges**.

The command also displays the file paths you will copy and paste into your source control tool setup:

- On Windows:

```
matlabroot\bin\win64\mlDiff.exe
```

```
matlabroot\bin\win64\mlMerge.exe
```

- On Linux:

```
matlabroot/bin/glnxa64/mlDiff
```

```
matlabroot/bin/glnxa64/mlMerge
```

On Mac:

```
matlabroot/bin/maci64/mlDiff
```

```
matlabroot/bin/maci64/mlMerge
```

Where *matlabroot* is replaced with the full path to your installation, for example, C:\Program Files\MATLAB\R2016b.

2 Set up diff.

- a In the MATLAB Command Window, copy the file path to `mlDiff`, for example, C:\Program Files\MATLAB\R2016b\bin\win64\mlDiff.exe.
- b In your source control tool, locate the diff setting, and add an entry to specify what to do with a particular file extension (for example, `.slx`). Paste in the file path to `mlDiff` that you copied from the MATLAB Command Window.
- c After the path to the script, or in the arguments box, add arguments to specify the input files. Look up the argument names specific to your source control tool. Specify the inputs for diffs, in this order: *leftFile*, *rightFile*.

For example, for Tortoise SVN:

```
"C:\Program Files\MATLAB\R2016b\bin\win64\mlDiff.exe" %base %mine
```

For Perforce® P4V:

```
"C:\Program Files\MATLAB\R2016b\bin\win64\mlDiff.exe" %1 %2
```

3 Set up merge.

- a In the MATLAB Command Window, copy the file path to `mlMerge`.
- b In your source control tool, locate the merge setting, and add an entry to specify what to do with a particular file extension (for example, `.slx`). Paste in the file path to `mlMerge` that you copied from the MATLAB Command Window.
- c After the path to the script, or in the arguments box, add arguments to specify the input files. Look up the argument names specific to your source control tool. Specify the inputs for merges, in this order: *base*, *mine*, *theirs*, and *merged* target file.

For example, for Tortoise SVN:

```
"C:\Program Files\MATLAB\R2016b\bin\win64\mlMerge.exe" %base %mine %theirs %merged
```

For Perforce P4V:

```
"C:\Program Files\MATLAB\R2016b\bin\win64\mlMerge.exe" %b %2 %1 %r
```

- 4 After this setup, when you use diff or merge, your external source control tool opens a report in MATLAB Comparison Tool. Use the report to view changes and resolve merges.

Your diff and merge operations use open MATLAB sessions if available, and only open MATLAB when necessary. The comparison only uses the specified MATLAB installation.

See Also

Related Examples

- “Compare and Merge Text”
- “Compare and Merge MAT-Files”
- “Compare Variables”
- “Merge Simulink Models from the Comparison Report” (Simulink)

MSSCCI Source Control Interface

Note MSSCCI support has been removed. Replace this functionality with one of the following options.

- Use a source control system that is part of the MathWorks “Source Control Integration” with the Current Folder browser.
 - Use the Source Control Software Development Kit to create a plug-in for your source control.
 - Use the MATLAB `system` function to access the command-line API for your source control tool. This option does not provide integration with the MATLAB Current Folder browser menus or source control status column.
-

If you use source control systems to manage your files, you can interface with the systems to perform source control actions from within the MATLAB, Simulink, and Stateflow products. Use menu items in the MATLAB, Simulink, or Stateflow products, or run functions in the MATLAB Command Window to interface with your source control systems.

The source control interface on Windows works with any source control system that conforms to the Microsoft Common Source Control standard, Version 1.1. If your source control system does not conform to the standard, use a Microsoft Source Code Control API wrapper product for your source control system so that you can interface with it from the MATLAB, Simulink, and Stateflow products.

This documentation uses the Microsoft Visual SourceSafe® software as an example. Your source control system might use different terminology and not support the same options or might use them in a different way. Regardless, you should be able to perform similar actions with your source control system based on this documentation.

Perform most source control interface actions from the Current Folder browser. You can also perform many of these actions for a single file from the MATLAB Editor, a Simulink model window, or a Stateflow chart window—for more information, see “Access MSSCCI Source Control from Editors” on page 31-71.

Set Up MSSCCI Source Control

Note MSSCCI support has been removed. Replace this functionality with one of the following options.

- Use a source control system that is part of the MathWorks “Source Control Integration” with the Current Folder browser.
 - Use the Source Control Software Development Kit to create a plug-in for your source control.
 - Use the MATLAB `system` function to access the command-line API for your source control tool. This option does not provide integration with the MATLAB Current Folder browser menus or source control status column.
-

In this section...

“Create Projects in Source Control System” on page 31-54
--

“Specify Source Control System with MATLAB Software” on page 31-56
--

“Register Source Control Project with MATLAB Software” on page 31-57
--

“Add Files to Source Control” on page 31-59

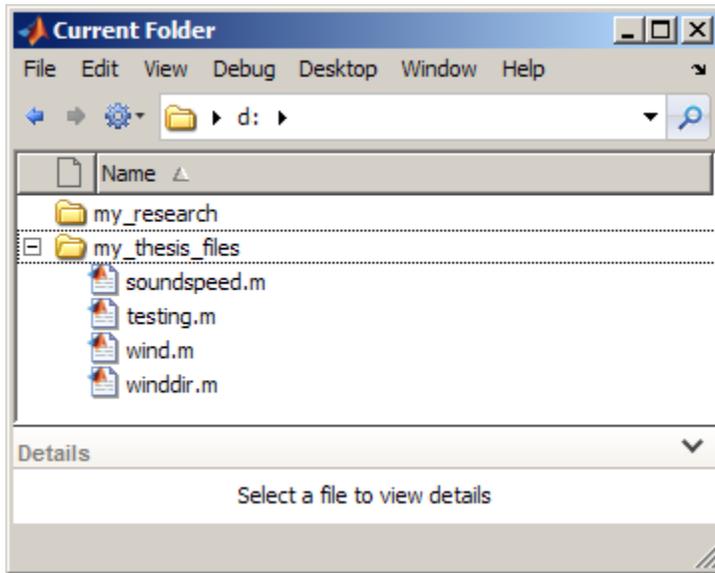
Create Projects in Source Control System

In your source control system, create the projects that your folders and files will be associated with.

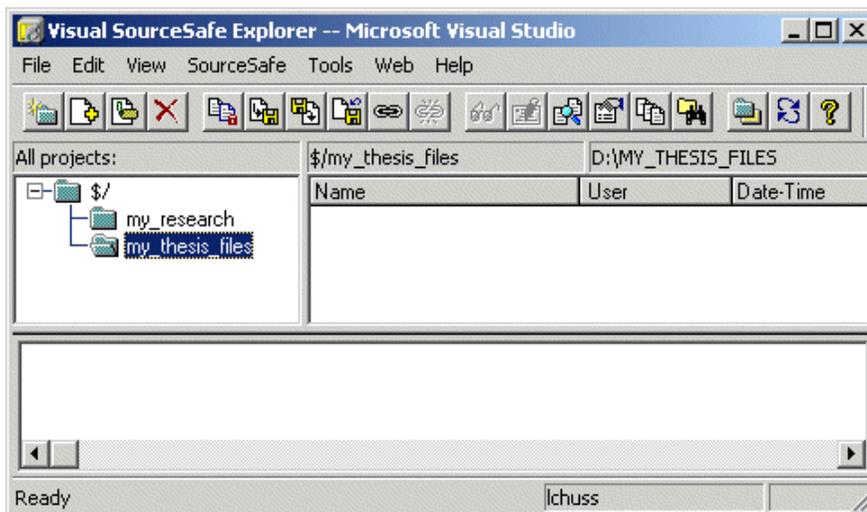
All files in a folder must belong to the same source control project. Be sure the working folder for the project in the source control system specifies the correct path to the folder on disk.

Example of Creating Source Control Project

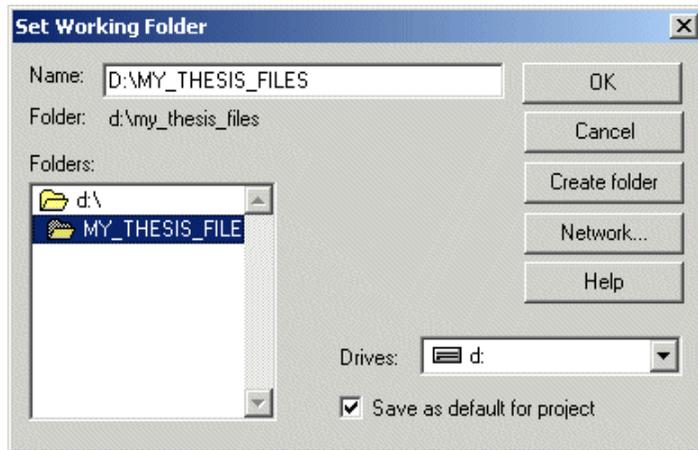
This example uses the project `my_thesis_files` in Microsoft Visual SourceSafe. This illustration of the Current Folder browser shows the path to the folder on disk, `D:\my_thesis_files`.



The following illustration shows the example project in the source control system.



To set the working folder in Microsoft Visual SourceSafe for this example, select `my_thesis_files`, right-click, select **Set Working Folder** from the context menu, and specify `D:\my_thesis_files` in the resulting dialog box.

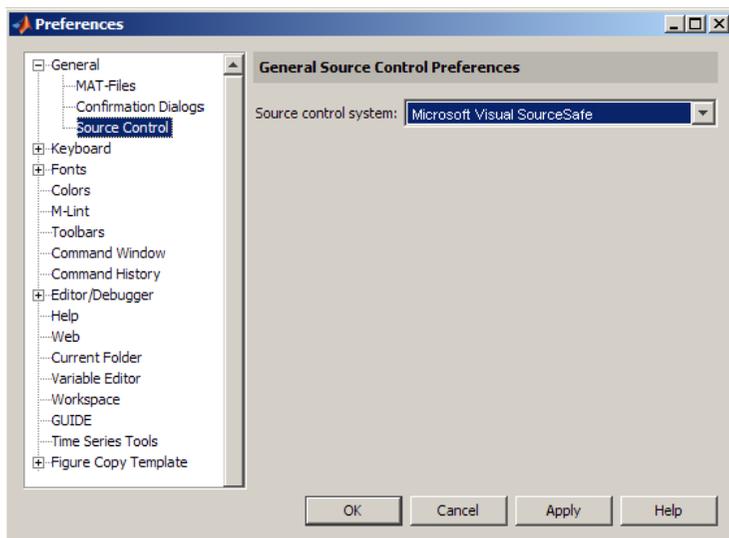


Specify Source Control System with MATLAB Software

In MATLAB, specify the source control system you want to access. On the **Home** tab, in the **Environment** section, click **Preferences > MATLAB > General > Source Control**.

The currently selected system is shown in the Preferences dialog box. The list includes all installed source control systems that support the Microsoft Common Source Control standard.

Select the source control system you want to interface with and click **OK**.



MATLAB remembers preferences between sessions, so you only need to perform this action again when you want to access a different source control system.

Source Control with 64-Bit Versions of MATLAB

If you run a 64-bit version of MATLAB and want MATLAB to interface with your source control system, your source control system must be 64-bit compliant. If you have a 32-bit source control system, or if you have a 64-bit source control system running in 32-bit compatibility mode, MATLAB cannot use it. In that event, MATLAB displays a warning about the problem in the Source Control preference pane.

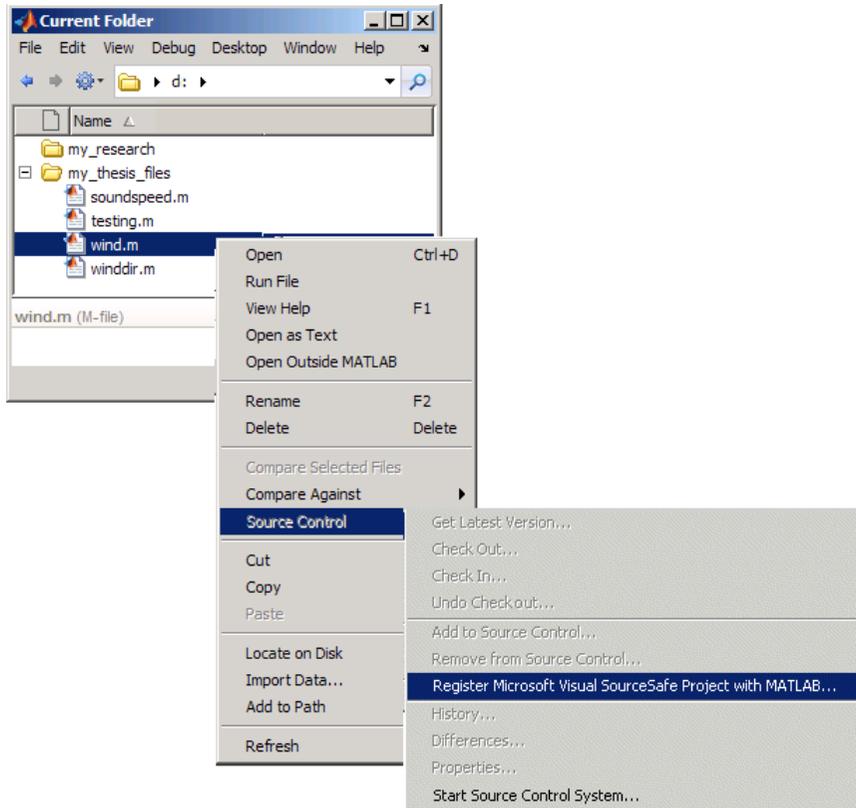
Register Source Control Project with MATLAB Software

Register a source control system project with a folder in MATLAB, that is, associate a source control system project with a folder and all files in that folder. Do this only one time for any file in the folder, which registers all files in that folder:

- 1 In the MATLAB Current Folder browser, select a file that is in the folder you want to associate with a project in your source control system. For example, select D:\my_thesis_files\wind.m. This will associate all files in the my_thesis_files folder.
- 2 Right-click, and from the context menu, select **Source Control > Register Name_of_Source_Control_System Project with MATLAB**. The

Name_of_Source_Control_System is the source control system you selected using preferences as described in “Specify Source Control System with MATLAB Software” on page 31-56.

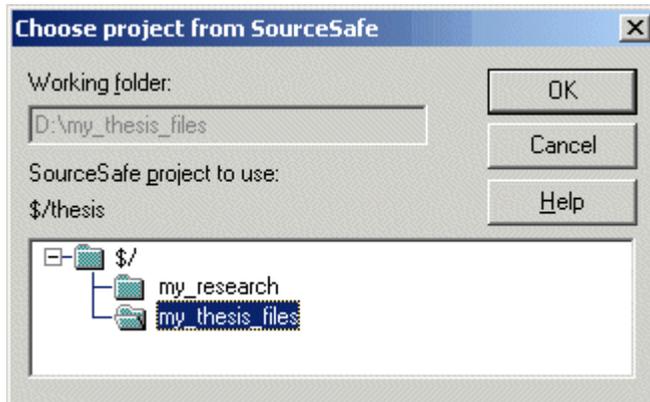
The following example shows Microsoft Visual SourceSafe.



- 3 In the resulting **Name_of_Source_Control_System Login** dialog box, provide the user name and password you use to access your source control system, and click **OK**.



- 4 In the resulting **Choose project from Name_of_Source_Control_System** dialog box, select the source control system project to associate with the folder and click **OK**. This example shows `my_thesis_files`.



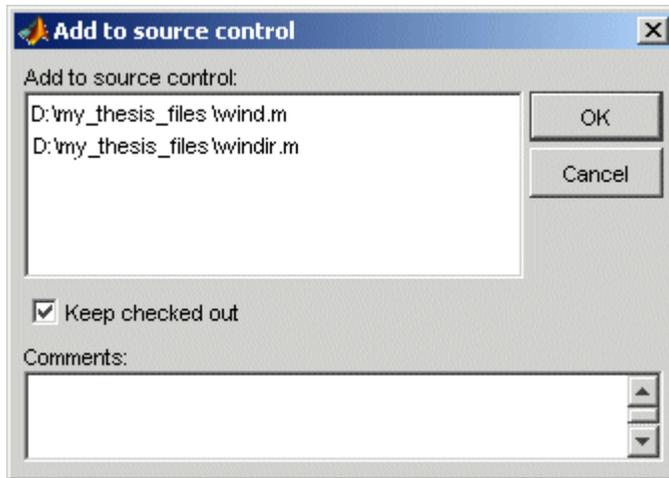
The selected file, its folder, and all files in the folder, are associated with the source control system project you selected. For the example, MATLAB associates all files in `D:\my_thesis_files` with the source control project `my_thesis_files`.

Add Files to Source Control

Add files to the source control system. Do this only once for each file:

- 1 In the Current Folder browser, select files you want to add to the source control system.
- 2 Right-click, and from the context menu, select **Source Control > Add to Source Control**.

- 3 The resulting **Add to source control** dialog box lists files you selected to add. You can add text in the **Comments** field. If you expect to use the files soon, select the **Keep checked out** check box (which is selected by default). Click **OK**.



If you try to add an unsaved file, the file is automatically saved upon adding.

Check Files In and Out from MSSCCI Source Control

Note MSSCCI support has been removed. Replace this functionality with one of the following options.

- Use a source control system that is part of the MathWorks “Source Control Integration” with the Current Folder browser.
 - Use the Source Control Software Development Kit to create a plug-in for your source control.
 - Use the MATLAB `system` function to access the command-line API for your source control tool. This option does not provide integration with the MATLAB Current Folder browser menus or source control status column.
-

In this section...
“Check Files Into Source Control” on page 31-61
“Check Files Out of Source Control” on page 31-62
“Undoing the Checkout” on page 31-63

Before checking files into and out of your source control system from the MATLAB desktop, be sure to set up your system for use with MATLAB as described in “Set Up MSSCCI Source Control” on page 31-54.

Check Files Into Source Control

After creating or modifying files using MATLAB software or related products, check the files into the source control system by performing these steps:

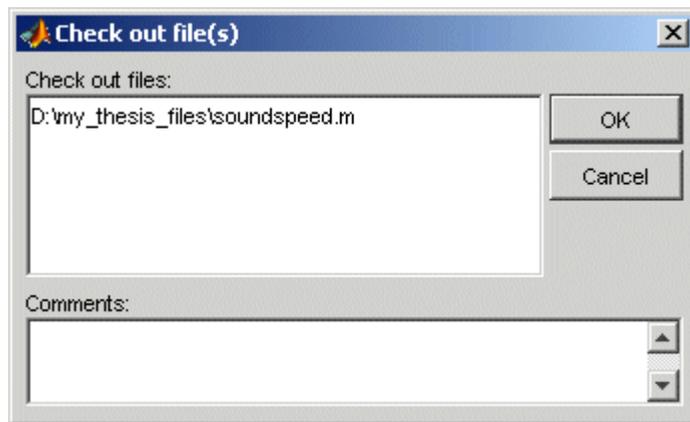
- 1** In the Current Folder browser, select the files to check in. A file can be open or closed when you check it in, but it must be saved, that is, it cannot contain unsaved changes.
- 2** Right-click, and from the context menu, select **Source Control > Check In**.
- 3** In the resulting **Check in file(s)** dialog box, you can add text in the **Comments** field. If you want to continue working on the files, select the check box **Keep checked out**. Click **OK**.

If a file contains unsaved changes when you try to check it in, you will be prompted to save the changes to complete the checkin. If you did not keep the file checked out and you keep the file open, note that it is a read-only version.

Check Files Out of Source Control

From MATLAB, to check out the files you want to modify, perform these steps:

- 1 In the Current Folder browser, select the files to check out.
- 2 Right-click, and from the context menu, select **Source Control > Check Out**.
- 3 The resulting **Check out file(s)** dialog box lists files you selected to check out. Enter comment text in the **Comments** field, which appears if your source control system supports comments on checkout. Click **OK**.



After checking out a file, make changes to it in MATLAB or another product, and save the file. For example, edit a file in the Editor.

If you try to change a file without first having checked it out, the file is read-only, as seen in the title bar, and you will not be able to save any changes. This protects you from accidentally overwriting the source control version of the file.

If you end the MATLAB session, the file remains checked out. You can check in the file from within MATLAB during a later session, or folder from your source control system.

Undoing the Checkout

You can undo the checkout for files. The files remain checked in, and do not have any of the changes you made since you last checked them out. To save any changes you have made since checking out a particular file click **Save**  on the **Editor** or **Live Editor** tab, select **Save As**, and supply a different file name before you undo the checkout.

To undo a checkout, follow these steps:

- 1 In the MATLAB Current Folder browser, select the files for which you want to undo the checkout.
- 2 Right-click, and from the context menu, select **Source Control > Undo Checkout**.

The MATLAB **Undo checkout** dialog box opens, listing the files you selected.



- 3 Click **OK**.

Additional MSSCCI Source Control Actions

Note MSSCCI support has been removed. Replace this functionality with one of the following options.

- Use a source control system that is part of the MathWorks “Source Control Integration” with the Current Folder browser.
 - Use the Source Control Software Development Kit to create a plug-in for your source control.
 - Use the MATLAB `system` function to access the command-line API for your source control tool. This option does not provide integration with the MATLAB Current Folder browser menus or source control status column.
-

In this section...
“Getting the Latest Version of Files for Viewing or Compiling” on page 31-64
“Removing Files from the Source Control System” on page 31-65
“Showing File History” on page 31-66
“Comparing the Working Copy of a File to the Latest Version in Source Control” on page 31-67
“Viewing Source Control Properties of a File” on page 31-69
“Starting the Source Control System” on page 31-69

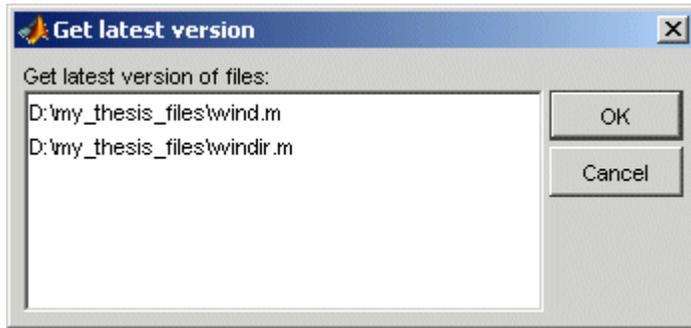
Getting the Latest Version of Files for Viewing or Compiling

You can get the latest version of a file from the source control system for viewing or running. Getting a file differs from checking it out. When you get a file, it is write protected so you cannot edit it, but when you check out a file, you can edit it.

To get the latest version, follow these steps:

- 1 In the MATLAB Current Folder browser, select the folders or files that you want to get. If you select files, you cannot select folders too.
- 2 Right-click, and from the context menu, select **Source Control > Get Latest Version**.

The MATLAB Get latest version dialog box opens, listing the files or folders you selected.



- 3 Click **OK**.

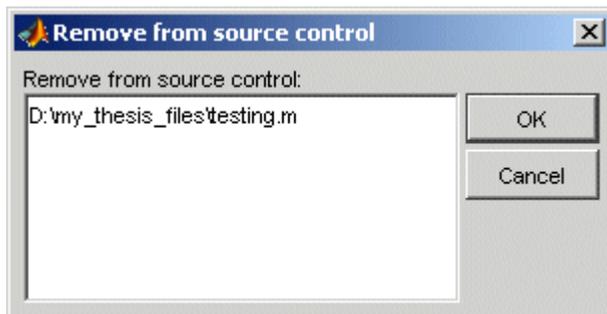
You can now open the file to view it, run the file, or check out the file for editing.

Removing Files from the Source Control System

To remove files from the source control system, follow these steps:

- 1 In the MATLAB Current Folder browser, select the files you want to remove.
- 2 Right-click, and from the context menu, select **Source Control > Remove from Source Control**.

The MATLAB **Remove from source control** dialog box opens, listing the files you selected.



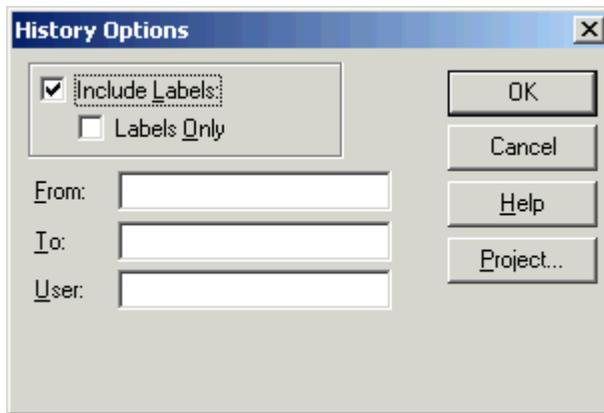
- 3 Click **OK**.

Showing File History

To show the history of a file in the source control system, follow these steps:

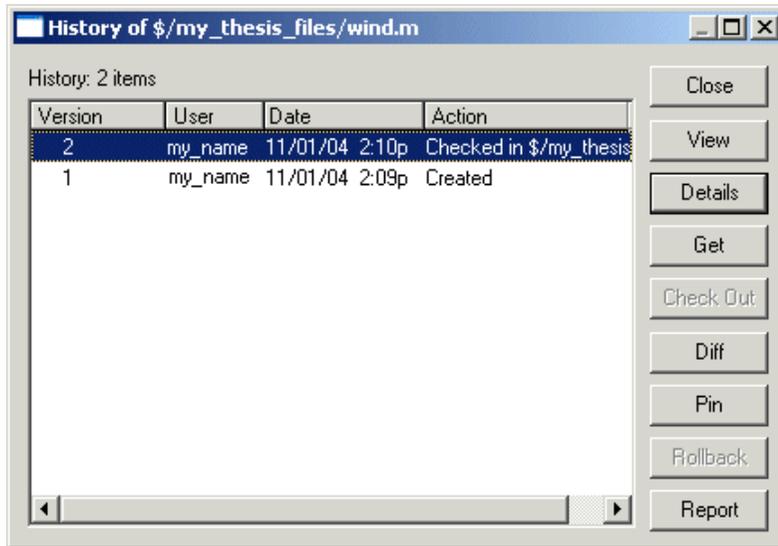
- 1 In the MATLAB Current Folder browser, select the file for which you want to view the history.
- 2 Right-click, and from the context menu, select **Source Control > History**.

A dialog box, which is specific to your source control system, opens. For Microsoft Visual SourceSafe, the **History Options** dialog box opens, as shown in the following example illustration.



- 3 Complete the dialog box to specify the range of history you want for the selected file and click **OK**. For example, enter my_name for **User**.

The history presented depends on your source control system. For Microsoft Visual SourceSafe, the **History** dialog box opens for that file, showing the file's history in the source control system.



Comparing the Working Copy of a File to the Latest Version in Source Control

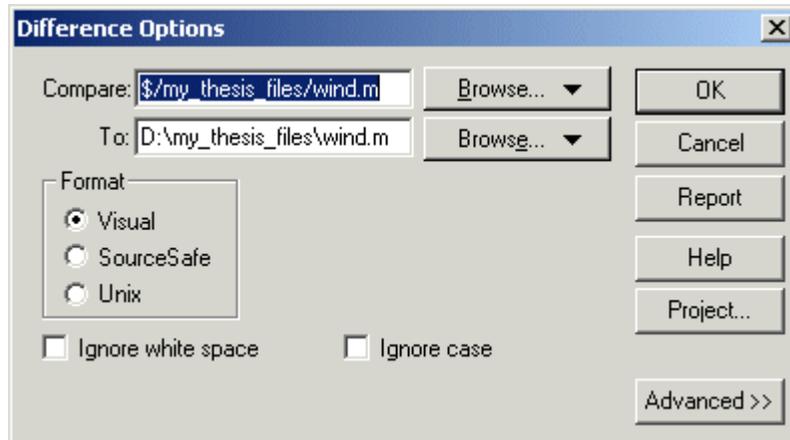
You can compare the current working copy of a file with the latest checked-in version of the file in the source control system. This highlights the differences between the two files, showing the changes you made since you checked out the file.

To view the differences, follow these steps:

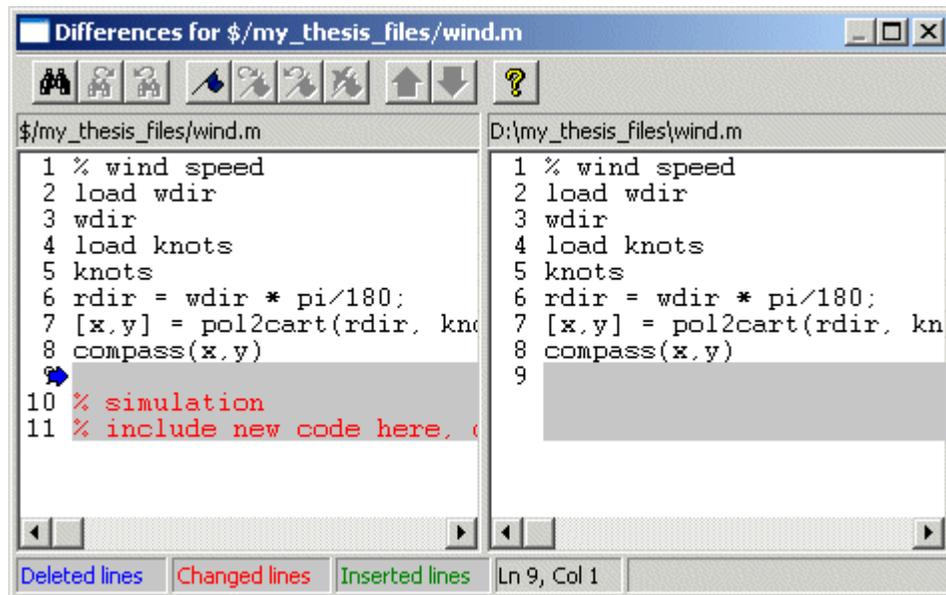
- 1 In the MATLAB Current Folder browser, select the file for which you want to view differences. This is a file that has been checked out and edited.
- 2 Right-click, and from the context menu, select **Source Control > Differences**.

A dialog box, which is specific to your source control system, opens. For Microsoft Visual SourceSafe, the **Difference Options** dialog box opens.

- 3 Review the default entries in the dialog box, make any needed changes, and click **OK**. The following example is for Microsoft Visual SourceSafe.



The method of presenting differences depends on your source control system. For Microsoft Visual SourceSafe, the **Differences for** dialog box opens. This highlights the differences between the working copy of the file and the latest checked-in version of the file.

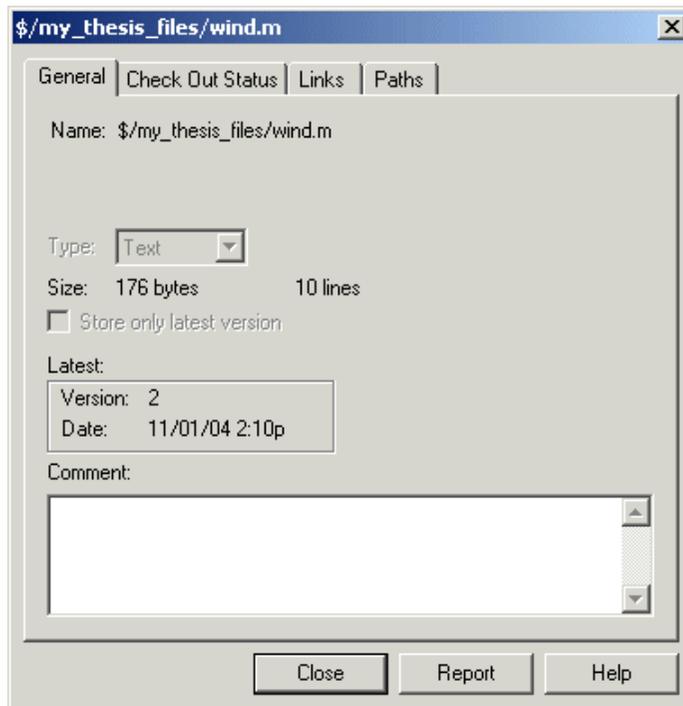


Viewing Source Control Properties of a File

To view the source control properties of a file, follow these steps:

- 1 In the MATLAB Current Folder browser, select the file for which you want to view properties.
- 2 Right-click, and from the context menu, select **Source Control > Properties**.

A dialog box, which is specific to your source control system, opens. The following example shows the Microsoft Visual SourceSafe properties dialog box.

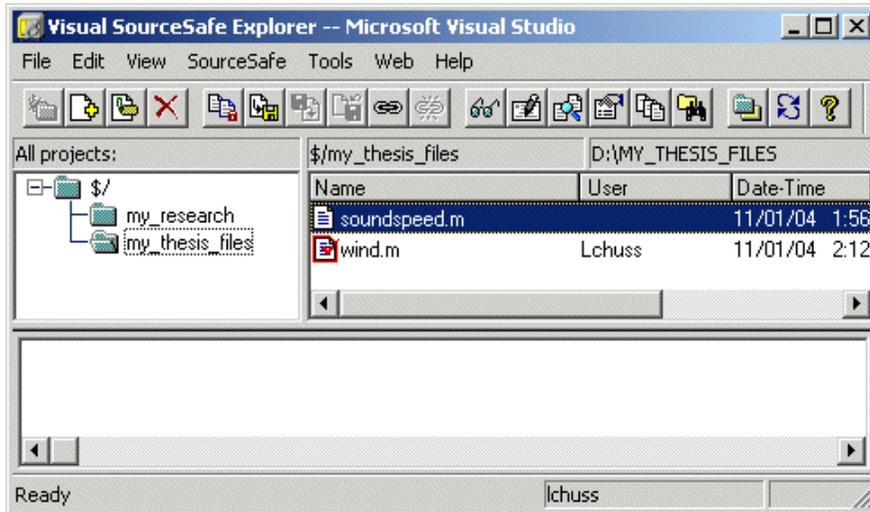


Starting the Source Control System

All the MATLAB source control actions automatically start the source control system to perform the action, if the source control system is not already open. If you want to start the source control system from MATLAB without performing a specific action source control action,

- 1 Right-click any folder or file in the MATLAB Current Folder browser
- 2 From the context menu, select **Source Control > Start Source Control System**.

The interface to your source control system opens, showing the source control project associated with the current folder in MATLAB. The following example shows the Microsoft Visual SourceSafe Explorer interface.



Access MSSCCI Source Control from Editors

Note MSSCCI support has been removed. Replace this functionality with one of the following options.

- Use a source control system that is part of the MathWorks “Source Control Integration” with the Current Folder browser.
 - Use the Source Control Software Development Kit to create a plug-in for your source control.
 - Use the MATLAB `system` function to access the command-line API for your source control tool. This option does not provide integration with the MATLAB Current Folder browser menus or source control status column.
-

You can create or open a file in the Editor, the Simulink or Stateflow products and perform most source control actions from their **File > Source Control** menus, rather than from the Current Folder browser. Following are some differences in the source control interface process when you use the Editor, Simulink, or Stateflow:

- You can perform actions on only one file at time.
- Some of the dialog boxes have a different icon in the title bar. For example, the **Check out file(s)** dialog box uses the MATLAB Editor icon instead of the MATLAB icon.
- You cannot add a new (Untitled) file, but must instead first save the file.
- You cannot register projects from the Simulink or Stateflow products. Instead, register a project using the Current Folder browser, as described in “Register Source Control Project with MATLAB Software” on page 31-57.

Troubleshoot MSSCCI Source Control Problems

Note MSSCCI support has been removed. Replace this functionality with one of the following options.

- Use a source control system that is part of the MathWorks “Source Control Integration” with the Current Folder browser.
 - Use the Source Control Software Development Kit to create a plug-in for your source control.
 - Use the MATLAB `system` function to access the command-line API for your source control tool. This option does not provide integration with the MATLAB Current Folder browser menus or source control status column.
-

In this section...

“Source Control Error: Provider Not Present or Not Installed Properly” on page 31-72

“Restriction Against @ Character” on page 31-73

“Add to Source Control Is the Only Action Available” on page 31-73

“More Solutions for Source Control Problems” on page 31-74

Source Control Error: Provider Not Present or Not Installed Properly

In some cases, MATLAB software recognizes your source control system but you cannot use source control features for MATLAB. Specifically, when you select **MATLAB > General > Source Control** in the Preferences dialog box, MATLAB lists your source control system, but you cannot perform any source control actions. Only the **Start Source Control System** item is available, and when you select it, MATLAB displays this error:

Source control provider is not present or not installed properly.

Often, this error occurs because a registry key that MATLAB requires from the source control application is not present. Make sure this registry key is present:

```
HKEY_LOCAL_MACHINE\SOFTWARE\SourceCodeControlProvider\  
InstalledSCCProviders
```

The registry key refers to another registry key that is similar to

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\SourceSafe\ScServerPath
```

This registry key has a path to a DLL-file in the file system. Make sure the DLL-file exists in that location. If you are not familiar with registry keys, ask your system administrator for help.

If this does not solve the problem and you use Microsoft Visual SourceSafe, try running a client setup for your source control application. When SourceSafe is installed on a server for a group to use, each machine client can run a setup but is not required to do so. However, some applications that interface with SourceSafe, including MATLAB, require you to run the client setup. Run the client setup, which should resolve the problem.

If the problem persists, access source control outside of MATLAB.

Restriction Against @ Character

Some source control systems, such as Perforce and Synergy™, reserve the @ character. Perforce, for example, uses it as a revision specifier. Therefore, you might experience problems if you use these source control systems with MATLAB files and folders that include the @ character in the folder or file name.

You might be able to work around this restriction by quoting nonstandard characters in file names, such as with an escape sequence, which some source control systems allow. Consult your source control system documentation or technical support resources for a workaround.

Add to Source Control Is the Only Action Available

To use source control features for a file in the Simulink or Stateflow products, the file's source control project must first be registered with MATLAB. When a file's source control project is *not* registered with MATLAB, all **MATLAB > General > Source Control** menu items on the Preferences dialog box are disabled except **Add to Source Control**. You can select **Add to Source Control**, which registers the project with MATLAB, or you can register the project using the Current Folder browser, as described in “Register Source Control Project with MATLAB Software” on page 31-57. You can then perform source control actions for all files in that project (folder).

More Solutions for Source Control Problems

The latest solutions for problems interfacing MATLAB with a source control system appear on the MathWorks Web page for support at <https://www.mathworks.com/support/>. Search Solutions and Technical Notes for “source control.”

Unit Testing

- “Write Test Using Live Script” on page 32-3
- “Write Script-Based Unit Tests” on page 32-7
- “Write Script-Based Test Using Local Functions” on page 32-14
- “Extending Script-Based Tests” on page 32-18
- “Run Tests in Editor” on page 32-22
- “Write Function-Based Unit Tests” on page 32-26
- “Write Simple Test Case Using Functions” on page 32-31
- “Write Test Using Setup and Teardown Functions” on page 32-36
- “Extending Function-Based Tests” on page 32-43
- “Author Class-Based Unit Tests in MATLAB” on page 32-48
- “Write Simple Test Case Using Classes” on page 32-52
- “Write Setup and Teardown Code Using Classes” on page 32-57
- “Types of Qualifications” on page 32-61
- “Tag Unit Tests” on page 32-64
- “Write Tests Using Shared Fixtures” on page 32-69
- “Create Basic Custom Fixture” on page 32-73
- “Create Advanced Custom Fixture” on page 32-76
- “Create Basic Parameterized Test” on page 32-83
- “Create Advanced Parameterized Test” on page 32-89
- “Use External Parameters in Parameterized Test” on page 32-98
- “Create Simple Test Suites” on page 32-103
- “Run Tests for Various Workflows” on page 32-106
- “Programmatically Access Test Diagnostics” on page 32-110
- “Add Plugin to Test Runner” on page 32-111
- “Write Plugins to Extend TestRunner” on page 32-114
- “Create Custom Plugin” on page 32-118

- “Write Plugin to Save Diagnostic Details” on page 32-124
- “Plugin to Generate Custom Test Output Format” on page 32-129
- “Analyze Test Case Results” on page 32-133
- “Analyze Failed Test Results” on page 32-136
- “Rerun Failed Tests” on page 32-139
- “Dynamically Filtered Tests” on page 32-143
- “Create Custom Constraint” on page 32-151
- “Create Custom Boolean Constraint” on page 32-154
- “Create Custom Tolerance” on page 32-157
- “Overview of App Testing Framework” on page 32-163
- “Write Test for App” on page 32-168
- “Write Test That Uses App Testing and Mocking Frameworks” on page 32-173
- “Overview of Performance Testing Framework” on page 32-179
- “Test Performance Using Scripts or Functions” on page 32-184
- “Test Performance Using Classes” on page 32-190
- “Measure Fast Executing Test Code” on page 32-198
- “Create Mock Object” on page 32-202
- “Specify Mock Object Behavior” on page 32-211
- “Qualify Mock Object Interaction” on page 32-218

Write Test Using Live Script

This example shows how to test a function that you create by writing a live script, 'TestRightTriLiveScriptExample.mlx'. The example function computes the angles of a right triangle, and you create a live-script-based unit test to test the function.

A live-script-based test must adhere to the following conventions:

- The name of the live-script file must start or end with the word 'test', which is case-insensitive.*
- Place each unit test into a separate section of the live-script file. The heading of each section becomes the name of the test element. If a section has no heading, MATLAB assigns a name to the test.*
- Consider how you are running your live-script-based test. If you run the test using the **Run** buttons in the Live Editor and MATLAB encounters a test failure, then it stops execution of the script and does not run any remaining tests. If you run the live script using the unit testing framework, such as with the `runtests` function, then if MATLAB encounters a test failure, it still runs remaining tests.*
- When a live script runs as a test, variables defined in one test are not accessible within other tests. Similarly, variables defined in other workspaces are not accessible to the tests.*

Outside of this example, in your current MATLAB folder, create a function in a file, `rightTri.m`. This function takes lengths of two sides of a triangle as input and returns the three angles of the corresponding right triangle. The input sides are the two shorter edges of the triangle, not the hypotenuse.

```
type rightTri.m

function angles = rightTri(sides)

A = atand(sides(1)/sides(2));
B = atand(sides(2)/sides(1));
hypotenuse = sides(1)/sind(A);
C = asind(hypotenuse*sind(A)/sides(1));

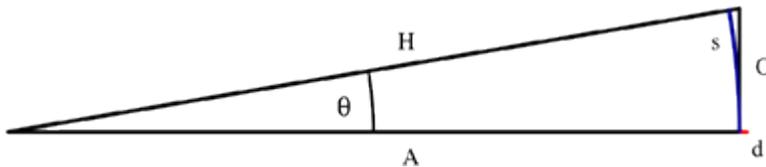
angles = [A B C];

end
```

Test: Small angle approximation

You can include equations and images in your live script to help document the test. Create the following test for the small angle approximation. Typically, when you compare floating-point values, you specify a tolerance for the comparison.

The `rightTri` function should return values consistent with the small angle approximation, such that $\sin(\theta) \approx \theta$.



```
angles = rightTri([1 1500]);
smallAngleInRadians = (pi/180)*angles(1); % convert to radians
approx = sin(smallAngleInRadians);
assert(abs(approx-smallAngleInRadians) <= 1e-10, 'Problem with small angle approximation')
```

Test: Sum of Angles $\sum_k a_k = 180^\circ$

You can have multiple `assert` statements in the same test. However, if the first assertion fails, the `MATLAB` does not evaluate the remaining statements.

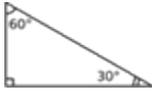
The sum of all angles of the resulting right triangle should always be 180 degrees.

```
angles = rightTri([7 9]);
assert(sum(angles) == 180)

angles = rightTri([4 4]);
assert(sum(angles) == 180)

angles = rightTri([2 2*sqrt(3)]);
assert(sum(angles) == 180)
```

Test: 30-60-90 triangle



Test that the sides of the triangle reduce to 1 and $\sqrt{3}$. In which case, the angles are 30°, 60°, and 90°.

```
tol = 1e-10;
angles = rightTri([2 2*sqrt(3)]);
assert(abs(angles(1)-30)<= tol)
assert(abs(angles(2)-60)<= tol)
assert(abs(angles(3)-90)<= tol)
```

Test: Isosceles Triangles

For isosceles triangles, both of the non-right angles must be 45 degrees; otherwise assert throws an error.

Test that two sides of the triangle are equal. In which case, the corresponding angles are equal.

```
angles = rightTri([4 4]);
assert(angles(1) == 45)
assert(angles(1) == angles(2))
```

*To run your tests, best practice is to use the testing framework via the `runtests` function instead of the **Run** button in the Live Editor. The testing framework provides additional diagnostic information. In the event of a test failure, the framework runs subsequent tests but the **Run** button in the Live Editor does not. For example, to run this test at the MATLAB command prompt, type `result = runtests('TestRightTriLiveScriptExample')`. For more information, see `runtests`.*

See Also

`assert` | `runtests`

Related Examples

- “Write Script-Based Unit Tests” on page 32-7

- “Write Function-Based Unit Tests” on page 32-26

Write Script-Based Unit Tests

This example shows how to write a script that tests a function that you create. The example function computes the angles of a right triangle, and you create a script-based unit test to test the function.

Create rightTri Function to Test

Create this function in a file, `rightTri.m`, in your current MATLAB® folder. This function takes lengths of two sides of a triangle as input and returns the three angles of the corresponding right triangle. The input sides are the two shorter edges of the triangle, not the hypotenuse.

```
function angles = rightTri(sides)

A = atand(sides(1)/sides(2));
B = atand(sides(2)/sides(1));
hypotenuse = sides(1)/sind(A);
C = asind(hypotenuse*sind(A)/sides(1));

angles = [A B C];

end
```

Create Test Script

In your working folder, create a new script, `rightTriTest.m`. Each unit test checks a different output of the `rightTri` function. A test script must adhere to the following conventions:

- The name of the script file must start or end with the word 'test', which is case-insensitive.
- Place each unit test into a separate section of the script file. Each section begins with two percent signs (%%), and the text that follows on the same line becomes the name of the test element. If no text follows the %%, MATLAB assigns a name to the test. If MATLAB encounters a test failure, it still runs remaining tests.
- In a test script, the shared variable section consists of any code that appears before the first explicit code section (the first line beginning with %%). Tests share the variables that you define in this section. Within a test, you can modify the values of

these variables. However, in subsequent tests, the value is reset to the value defined in the shared variables section.

- In the shared variables section (first code section), define any preconditions necessary for your tests. If the inputs or outputs do not meet this precondition, MATLAB does not run any of the tests. MATLAB marks the tests as failed and incomplete.
- When a script is run as a test, variables defined in one test are not accessible within other tests unless they are defined in the shared variables section (first code section). Similarly, variables defined in other workspaces are not accessible to the tests.
- If the script file does not include any code sections, MATLAB generates a single test element from the full contents of the script file. The name of the test element is the same as the script file name. In this case, if MATLAB encounters a failed test, it halts execution of the entire script.

In `rightTriTest.m`, write four tests to test the output of `rightTri`. Use the `assert` function to test the different conditions. In the shared variables section, define four triangle geometries and define a precondition that the `rightTri` function returns a right triangle.

```
% test triangles
tri = [7 9];
triIso = [4 4];
tri306090 = [2 2*sqrt(3)];
triSkewed = [1 1500];

% preconditions
angles = rightTri(tri);
assert(angles(3) == 90, 'Fundamental problem: rightTri not producing right triangle')

%% Test 1: sum of angles
angles = rightTri(tri);
assert(sum(angles) == 180)

angles = rightTri(triIso);
assert(sum(angles) == 180)

angles = rightTri(tri306090);
assert(sum(angles) == 180)

angles = rightTri(triSkewed);
assert(sum(angles) == 180)
```

```

%% Test 2: isosceles triangles
angles = rightTri(triIso);
assert(angles(1) == 45)
assert(angles(1) == angles(2))

%% Test 3: 30-60-90 triangle
angles = rightTri(tri306090);
assert(angles(1) == 30)
assert(angles(2) == 60)
assert(angles(3) == 90)

%% Test 4: Small angle approximation
angles = rightTri(triSkewed);
smallAngle = (pi/180)*angles(1); % radians
approx = sin(smallAngle);
assert(approx == smallAngle, 'Problem with small angle approximation')

```

Test 1 tests the summation of the triangle angles. If the summation is not equal to 180 degrees, `assert` throws an error.

Test 2 tests that if two sides are equal, the corresponding angles are equal. If the non-right angles are not both equal to 45 degrees, the `assert` function throws an error.

Test 3 tests that if the triangle sides are 1 and $\sqrt{3}$, the angles are 30, 60, and 90 degrees. If this condition is not true, `assert` throws an error.

Test 4 tests the small-angle approximation. The small-angle approximation states that for small angles the sine of the angle in radians is approximately equal to the angle. If it is not true, `assert` throws an error.

Run Tests

Execute the `runtests` function to run the four tests in `rightTriTest.m`. The `runtests` function executes each test in each code section individually. If Test 1 fails, MATLAB still runs the remaining tests. If you execute `rightTriTest` as a script instead of by using `runtests`, MATLAB halts execution of the entire script if it encounters a failed assertion. Additionally, when you run tests using the `runtests` function, MATLAB provides informative test diagnostics.

```
result = runtests('rightTriTest');
```

```
Running rightTriTest
```

```
..
```

```

=====
Error occurred in rightTriTest/Test3_30_60_90Triangle and it did not run to completion

-----
Error ID:
-----
'MATLAB:assertion:failed'

-----
Error Details:
-----
Error using rightTriTest (line 31)
Assertion failed.
=====
.
=====
Error occurred in rightTriTest/Test4_SmallAngleApproximation and it did not run to completion

-----
Error ID:
-----
''

-----
Error Details:
-----
Error using rightTriTest (line 39)
Problem with small angle approximation
=====
.
Done rightTriTest
-----

Failure Summary:

Name                                     Failed  Incomplete  Reason(s)
=====
rightTriTest/Test3_30_60_90Triangle      X       X           Errored.
-----
rightTriTest/Test4_SmallAngleApproximation X       X           Errored.
=====

```

The test for the 30-60-90 triangle and the test for the small-angle approximation fail in the comparison of floating-point numbers. Typically, when you compare floating-point values, you specify a tolerance for the comparison. In Test 3 and Test 4, MATLAB throws

an error at the failed assertion and does not complete the test. Therefore, the test is marked as both Failed and Incomplete.

To provide diagnostic information (`Error Details`) that is more informative than 'Assertion failed' (Test 3), consider passing a message to the `assert` function (as in Test 4). Or you can also consider using function-based unit tests.

Revise Test to Use Tolerance

Save `rightTriTest.m` as `rightTriTolTest.m`, and revise Test 3 and Test 4 to use a tolerance. In Test 3 and Test 4, instead of asserting that the angles are equal to an expected value, assert that the difference between the actual and expected values is less than or equal to a specified tolerance. Define the tolerance in the shared variables section of the test script so it is accessible to both tests.

For script-based unit tests, manually verify that the difference between two values is less than a specified tolerance. If instead you write a function-based unit test, you can access built-in constraints to specify a tolerance when comparing floating-point values.

```
% test triangles
tri = [7 9];
triIso = [4 4];
tri306090 = [2 2*sqrt(3)];
triSkewed = [1 1500];

% Define an absolute tolerance
tol = 1e-10;

% preconditions
angles = rightTri(tri);
assert(angles(3) == 90, 'Fundamental problem: rightTri not producing right triangle')

%% Test 1: sum of angles
angles = rightTri(tri);
assert(sum(angles) == 180)

angles = rightTri(triIso);
assert(sum(angles) == 180)

angles = rightTri(tri306090);
assert(sum(angles) == 180)

angles = rightTri(triSkewed);
```

```
assert(sum(angles) == 180)

%% Test 2: isosceles triangles
angles = rightTri(triIso);
assert(angles(1) == 45)
assert(angles(1) == angles(2))

%% Test 3: 30-60-90 triangle
angles = rightTri(tri306090);
assert(abs(angles(1)-30) <= tol)
assert(abs(angles(2)-60) <= tol)
assert(abs(angles(3)-90) <= tol)

%% Test 4: Small angle approximation
angles = rightTri(triSkewed);
smallAngle = (pi/180)*angles(1); % radians
approx = sin(smallAngle);
assert(abs(approx-smallAngle) <= tol, 'Problem with small angle approximation')
```

Rerun the tests.

```
result = runtests('rightTriTolTest');
```

```
Running rightTriTolTest
```

```
....
```

```
Done rightTriTolTest
```

All the tests pass.

Create a table of test results.

```
rt = table(result)
```

```
rt =
```

```
4x6 table
```

Name	Passed	Failed	Incomplete
'rightTriTolTest/Test1_SumOfAngles'	true	false	false

'rightTriTolTest/Test2_IsoscelesTriangles'	true	false	false
'rightTriTolTest/Test3_30_60_90Triangle'	true	false	false
'rightTriTolTest/Test4_SmallAngleApproximation'	true	false	false

See Also

assert | runtests

Related Examples

- “Write Script-Based Test Using Local Functions” on page 32-14
- “Write Function-Based Unit Tests” on page 32-26

Write Script-Based Test Using Local Functions

This example shows how to write a script-based test that uses local functions as helper functions. The example function approximates the sine and cosine of an angle. The script-based test checks the approximation using local functions to check for equality within a tolerance.

Create approxSinCos Function to Test

Create this function in a file, `approxSinCos.m`, in your current MATLAB folder. This function takes an angle in radians and approximates the sine and cosine of the angle using Taylor series.

```
function [sinA,cosA] = approxSinCos(x)
% For a given angle in radians, approximate the sine and cosine of the angle
% using Taylor series.
sinA = x;
cosA = 1;
altSign = -1;
for n = 3:2:26
sinA = sinA + altSign*(x^n)/factorial(n);
cosA = cosA + altSign*(x^(n-1))/factorial(n-1);
altSign = -altSign;
end
```

Create Test Script

In your current MATLAB folder, create a new script, `approxSinCosTest.m`.

Note: Including functions in scripts requires MATLAB® R2016b or later.

```
%% Test 0rad
% Test expected values of 0
[sinApprox,cosApprox] = approxSinCos(0);
assertWithAbsTol(sinApprox,0)
assertWithRelTol(cosApprox,1)

%% Test 2pi
% Test expected values of 2pi
[sinApprox,cosApprox] = approxSinCos(2*pi);
assertWithAbsTol(sinApprox,0)
```

```

assertWithRelTol(cosApprox,1)

%% Test pi over 4 equality
% Test sine and cosine of pi/4 are equal
[sinApprox,cosApprox] = approxSinCos(pi/4);
assertWithRelTol(sinApprox,cosApprox,'sine and cosine should be equal')

%% Test matches MATLAB fcn
% Test values of 2pi/3 match MATLAB output for the sin and cos functions
x = 2*pi/3;
[sinApprox,cosApprox] = approxSinCos(x);
assertWithRelTol(sinApprox,sin(x),'sin does not match')
assertWithRelTol(cosApprox,cos(x),'cos does not match')

function assertWithAbsTol(actVal,expVal,varargin)
% Helper function to assert equality within an absolute tolerance.
% Takes two values and an optional message and compares
% them within an absolute tolerance of 1e-6.
tol = 1e-6;
tf = abs(actVal-expVal) <= tol;
assert(tf, varargin{:});
end

function assertWithRelTol(actVal,expVal,varargin)
% Helper function to assert equality within a relative tolerance.
% Takes two values and an optional message and compares
% them within a relative tolerance of 0.1%.
relTol = 0.001;
tf = abs(expVal - actVal) <= relTol.*abs(expVal);
assert(tf, varargin{:});
end

```

Each unit test uses `assert` to check different output of the `approxSinCos` function. Typically, when you compare floating-point values, you specify a tolerance for the comparison. The local functions `assertWithAbsTol` and `assertWithRelTol` are helper functions to compute whether the actual and expected values are equal within the specified absolute or relative tolerance.

- Test `0rad` tests whether the computed and expected values for an angle of 0 radians are within an absolute tolerance of `1e-6` or a relative tolerance 0.1%. Typically, you use absolute tolerance to compare values close to 0.

- Test `2pi` tests whether the computed and expected values for an angle of 2π radians are equal within an absolute tolerance of $1e-6$ or a relative tolerance 0.1%.
- Test `pi over 4 equality` tests whether the sine and cosine of $\pi/4$ are equal within a relative tolerance of 0.1%.
- Test `matches MATLAB fcn` tests whether the computed sine and cosine of $2\pi/3$ are equal to the values from the `sin` and `cos` functions within a relative tolerance of 0.1%.

Run Tests

Execute the `runtests` function to run the four tests in `approxSinCosTest.m`. The `runtests` function executes each test individually. If one test fails, MATLAB still runs the remaining tests. If you execute `approxSinCosTest` as a script instead of using `runtests`, MATLAB halts execution of the entire script if it encounters a failed assertion. Additionally, when you run tests using the `runtests` function, MATLAB provides informative test diagnostics.

```
results = runtests('approxSinCosTest');
```

```
Running approxSinCosTest
....
Done approxSinCosTest
```

All the tests pass.

Create a table of test results.

```
rt = table(results)
```

```
rt =
```

```
4x6 table
```

Name	Passed	Failed	Incomplete	Duration
'approxSinCosTest/Test0rad'	true	false	false	0.43
'approxSinCosTest/Test2pi'	true	false	false	0.013
'approxSinCosTest/TestPiOver4Equality'	true	false	false	0.0095

'approxSinCosTest/TestMatchesMATLABFcn' true false false 0.01

See Also

assert | runtests

Related Examples

- “Write Script-Based Unit Tests” on page 32-7

More About

- “Add Functions to Scripts” on page 18-18

Extending Script-Based Tests

In this section...

“Test Suite Creation” on page 32-18

“Test Selection” on page 32-19

“Programmatic Access of Test Diagnostics” on page 32-20

“Test Runner Customization” on page 32-20

Typically, with script-based tests, you create a test file, and pass the file name to the `runtests` function without explicitly creating a suite of `Test` objects. If you create an explicit test suite, there are additional features available in script-based testing. These features include selecting tests and using plugins to customize the test runner. For additional functionality, consider using “Function-Based Unit Tests” or “Class-Based Unit Tests”.

Test Suite Creation

To create a test suite from a script-based test directly, use the `testsuite` function. For a more explicit test suite creation, use the `fromFile` method of `TestSuite`. Then you can use the `run` method instead of the `runtests` function to run the tests. For example, if you have a script-based test in a file `rightTriTolTest.m`, these three approaches are equivalent.

```
% Implicit test suite
result = runtests('rightTriTolTest.m');

% Explicit test suite
suite = testsuite('rightTriTolTest.m');
result = run(suite);

% Explicit test suite
suite = matlab.unittest.TestSuite.fromFile('rightTriTolTest.m');
result = run(suite);
```

Also, you can create a test suite from all the test files in a specified folder using the `TestSuite.fromFolder` method. If you know the name of a particular test in your script-based test file, you can create a test suite from that test using `TestSuite.fromName`.

Test Selection

With an explicit test suite, use selectors to refine your suite. Several of the selectors are applicable only for class-based tests, but you can select tests for your suite based on the test name:

- Use the 'Name' name-value pair argument in a suite generation method, such as `fromFile`.
- Use a `selectors` instance and optional `constraints` instance.

Use these approaches in a suite generation method, such as `fromFile`, or create a suite and filter it using the `TestSuite.selectIf` method. For example, in this listing, the four values of `suite` are equivalent.

```
import matlab.unittest.selectors.HasName
import matlab.unittest.constraints.ContainsSubstring
import matlab.unittest.TestSuite.fromFile

f = 'rightTriTolTest.m';
selector = HasName(ContainsSubstring('Triangle'));

% fromFile, name-value pair
suite = TestSuite.fromFile(f, 'Name', '*Triangle*')

% fromFile, selector
suite = TestSuite.fromFile(f, selector)

% selectIf, name-value pair
fullSuite = TestSuite.fromFile(f);
suite = selectIf(fullSuite, 'Name', '*Triangle*')

% selectIf, selector
fullSuite = TestSuite.fromFile(f);
suite = selectIf(fullSuite, selector)
```

If you use one of the suite creation methods with a selector or name-value pair, the testing framework creates the filtered suite. If you use the `TestSuite.selectIf` method, the testing framework creates a full test suite and then filters it. For large test suites, this approach can have performance implications.

Programmatic Access of Test Diagnostics

If you run tests with the `runtests` function or the `run` method of `TestSuite` or `TestCase`, the test framework uses a `DiagnosticsRecordingPlugin` plugin that records diagnostics on test results.

After you run tests, you can access recorded diagnostics via the `DiagnosticRecord` field in the `Details` property on `TestResult`. For example, if your test results are stored in the variable `results`, find the recorded diagnostics for the second test in the suite by invoking `records = result(2).Details.DiagnosticRecord`.

The recorded diagnostics are `DiagnosticRecord` objects. To access particular types of test diagnostics for a particular test, use the `selectFailed`, `selectPassed`, `selectIncomplete`, and `selectLogged` methods of the `DiagnosticRecord` class.

By default, the `DiagnosticsRecordingPlugin` plugin records qualification failures and logged events at the `matlab.unittest.Verbosity.Terse` level of verbosity. For more information, see `DiagnosticsRecordingPlugin` and `DiagnosticRecord`.

Test Runner Customization

Use a `TestRunner` object to customize the way the framework runs a test suite. With a `TestRunner` object you can:

- Produce no output in the command window using the `withNoPlugins` method.
- Run tests in parallel using the `runInParallel` method.
- Add plugins to the test runner using the `addPlugin` method.

For example, use test suite, `suite`, to create a silent test runner and run the tests with the `run` method of `TestRunner`.

```
runner = matlab.unittest.TestRunner.withNoPlugins;  
results = runner.run(suite);
```

Use plugins to customize the test runner further. For example, you can redirect output, determine code coverage, or change how the test runner responds to warnings. For more information, see “Add Plugin to Test Runner” on page 32-111 and the `plugins` classes.

See Also

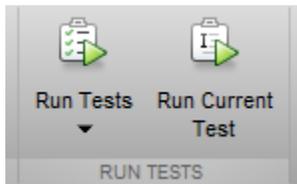
`TestRunner` | `TestSuite` | `matlab.unittest.constraints` | `plugins` | `selectors`

Related Examples

- “Add Plugin to Test Runner” on page 32-111

Run Tests in Editor

This example shows how to run a test while working in the Editor. When you open a function-based test or class-based test in the Editor, the **Editor** tab contains a **Run Tests** section instead of a **Run** section. This section provides several options to run the tests in your current file.



The **Run Tests** button runs all the tests in the file. The **Run Current Test** button runs the test at your current cursor location.

In the Editor, create a test in a file named `sampleTest.m`. When you save the test, the **Run** section in the **Editor** tab changes to **Run Tests**.

```
function tests = sampleTest
    tests = functiontests(localfunctions);
end

function testA(testCase)
    verifyEqual(testCase,5,5)
end

function testB(testCase)
    verifyGreaterThan(testCase,42,13)
end

function testC(testCase)
    verifySubstring(testCase,'hello, world','llo')
end
```

Click the **Run Tests** icon. MATLAB displays the command it uses to run the tests in the Command Window, and the test output follows. MATLAB runs all three tests from `sampleTest.m`.

```
runtests('sampleTest')
Running sampleTest
...
```

```
Done sampleTest
```

```
ans =
```

```
1x3 TestResult array with properties:
```

```
    Name  
    Passed  
    Failed  
    Incomplete  
    Duration  
    Details
```

```
Totals:
```

```
    3 Passed, 0 Failed, 0 Incomplete.  
    0.0071673 seconds testing time.
```

In the Editor, place your cursor in the `testB` function and click the **Run Current Test** icon. MATLAB runs `testB` only.

```
runtests('sampleTest','ProcedureName','testB')
```

```
Running sampleTest
```

```
·  
Done sampleTest
```

```
ans =
```

```
TestResult with properties:
```

```
    Name: 'sampleTest/testB'  
    Passed: 1  
    Failed: 0  
    Incomplete: 0  
    Duration: 9.9411e-04  
    Details: [1x1 struct]
```

```
Totals:
```

```
    1 Passed, 0 Failed, 0 Incomplete.  
    0.00099411 seconds testing time.
```

In addition to running tests, you can customize the test run using the test options under the **Run Tests** icon. MATLAB uses test options whether you run all the tests in a file or just the test at your cursor location.

Test Option	Description
Clear Command Window	Clears the Command Window before running tests.
Strict	<p>Applies strict checks while running tests. For example, the framework generates a qualification failure if a test issues a warning.</p> <p>Tests that run with this option selected have the 'Strict' option of <code>runtests</code> set to <code>true</code>.</p>
Parallel	<p>Runs tests in parallel. This option is only available if the Parallel Computing Toolbox is installed.</p> <p>Tests that run with this option selected have the 'UseParallel' option of <code>runtests</code> set to <code>true</code>.</p>
Output Detail	<p>Controls the amount of detail displayed for a test run.</p> <p>For example, tests that run with Output Detail specified as 0: None have the 'OutputDetail' option of <code>runtests</code> set to 0.</p>
Logging Level	<p>Displays diagnostics logged by the <code>TestCase.log</code> method at the specified verbosity level or lower.</p> <p>For example, tests that run with Logging Level specified as 3: Detailed have the 'LogLevel' option of <code>runtests</code> set to 3.</p>

When you select a test option, the selection persists for the duration of your current MATLAB session.

See Also

runtests

Write Function-Based Unit Tests

In this section...

“Create Test Function” on page 32-26

“Run the Tests” on page 32-29

“Analyze the Results” on page 32-29

Create Test Function

Your test function is a single MATLAB file that contains a main function and your individual local test functions. Optionally, you can include file fixture and fresh fixture functions. File fixtures consist of setup and teardown functions shared across all the tests in a file. These functions are executed once per test file. Fresh fixtures consist of setup and teardown functions that are executed before and after each local test function.

Create the Main Function

The main function collects all of the local test functions into a test array. Since it is the main function, the function name corresponds to the name of your `.m` file and follows the naming convention of starting or ending in the word ‘test’, which is case-insensitive. In this sample case, the MATLAB file is `exampleTest.m`. The main function needs to make a call to `functiontests` to generate a test array, `tests`. Use `localfunctions` as the input to `functiontests` to automatically generate a cell array of function handles to all the local functions in your file. This is a typical main function.

```
function tests = exampleTest
tests = functiontests(localfunctions);
end
```

Create Local Test Functions

Individual test functions are included as local functions in the same MATLAB file as the main (test-generating) function. These test function names must begin or end with the case-insensitive word, ‘test’. Each of the local test functions must accept a single input, which is a function test case object, `testCase`. The Unit Test Framework automatically generates this object. For more information on creating test functions, see “Write Simple Test Case Using Functions” on page 32-31 and “Types of Qualifications” on page 32-61. This is a typical example of skeletal local-test functions.

```
function testFunctionOne(testCase)
% Test specific code
end

function FunctionTwotest(testCase)
% Test specific code
end
```

Create Optional Fixture Functions

Setup and teardown code, also referred to as test fixture functions, set up the pretest state of the system and return it to the original state after running the test. There are two types of these functions: file fixture functions that run once per test file, and fresh fixture functions that run before and after each local test function. These functions are not required to generate tests. In general, it is preferable to use fresh fixtures over file fixtures to increase unit test encapsulation.

A function test case object, `testCase`, must be the only input to file fixture and fresh fixture functions. The Unit Test Framework automatically generates this object. The `TestCase` object is a means to pass information between setup functions, test functions, and teardown functions. Its `TestData` property is, by default, a `struct`, which allows easy addition of fields and data. Typical uses for this test data include paths and graphics handles. For an example using the `TestData` property, see “Write Test Using Setup and Teardown Functions” on page 32-36.

File Fixture Functions

Use file fixture functions to share setup and teardown functions across all the tests in a file. The names for the file fixture functions must be `setupOnce` and `teardownOnce`, respectively. These functions execute a single time for each file. You can use file fixtures to set a path before testing, and then reset it to the original path after testing. This is a typical example of skeletal file fixture setup and teardown code.

```
function setupOnce(testCase) % do not change function name
% set a new path, for example
end

function teardownOnce(testCase) % do not change function name
% change back to original path, for example
end
```

Fresh Fixture Functions

Use fresh fixture functions to set up and tear down states for each local test function. The names for these fresh fixture functions must be `setup` and `teardown`, respectively. You can use fresh fixtures to obtain a new figure before testing and to close the figure after testing. This is typical example of skeletal test function level setup and teardown code.

```
function setup(testCase) % do not change function name
% open a figure, for example
end

function teardown(testCase) % do not change function name
% close figure, for example
end
```

Program Listing Template

```
%% Main function to generate tests
function tests = exampleTest
tests = functiontests(localfunctions);
end

%% Test Functions
function testFunctionOne(testCase)
% Test specific code
end

function FunctionTwotest(testCase)
% Test specific code
end

%% Optional file fixtures
function setupOnce(testCase) % do not change function name
% set a new path, for example
end

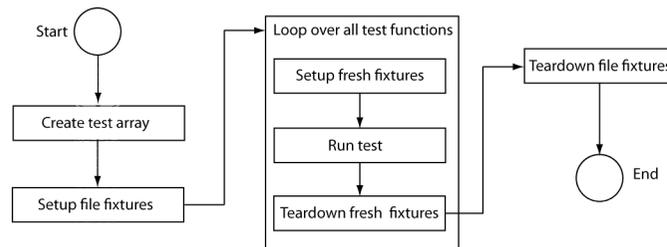
function teardownOnce(testCase) % do not change function name
% change back to original path, for example
end

%% Optional fresh fixtures
function setup(testCase) % do not change function name
% open a figure, for example
end
```

```
function teardown(testCase) % do not change function name
% close figure, for example
end
```

Run the Tests

The figure below details the tasks executed when you run the tests.



To run tests from the command prompt, use the `runtests` command with your MATLAB test file as input. For example:

```
results = runtests('exampleTest.m')
```

Alternatively, you can run tests using the `run` function.

```
results = run(exampleTest)
```

For more information on running tests see the `runtests` reference page and “Run Tests for Various Workflows” on page 32-106.

Analyze the Results

To analyze the test results, examine the output structure from `runtests` or `run`. For each test, the result contains the name of the test function, whether it passed, failed, or didn't complete, and the time it took to run the test. For more information, see “Analyze Test Case Results” on page 32-133 and “Analyze Failed Test Results” on page 32-136.

See Also

`functiontests` | `localfunctions` | `runtests`

Related Examples

- “Write Simple Test Case Using Functions” on page 32-31
- “Write Test Using Setup and Teardown Functions” on page 32-36

Write Simple Test Case Using Functions

This example shows how to write a unit test for a MATLAB function, `quadraticSolver.m`.

Create `quadraticSolver.m` Function

This MATLAB function solves quadratic equations. Create this function in a folder on your MATLAB path.

```
function roots = quadraticSolver(a, b, c)
% quadraticSolver returns solutions to the
% quadratic equation a*x^2 + b*x + c = 0.

if ~isa(a,'numeric') || ~isa(b,'numeric') || ~isa(c,'numeric')
    error('quadraticSolver:InputMustBeNumeric', ...
        'Coefficients must be numeric.');
```

$$\text{roots}(1) = \frac{-b + \sqrt{b^2 - 4*a*c}}{2*a};$$
$$\text{roots}(2) = \frac{-b - \sqrt{b^2 - 4*a*c}}{2*a};$$

```
end
```

Create `solverTest` Function

Create this function in a folder on your MATLAB path.

```
function tests = solverTest
tests = functiontests(localfunctions);
end
```

A call to `functiontests` using `localfunctions` as input creates an array of tests from each local function in the `solverTest.m` file. Each test is a local function that follows the naming convention of having 'test' at the beginning or end of the function name. Local functions that do not follow this convention are not included in the test array. Test functions must accept a single input argument into which the test framework passes a function test case object. The function uses this object for verifications, assertions, assumptions, and fatal assertions. It contains a `TestData` structure that allows data to be passed between setup, test, and teardown functions.

Create Test Function for Real Solutions

Create a test function, `testRealSolution`, to verify that `quadraticSolver` returns the correct value for real solutions. For example, the equation $x^2 - 3x + 2 = 0$ has real solutions $x = 1$ and $x = 2$. This function calls `quadraticSolver` with the inputs of this equation. The expected solution, `expSolution`, is `[2,1]`.

Use the qualification function, `verifyEqual`, to compare the output of the function, `actSolution`, to the desired output, `expSolution`. If the qualification fails, the framework continues executing the test. Typically, when using `verifyEqual` on floating point values, you specify a tolerance for the comparison. For more information, see `matlab.unittest.constraints`.

Add this function to the `solverTest.m` file.

```
function testRealSolution(testCase)
actSolution = quadraticSolver(1,-3,2);
expSolution = [2 1];
verifyEqual(testCase,actSolution,expSolution)
end
```

Create Test Function for Imaginary Solutions

Create a test to verify that `quadraticSolver` returns the right value for imaginary solutions. For example, the equation $x^2 + 2x + 10 = 0$ has imaginary solutions $x = -1 + 3i$ and $x = -1 - 3i$. Typically, when using `verifyEqual` on floating point values, you specify a tolerance for the comparison. For more information, see `matlab.unittest.constraints`.

Add this function, `testImaginarySolution`, to the `solverTest.m` file.

```
function testImaginarySolution(testCase)
actSolution = quadraticSolver(1,2,10);
expSolution = [-1+3i -1-3i];
verifyEqual(testCase,actSolution,expSolution)
end
```

The order of the tests within the `solverTest.m` file does not matter because they are fully independent test cases.

Save solverTest Function

The following is the complete `solverTest.m` test file. Save this file in a folder on your MATLAB path.

```

function tests = solverTest
tests = functiontests(localfunctions);
end

function testRealSolution(testCase)
actSolution = quadraticSolver(1,-3,2);
expSolution = [2 1];
verifyEqual(testCase,actSolution,expSolution)
end

function testImaginarySolution(testCase)
actSolution = quadraticSolver(1,2,10);
expSolution = [-1+3i -1-3i];
verifyEqual(testCase,actSolution,expSolution)
end

```

Run Tests in solverTest Function

Run the tests.

```
results = runtests('solverTest.m')
```

```
Running solverTest
```

```
..
Done solverTest
```

```
results =
```

```
1x2 TestResult array with properties:
```

```

Name
Passed
Failed
Incomplete
Duration

```

```
Totals:
```

```

2 Passed, 0 Failed, 0 Incomplete.
0.19172 seconds testing time.

```

Both of the tests passed.

Introduce an Error in quadraticSolver.m and Run Tests

Cause one of the tests to fail by forcing roots in quadraticSolver.m to be real. Before ending the function, add the line: `roots = real(roots);`. (Do not change solverTest.m.) Save the file and run the tests.

```
results = runtests('solverTest.m')
```

```
Running solverTest
```

```
.
=====
Verification failed in solverTest/testImaginarySolution.

-----
Framework Diagnostic:
-----
verifyEqual failed.
--> Complexity does not match.

    Actual Complexity:
        Real
    Expected Complexity:
        Complex

Actual Value:
    -1    -1
Expected Value:
    -1.0000000000000000 + 3.0000000000000000i -1.0000000000000000 - 3.0000000000000000i

-----
Stack Information:
-----
In C:\work\solverTest.m (testImaginarySolution) at 14
=====
```

```
.
Done solverTest
```

```
-----
Failure Summary:
```

Name	Failed	Incomplete	Reason(s)
solverTest/testImaginarySolution	X		Failed by verification.

```
results =
```

```
1x2 TestResult array with properties:
```

```
Name
Passed
Failed
Incomplete
Duration
```

```
Totals:  
 1 Passed, 1 Failed, 0 Incomplete.  
 0.043751 seconds testing time.
```

The imaginary test verification failed.

Restore `quadraticSolver.m` to its previous, correct version by removing the `roots = real(roots);` code.

See Also

`matlab.unittest.constraints`

More About

- “Write Function-Based Unit Tests” on page 32-26
- “Types of Qualifications” on page 32-61

Write Test Using Setup and Teardown Functions

This example shows how to write a unit test for a couple of MATLAB® figure axes properties using fresh fixtures and file fixtures.

Create axesPropertiesTest File

Create a file containing the main function that tests figure axes properties and include two test functions. One function verifies that the x-axis limits are correct, and the other one verifies that the face color of a surface is correct.

In a folder on your MATLAB path, create `axesPropertiesTest.m`. In the main function of this file, have `functiontests` create an array of tests from each local function in `axesPropertiesTest.m` with a call to the `localfunctions` function.

```
% Copyright 2015 The MathWorks, Inc.
```

```
function tests = axesPropertiesTest
tests = functiontests(localfunctions);
end
```

Create File Fixture Functions

File fixture functions are setup and teardown code that runs a single time in your test file. These fixtures are shared across the test file. In this example, the file fixture functions create a temporary folder and set it as the current working folder. They also create and save a new figure for testing. After tests are complete, the framework reinstates the original working folder and deletes the temporary folder and saved figure.

In this example, a helper function creates a simple figure — a red cylinder. In a more realistic scenario, this code is part of the product under test and is computationally expensive, thus motivating the intent to create the figure only once and to load independent copies of the result for each test function. For this example, however, you want to create this helper function as a local function to `axesPropertiesTest`. Note that the test array does not include the function because its name does not start or end with 'test'.

Write a helper function that creates a simple red cylinder and add it as a local function to `axesPropertiesTest`.

```
% Copyright 2015 The MathWorks, Inc.
```

```
function f = createFigure
f = figure;
ax = axes('Parent', f);
cylinder(ax,10)
h = findobj(ax,'Type','surface');
h.FaceColor = [1 0 0];
end
```

You must name the setup and teardown functions of a file test fixture `setupOnce` and `teardownOnce`, respectively. These functions take a single input argument, `testCase`, into which the test framework automatically passes a function test case object. This test case object contains a `TestData` structure that allows data to pass between setup, test, and teardown functions. In this example, the `TestData` structure uses assigned fields to store the original path, the temporary folder name, and the figure file name.

Create the setup and teardown functions as a local functions to `axesPropertiesTest`.

```
% Copyright 2015 The MathWorks, Inc.
```

```
function setupOnce(testCase)
% create and change to temporary folder
testCase.TestData.origPath = pwd;
testCase.TestData.tmpFolder = ['tmpFolder' datestr(now,30)];
mkdir(testCase.TestData.tmpFolder)
cd(testCase.TestData.tmpFolder)

% create and save a figure
testCase.TestData.figName = 'tmpFig.fig';
aFig = createFigure;
saveas(aFig,testCase.TestData.figName,'fig')
close(aFig)
end

function teardownOnce(testCase)
delete(testCase.TestData.figName)
cd(testCase.TestData.origPath)
rmdir(testCase.TestData.tmpFolder)
```

```
end
```

Create Fresh Fixture Functions

Fresh fixtures are function level setup and teardown code that runs before and after each test function in your file. In this example, the functions open the saved figure and find the handles. After testing, the framework closes the figure.

You must name fresh fixture functions `setup` and `teardown`, respectively. Similar to the file fixture functions, these functions take a single input argument, `testCase`. In this example, these functions create a new field in the `TestData` structure that includes handles to the figure and to the axes. This allows information to pass between setup, test, and teardown functions.

Create the setup and teardown functions as a local functions to `axesPropertiesTest`. Open the saved figure for each test to ensure test independence.

```
% Copyright 2015 The MathWorks, Inc.  
  
function setup(testCase)  
testCase.TestData.Figure = openfig(testCase.TestData.figName);  
testCase.TestData.Axes = findobj(testCase.TestData.Figure, ...  
    'Type', 'Axes');  
end  
  
function teardown(testCase)  
close(testCase.TestData.Figure)  
end
```

In addition to custom setup and teardown code, the Unit Testing Framework provides some classes for creating fixtures. For more information see `matlab.unittest.fixtures`.

Create Test Functions

Each test is a local function that follows the naming convention of having 'test' at the beginning or end of the function name. The test array does not include local functions that do not follow this convention. Similar to setup and teardown functions, individual test functions must accept a single input argument, `testCase`. Use this test case object for verifications, assertions, assumptions, and fatal assertions functions.

The `testDefaultXLim` function test verifies that the x-axis limits are large enough to display the cylinder. The lower limit needs to be less than `-10`, and the upper limit needs to be greater than `10`. These values come from the figure generated in the helper function — a cylinder with a `10` unit radius centered on the origin. This test function opens the figure created and saved in the `setupOnce` function, queries the axes limit, and verifies the limits are correct. The qualification functions, `verifyLessThanOrEqual` and `verifyGreaterThanOrEqual`, takes the test case, the actual value, the expected value, and optional diagnostic information to display in the case of failure as inputs.

Create the `testDefaultXLim` function as local function to `axesPropertiesTest`.

```
% Copyright 2015 The MathWorks, Inc.

function testDefaultXLim(testCase)
xlim = testCase.TestData.Axes.XLim;
verifyLessThanOrEqual(testCase, xlim(1), -10,...
    'Minimum x-limit was not small enough')
verifyGreaterThanOrEqual(testCase, xlim(2), 10,...
    'Maximum x-limit was not big enough')
end
```

The `surfaceColorTest` function accesses the figure that you created and saved in the `setupOnce` function. `surfaceColorTest` queries the face color of the cylinder and verifies that it is red. The color red has an RGB value of `[1 0 0]`. The qualification function, `verifyEqual`, takes as inputs the test case, the actual value, the expected value, and optional diagnostic information to display in the case of failure. Typically when using `verifyEqual` on floating point-values, you specify a tolerance for the comparison. For more information, see `matlab.unittest.constraints`.

Create the `surfaceColorTest` function as local function to `axesPropertiesTest`.

```
% Copyright 2015 The MathWorks, Inc.

function surfaceColorTest(testCase)
h = findobj(testCase.TestData.Axes, 'Type', 'surface');
co = h.FaceColor;
verifyEqual(testCase, co, [1 0 0], 'FaceColor is incorrect')
end
```

Now the `axesPropertiesTest.m` file is complete with a main function, file fixture functions, fresh fixture functions, and two local test functions. You are ready to run the tests.

Run Tests

The next step is to run the tests using the `runtests` function. In this example, the call to `runtests` results in the following steps:

- 1 The main function creates a test array.
- 2 The file fixture records the working folder, creates a temporary folder, sets the temporary folder as the working folder, then generates and saves a figure.
- 3 The fresh fixture setup opens the saved figure and finds the handles.
- 4 The `testDefaultXLim` test is run.
- 5 The fresh fixture teardown closes the figure.
- 6 The fresh fixture setup opens the saved figure and finds the handles.
- 7 The `surfaceColorTest` test is run.
- 8 The fresh fixture teardown closes the figure.
- 9 The file fixture teardown deletes the saved figure, changes back to the original path and deletes the temporary folder.

At the command prompt, generate and run the test suite.

```
results = runtests('axesPropertiesTest.m')
```

```
Running axesPropertiesTest
```

```
..
```

```
Done axesPropertiesTest
```

```
_____
```

```
results =
```

```
    1x2 TestResult array with properties:
```

```
    Name  
    Passed  
    Failed  
    Incomplete  
    Duration  
    Details
```

```
Totals:
  2 Passed, 0 Failed, 0 Incomplete.
  2.4886 seconds testing time.
```

Create Table of Test Results

To access functionality available to tables, create one from the `TestResult` object.

```
rt = table(results)
```

```
rt =
```

```
2x6 table
```

Name	Passed	Failed	Incomplete	Duration
'axesPropertiesTest/testDefaultXLim'	true	false	false	1.4265
'axesPropertiesTest/surfaceColorTest'	true	false	false	1.0621

Export test results to an Excel® spreadsheet.

```
writetable(rt, 'myTestResults.xls')
```

Sort the test results by increasing duration.

```
sortrows(rt, 'Duration')
```

```
ans =
```

```
2x6 table
```

Name	Passed	Failed	Incomplete	Duration
'axesPropertiesTest/surfaceColorTest'	true	false	false	1.0621

'axesPropertiesTest/testDefaultXLim' true false false 1.4265

See Also

`matlab.unittest.constraints` | `matlab.unittest.fixtures`

More About

- “Write Function-Based Unit Tests” on page 32-26
- “Types of Qualifications” on page 32-61

Extending Function-Based Tests

In this section...

“Fixtures for Setup and Teardown Code” on page 32-43

“Test Logging and Verbosity” on page 32-44

“Test Suite Creation” on page 32-45

“Test Selection” on page 32-45

“Test Running” on page 32-46

“Programmatic Access of Test Diagnostics” on page 32-46

“Test Runner Customization” on page 32-47

Typically, with function-based tests, you create a test file and pass the file name to the `runtests` function without explicitly creating a suite of `Test` objects. However, if you create an explicit test suite, additional features are available in function-based testing. These features include:

- Test logging and verbosity
- Test selection
- Plugins to customize the test runner

For additional functionality, consider using “Class-Based Unit Tests”.

Fixtures for Setup and Teardown Code

When writing tests, use the `TestCase.applyFixture` method to handle setup and teardown code for actions such as:

- Changing the current working folder
- Adding a folder to the path
- Creating a temporary folder
- Suppressing the display of warnings

These fixtures take the place of manually coding the actions in the `setUpOnce`, `tearDownOnce`, `setUp`, and `tearDown` functions of your function-based test.

For example, if you manually write setup and teardown code to set up a temporary folder for each test, and then you make that folder your current working folder, your setup and teardown functions could look like this.

```
function setup(testCase)
% store current folder
testCase.TestData.origPath = pwd;

% create temporary folder
testCase.TestData.tmpFolder = ['tmpFolder' datestr(now,30)];
mkdir(testCase.TestData.tmpFolder)

% change to temporary folder
cd(testCase.TestData.tmpFolder)
end

function teardown(testCase)
% change to original folder
cd(testCase.TestData.origPath)

% delete temporary folder
rmdir(testCase.TestData.tmpFolder)
end
```

However, you also can use a fixture to replace both of those functions with just a modified setup function. The fixture stores the information necessary to restore the initial state and performs the teardown actions.

```
function setup(testCase)
% create temporary folder
f = testCase.applyFixture(matlab.unittest.fixtures.TemporaryFolderFixture);

% change to temporary folder
testCase.applyFixture(matlab.unittest.fixtures.CurrentFolderFixture(f.Folder));
end
```

Test Logging and Verbosity

Your test functions can use the `TestCase.log` method. By default, the test runner reports diagnostics logged at verbosity level 1 (Terse). Use the `LoggingPlugin.withVerbosity` method to respond to messages of other verbosity levels. Construct a `TestRunner` object, add the `LoggingPlugin`, and run the suite with the `TestRunner.run` method. For more information on creating a test runner, see “Test Runner Customization” on page 32-47.

Test Suite Creation

Calling your function-based test returns a suite of `Test` objects. You also can use the `testsuite` function or the `TestSuite.fromFile` method. If you want a particular test and you know the test name, you can use `TestSuite.fromName`. If you want to create a suite from all tests in a particular folder, you can use `TestSuite.fromFolder`.

Test Selection

With an explicit test suite, use selectors to refine your suite. Several of the selectors are applicable only for class-based tests, but you can select tests for your suite based on the test name:

- Use the 'Name' name-value pair argument in a suite generation method, such as `fromFile`.
- Use a `selectors` instance and optional `constraints` instance.

Use these approaches in a suite generation method, such as `fromFile`, or create a suite and filter it using the `TestSuite.selectIf` method. For example, in this listing, the four values of `suite` are equivalent.

```
import matlab.unittest.selectors.HasName
import matlab.unittest.constraints.ContainsSubstring
import matlab.unittest.TestSuite.fromFile

f = 'rightTriTolTest.m';
selector = HasName(ContainsSubstring('Triangle'));

% fromFile, name-value pair
suite = TestSuite.fromFile(f, 'Name', '*Triangle*')

% fromFile, selector
suite = TestSuite.fromFile(f, selector)

% selectIf, name-value pair
fullSuite = TestSuite.fromFile(f);
suite = selectIf(fullSuite, 'Name', '*Triangle*')

% selectIf, selector
fullSuite = TestSuite.fromFile(f);
suite = selectIf(fullSuite, selector)
```

If you use one of the suite creation methods with a selector or name-value pair, the testing framework creates the filtered suite. If you use the `TestSuite.selectIf` method, the testing framework creates a full test suite and then filters it. For large test suites, this approach can have performance implications.

Test Running

There are several ways to run a function-based test.

To Run All Tests	Use Function
In a file	<code>runtests</code> with the name of the test file
In a suite	<code>TestSuite.run</code> with the suite
In a suite with a custom test runner	<code>TestRunner.run</code> . (See “Test Runner Customization” on page 32-47.)

For more information, see “Run Tests for Various Workflows” on page 32-106.

Programmatic Access of Test Diagnostics

If you run tests with the `runtests` function or the `run` method of `TestSuite` or `TestCase`, the test framework uses a `DiagnosticsRecordingPlugin` plugin that records diagnostics on test results.

After you run tests, you can access recorded diagnostics via the `DiagnosticRecord` field in the `Details` property on `TestResult`. For example, if your test results are stored in the variable `results`, find the recorded diagnostics for the second test in the suite by invoking `records = result(2).Details.DiagnosticRecord`.

The recorded diagnostics are `DiagnosticRecord` objects. To access particular types of test diagnostics for a particular test, use the `selectFailed`, `selectPassed`, `selectIncomplete`, and `selectLogged` methods of the `DiagnosticRecord` class.

By default, the `DiagnosticsRecordingPlugin` plugin records qualification failures and logged events at the `matlab.unittest.Verbosity.Terse` level of verbosity. For more information, see `DiagnosticsRecordingPlugin` and `DiagnosticRecord`.

Test Runner Customization

Use a `TestRunner` object to customize the way the framework runs a test suite. With a `TestRunner` object you can:

- Produce no output in the command window using the `withNoPlugins` method.
- Run tests in parallel using the `runInParallel` method.
- Add plugins to the test runner using the `addPlugin` method.

For example, use test suite, `suite`, to create a silent test runner and run the tests with the `run` method of `TestRunner`.

```
runner = matlab.unittest.TestRunner.withNoPlugins;  
results = runner.run(suite);
```

Use plugins to customize the test runner further. For example, you can redirect output, determine code coverage, or change how the test runner responds to warnings. For more information, see “Add Plugin to Test Runner” on page 32-111 and the `plugins` classes.

See Also

`matlab.unittest.TestCase` | `matlab.unittest.TestSuite` |
`matlab.unittest.constraints` | `matlab.unittest.diagnostics` |
`matlab.unittest.qualifications` | `matlab.unittest.selectors`

Related Examples

- “Run Tests for Various Workflows” on page 32-106
- “Add Plugin to Test Runner” on page 32-111

Author Class-Based Unit Tests in MATLAB

To test a MATLAB program, write a unit test using qualifications that are methods for testing values and responding to failures.

The Test Class Definition

A test class must inherit from `matlab.unittest.TestCase` and contain a methods block with the `Test` attribute. The methods block contains functions, each of which is a unit test. A general, basic class definition follows.

```
%% Test Class Definition
classdef MyComponentTest < matlab.unittest.TestCase

    %% Test Method Block
    methods (Test)
        % includes unit test functions
    end
end
```

The Unit Tests

A unit test is a method that determines the correctness of a unit of software. Each unit test is contained within a methods block. The function must accept a `TestCase` instance as an input.

```
%% Test Class Definition
classdef MyComponentTest < matlab.unittest.TestCase

    %% Test Method Block
    methods (Test)

        %% Test Function
        function testASolution(testCase)
            %% Exercise function under test
            % act = the value from the function under test

            %% Verify using test qualification
            % exp = your expected value
            % testCase.<qualification method>(act,exp);
        end
    end
end
```

```

end
end

```

Qualifications are methods for testing values and responding to failures. This table lists the types of qualifications.

Verifications	Use this qualification to produce and record failures without throwing an exception. The remaining tests run to completion.	<code>matlab.unittest.qualifications.Verifiable</code>
Assumptions	Use this qualification to ensure that a test runs only when certain preconditions are satisfied. However, running the test without satisfying the preconditions does not produce a test failure. When an assumption failure occurs, the testing framework marks the test as filtered.	<code>matlab.unittest.qualifications.Assumable</code>
Assertions	Use this qualification to ensure that the preconditions of the current test are met.	<code>matlab.unittest.qualifications.Assertable</code>
Fatal assertions	Use this qualification when the failure at the assertion point renders the remainder of the current test method invalid or the state is unrecoverable.	<code>matlab.unittest.qualifications.FatalAssertable</code>

The MATLAB Unit Testing Framework provides approximately 25 qualification methods for each type of qualification. For example, use `verifyClass` or `assertClass` to test that a value is of an expected class, and use `assumeTrue` or `fatalAssertTrue` to test if the actual value is true. For a summary of qualification methods, see “Types of Qualifications” on page 32-61.

Often, each unit test function obtains an actual value by exercising the code that you are testing and defines the associated expected value. For example, if you are testing the

`plus` function, the actual value might be `plus(2,3)` and the expected value 5. Within the test function, you pass the actual and expected values to a qualification method. For example:

```
testCase.verifyEqual(plus(2,3),5)
```

For an example of a basic unit test, see “Write Simple Test Case Using Classes” on page 32-52.

Additional Features for Advanced Test Classes

The MATLAB Unit Testing Framework includes several features for authoring more advanced test classes:

- Setup and teardown methods blocks to implicitly set up the pretest state of the system and return it to the original state after running the tests. For an example of a test class with setup and teardown code, see “Write Setup and Teardown Code Using Classes” on page 32-57.
- Advanced qualification features, including actual value proxies, test diagnostics, and a constraint interface. For more information, see `matlab.unittest.constraints` and `matlab.unittest.diagnostics`.
- Parameterized tests to combine and execute tests on the specified lists of parameters. For more information, see “Create Basic Parameterized Test” on page 32-83 and “Create Advanced Parameterized Test” on page 32-89.
- Ready-to-use fixtures for handling the setup and teardown of frequently used testing actions and for sharing fixtures between classes. For more information, see `matlab.unittest.fixtures` and “Write Tests Using Shared Fixtures” on page 32-69.
- Ability to create custom test fixtures. For more information see “Create Basic Custom Fixture” on page 32-73 and “Create Advanced Custom Fixture” on page 32-76.

See Also

Related Examples

- “Write Simple Test Case Using Classes” on page 32-52
- “Write Setup and Teardown Code Using Classes” on page 32-57

- “Create Simple Test Suites” on page 32-103
- “Run Tests for Various Workflows” on page 32-106
- “Analyze Test Case Results” on page 32-133
- “Analyze Failed Test Results” on page 32-136

Write Simple Test Case Using Classes

This example shows how to write a unit test for a MATLAB® function, `quadraticSolver.m`.

Create `quadraticSolver.m` Function

The following MATLAB function solves quadratic equations. Create this function in a folder on your MATLAB path.

```
function roots = quadraticSolver(a, b, c)
% quadraticSolver returns solutions to the
% quadratic equation a*x^2 + b*x + c = 0.

if ~isa(a,'numeric') || ~isa(b,'numeric') || ~isa(c,'numeric')
    error('quadraticSolver:InputMustBeNumeric', ...
        'Coefficients must be numeric.');
```

end

```
roots(1) = (-b + sqrt(b^2 - 4*a*c)) / (2*a);
roots(2) = (-b - sqrt(b^2 - 4*a*c)) / (2*a);

end
```

Create `SolverTest` Class Definition

To use the `matlab.unittest` framework, write MATLAB functions (tests) in the form of a *test case*, a class derived from `matlab.unittest.TestCase`.

Create a subclass, `SolverTest`.

```
% Copyright 2015 The MathWorks, Inc.

classdef SolverTest < matlab.unittest.TestCase

    methods (Test)

    end
```

end

The following steps show how to create specific tests. Put these tests inside the methods block with the (Test) attribute.

Create Test Method for Real Solutions

Create a test method, `testRealSolution`, to verify that `quadraticSolver` returns the right value for real solutions. For example, the equation $x^2 - 3x + 2 = 0$ has real solutions $x = 1$ and $x = 2$. This method calls `quadraticSolver` with the inputs of this equation. The solution, `expSolution`, is `[2,1]`.

Use the `matlab.unittest.TestCase` method, `verifyEqual` to compare the output of the function, `actSolution`, to the desired output, `expSolution`. If the qualification fails, the test continues execution.

```
% Copyright 2015 The MathWorks, Inc.

function testRealSolution(testCase)
    actSolution = quadraticSolver(1,-3,2);
    expSolution = [2,1];
    testCase.verifyEqual(actSolution,expSolution)
end
```

Add this function inside the methods (Test) block.

Create Test Method for Imaginary Solutions

Create a test to verify that `quadraticSolver` returns the right value for imaginary solutions. For example, the equation $x^2 - 2x + 10 = 0$ has imaginary solutions $x = -1 + 3i$ and $x = -1 - 3i$. Add this function, `testImaginarySolution`, inside the methods (Test) block.

```
% Copyright 2015 The MathWorks, Inc.

function testImaginarySolution(testCase)
    actSolution = quadraticSolver(1,2,10);
```

```
    expSolution = [-1+3i, -1-3i];
    testCase.verifyEqual(actSolution,expSolution)
end
```

The order of the tests within the block does not matter.

Save Class Definition

The following is the complete SolverTest class definition. Save this file in a folder on your MATLAB path.

```
classdef SolverTest < matlab.unittest.TestCase
    % SolverTest tests solutions to the quadratic equation
    % a*x^2 + b*x + c = 0

    methods (Test)
        function testRealSolution(testCase)
            actSolution = quadraticSolver(1,-3,2);
            expSolution = [2,1];
            testCase.verifyEqual(actSolution,expSolution);
        end
        function testImaginarySolution(testCase)
            actSolution = quadraticSolver(1,2,10);
            expSolution = [-1+3i, -1-3i];
            testCase.verifyEqual(actSolution,expSolution);
        end
    end
end

end
```

Run Tests in SolverTest Test Case

Run all the tests in the SolverTest class definition file.

```
testCase = SolverTest;
res = run(testCase)
```

```
Running SolverTest
..
Done SolverTest
```

```

res =

    1x2 TestResult array with properties:

        Name
        Passed
        Failed
        Incomplete
        Duration
        Details

Totals:
    2 Passed, 0 Failed, 0 Incomplete.
    0.056647 seconds testing time.

```

Run Single Test Method

To run the single test, `testRealSolution`:

```

testCase = SolverTest;
res = run(testCase, 'testRealSolution')

```

```

Running SolverTest
.
Done SolverTest

```

```

res =

    TestResult with properties:

        Name: 'SolverTest/testRealSolution'
        Passed: 1
        Failed: 0
        Incomplete: 0
        Duration: 0.0139
        Details: [1x1 struct]

Totals:
    1 Passed, 0 Failed, 0 Incomplete.

```

0.013942 seconds testing time.

See Also

Related Examples

- “Author Class-Based Unit Tests in MATLAB” on page 32-48
- “Write Setup and Teardown Code Using Classes” on page 32-57
- “Analyze Test Case Results” on page 32-133
- “Create Simple Test Suites” on page 32-103

Write Setup and Teardown Code Using Classes

In this section...

“Test Fixtures” on page 32-57

“Test Case with Method-Level Setup Code” on page 32-57

“Test Case with Class-Level Setup Code” on page 32-58

Test Fixtures

Test fixtures are setup and teardown code that sets up the pretest state of the system and returns it to the original state after running the test. Setup and teardown methods are defined in the `TestCase` class by the following method attributes:

- `TestMethodSetup` and `TestMethodTeardown` methods run before and after each test method.
- `TestClassSetup` and `TestClassTeardown` methods run before and after all test methods in the test case.

The testing framework guarantees that `TestMethodSetup` and `TestClassSetup` methods of superclasses are executed before those in subclasses.

It is good practice for test authors to perform all teardown activities from within the `TestMethodSetup` and `TestClassSetup` blocks using the `addTeardown` method instead of implementing corresponding teardown methods in the `TestMethodTeardown` and `TestClassTeardown` blocks. This guarantees the teardown is executed in the reverse order of the setup and also ensures that the test content is exception safe.

Test Case with Method-Level Setup Code

The following test case, `FigurePropertiesTest`, contains setup code at the method level. The `TestMethodSetup` method creates a figure before running each test, and `TestMethodTeardown` closes the figure afterwards. As discussed previously, you should try to define teardown activities with the `addTeardown` method. However, for illustrative purposes, this example shows the implementation of a `TestMethodTeardown` block.

```
classdef FigurePropertiesTest < matlab.unittest.TestCase
    properties
```

```
        TestFigure
    end

    methods(TestMethodSetup)
        function createFigure(testCase)
            % comment
            testCase.TestFigure = figure;
        end
    end

    methods(TestMethodTeardown)
        function closeFigure(testCase)
            close(testCase.TestFigure)
        end
    end

    methods(Test)

        function defaultCurrentPoint(testCase)

            cp = testCase.TestFigure.CurrentPoint;
            testCase.verifyEqual(cp, [0 0], ...
                'Default current point is incorrect')
        end

        function defaultCurrentObject(testCase)
            import matlab.unittest.constraints.IsEmpty

            co = testCase.TestFigure.CurrentObject;
            testCase.verifyThat(co, IsEmpty, ...
                'Default current object should be empty')
        end
    end

end

end
```

Test Case with Class-Level Setup Code

The following test case, `BankAccountTest`, contains setup code at the class level.

To setup the `BankAccountTest`, which tests the `BankAccount` class example described in “Developing Classes — Typical Workflow”, add a `TestClassSetup` method, `addBankAccountClassToPath`. This method adds the path to the `BankAccount`

example file. Typically, you set up the path using a PathFixture. this example performs the setup and teardown activities manually for illustrative purposes.

```
classdef BankAccountTest < matlab.unittest.TestCase
    % Tests the BankAccount class.

    methods (TestClassSetup)
        function addBankAccountClassToPath(testCase)
            p = path;
            testCase.addTeardown(@path,p);
            addpath(fullfile(matlabroot, 'help', 'techdoc', 'matlab_oop', ...
                'examples'));
        end
    end

    methods (Test)
        function testConstructor(testCase)
            b = BankAccount(1234, 100);
            testCase.verifyEqual(b.AccountNumber, 1234, ...
                'Constructor failed to correctly set account number');
            testCase.verifyEqual(b.AccountBalance, 100, ...
                'Constructor failed to correctly set account balance');
        end

        function testConstructorNotEnoughInputs(testCase)
            import matlab.unittest.constraints.Throws;
            testCase.verifyThat(@()BankAccount, ...
                Throws('MATLAB:minrhs'));
        end

        function testDesposit(testCase)
            b = BankAccount(1234, 100);
            b.deposit(25);
            testCase.verifyEqual(b.AccountBalance, 125);
        end

        function testWithdraw(testCase)
            b = BankAccount(1234, 100);
            b.withdraw(25);
            testCase.verifyEqual(b.AccountBalance, 75);
        end

        function testNotifyInsufficientFunds(testCase)
            callbackExecuted = false;
        end
    end
end
```

```
function testCallback(~,~)
    callbackExecuted = true;
end

b = BankAccount(1234, 100);
b.addListener('InsufficientFunds', @testCallback);

b.withdraw(50);
testCase.assertFalse(callbackExecuted, ...
    'The callback should not have executed yet');
b.withdraw(60);
testCase.verifyTrue(callbackExecuted, ...
    'The listener callback should have fired');
end
end
end
```

See Also

`addTeardown` | `matlab.unittest.TestCase`

Related Examples

- “Author Class-Based Unit Tests in MATLAB” on page 32-48
- “Write Simple Test Case Using Classes” on page 32-52

Types of Qualifications

Qualifications are functions for testing values and responding to failures. There are four types of qualifications:

- Verifications — Produce and record failures without throwing an exception, meaning the remaining tests run to completion.
- Assumptions — Ensure that a test runs only when certain preconditions are satisfied and the event should not produce a test failure. When an assumption failure occurs, the testing framework marks the test as filtered.
- Assertions — Ensure that the preconditions of the current test are met.
- Fatal assertions — Use this qualification when the failure at the assertion point renders the remainder of the current test method invalid or the state is unrecoverable.

Type of Test	Verification	Assumption	Assertion	Fatal Assertion
Value is true.	<code>verifyTrue</code>	<code>assumeTrue</code>	<code>assertTrue</code>	<code>fatalAssertTrue</code>
Value is false.	<code>verifyFalse</code>	<code>assumeFalse</code>	<code>assertFalse</code>	<code>fatalAssertFalse</code>
Value is equal to specified value.	<code>verifyEqual</code>	<code>assumeEqual</code>	<code>assertEqual</code>	<code>fatalAssertEqual</code>
Value is not equal to specified value.	<code>verifyNotEqual</code>	<code>assumeNotEqual</code>	<code>assertNotEqual</code>	<code>fatalAssertNotEqual</code>
Two values are handles to same instance.	<code>verifySameHandle</code>	<code>assumeSameHandle</code>	<code>assertSameHandle</code>	<code>fatalAssertSameHandle</code>
Value is not handle to specified instance.	<code>verifyNotSameHandle</code>	<code>assumeNotSameHandle</code>	<code>assertNotSameHandle</code>	<code>fatalAssertNotSameHandle</code>
Function returns true when evaluated.	<code>verifyReturnsTrue</code>	<code>assumeReturnsTrue</code>	<code>assertReturnsTrue</code>	<code>fatalAssertReturnsTrue</code>
Test produces unconditional failure.	<code>verifyFail</code>	<code>assumeFail</code>	<code>assertFail</code>	<code>fatalAssertFail</code>

Type of Test	Verification	Assumption	Assertion	Fatal Assertion
Value meets given constraint.	verifyThat	assumeThat	assertThat	fatalAssertThat
Value is greater than specified value.	verifyGreaterThan	assumeGreaterThan	assertGreaterThan	fatalAssertGreaterThan
Value is greater than or equal to specified value.	verifyGreaterThanOrEqual	assumeGreaterThanOrEqual	assertGreaterThanOrEqual	fatalAssertGreaterThanOrEqual
Value is less than specified value.	verifyLessThan	assumeLessThan	assertLessThan	fatalAssertLessThan
Value is less than or equal to specified value.	verifyLessThanOrEqual	assumeLessThanOrEqual	assertLessThanOrEqual	fatalAssertLessThanOrEqual
Value is exact specified class.	verifyClass	assumeClass	assertClass	fatalAssertClass
Value is object of specified type.	verifyInstanceOf	assumeInstanceOf	assertInstanceOf	fatalAssertInstanceOf
Value is empty.	verifyEmpty	assumeEmpty	assertEmpty	fatalAssertEmpty
Value is not empty.	verifyNotEmpty	assumeNotEmpty	assertNotEmpty	fatalAssertNotEmpty
Value has specified size.	verifySize	assumeSize	assertSize	fatalAssertSize
Value has specified length.	verifyLength	assumeLength	assertLength	fatalAssertLength
Value has specified element count.	verifyNumElements	assumeNumElements	assertNumElements	fatalAssertNumElements
String contains specified string.	verifySubstring	assumeSubstring	assertSubstring	fatalAssertSubstring
Text matches specified regular expression.	verifyMatches	assumeMatches	assertMatches	fatalAssertMatches

Type of Test	Verification	Assumption	Assertion	Fatal Assertion
Function throws specified exception.	<code>verifyError</code>	<code>assumeError</code>	<code>assertError</code>	<code>fatalAssertError</code>
Function issues specified warning.	<code>verifyWarning</code>	<code>assumeWarning</code>	<code>assertWarning</code>	<code>fatalAssertWarning</code>
Function issues no warnings.	<code>verifyWarningFree</code>	<code>assumeWarningFree</code>	<code>assertWarningFree</code>	<code>fatalAssertWarningFree</code>

See Also

`Assertable` | `Assumable` | `FatalAssertable` | `Verifiable` | `matlab.unittest.qualifications`

Tag Unit Tests

You can use test tags to group tests into categories and then run tests with specified tags. Typical test tags identify a particular feature or describe the type of test.

Tag Tests

To define test tags, use a cell array of meaningful character vectors or a string array. For example, `TestTags = {'Unit'}` or `TestTags = ["Unit", "FeatureA"]`.

- To tag individual tests, use the `TestTags` method attribute.
- To tag all the tests within a class, use the `TestTags` class attribute. If you use the `TestTags` class attribute in a superclass, tests in the subclasses inherit the tags.

This sample test class, `ExampleTagTest`, uses the `TestTags` method attribute to tag individual tests.

```
classdef ExampleTagTest < matlab.unittest.TestCase
    methods (Test)
        function testA (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'Unit'})
        function testB (testCase)
            % test code
        end
        function testC (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'Unit', 'FeatureA'})
        function testD (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'System', 'FeatureA'})
        function testE (testCase)
            % test code
        end
    end
end
```

Several of the tests in class `ExampleTagTest` are tagged. For example, `testD` is tagged with 'Unit' and 'FeatureA'. One test, `testA`, is not tagged.

This sample test class, `ExampleTagClassTest`, uses a `TestTags` class attribute to tag all the tests within the class, and a `TestTags` method attribute to add tags to individual tests.

```
classdef (TestTags = {'FeatureB'}) ...
    ExampleTagClassTest < matlab.unittest.TestCase
    methods (Test)
        function testF (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'FeatureC', 'System'})
        function testG (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'System', 'FeatureA'})
        function testH (testCase)
            % test code
        end
    end
end
```

Each test in class `ExampleTagClassTest` is tagged with 'FeatureB'. Additionally, individual tests are tagged with various tags including 'FeatureA', 'FeatureC', and 'System'.

Select and Run Tests

There are three ways of selecting and running tagged tests:

- “Run Selected Tests Using `runtests`” on page 32-65
- “Select Tests Using TestSuite Methods” on page 32-66
- “Select Tests Using HasTag Selector” on page 32-67

Run Selected Tests Using `runtests`

Use the `runtests` function to select and run tests without explicitly creating a test suite. Select and run all the tests from `ExampleTagTest` and `ExampleTagClassTest` that include the 'FeatureA' tag.

```
results = runtests({'ExampleTagTest', 'ExampleTagClassTest'}, 'Tag', 'FeatureA');
```

```
Running ExampleTagTest
```

```
..
```

```
Done ExampleTagTest
```

```
-----
```

```
Running ExampleTagClassTest
```

```
.
```

```
Done ExampleTagClassTest
```

```
-----
```

`runtests` selected and ran three tests.

Display the results in a table.

```
table(results)
```

```
ans =
```

```
3x6 table
```

Name	Passed	Failed	Incomplete	Duration	De
'ExampleTagTest/testE'	true	false	false	0.00039529	[1x
'ExampleTagTest/testD'	true	false	false	0.00045658	[1x
'ExampleTagClassTest/testH'	true	false	false	0.00043899	[1x

The selected tests are `testE` and `testD` from `ExampleTagTest`, and `testH` from `ExampleTagClassTest`.

Select Tests Using TestSuite Methods

Create a suite of tests from the `ExampleTagTest` class that are tagged with 'FeatureA'.

```
import matlab.unittest.TestSuite
sA = TestSuite.fromClass(?ExampleTagTest, 'Tag', 'FeatureA');
```

Create a suite of tests from the `ExampleTagClassTest` class that are tagged with 'FeatureC'.

```
sB = TestSuite.fromFile('ExampleTagClassTest.m', 'Tag', 'FeatureC');
```

Concatenate the suite and view the names of the tests.

```

suite = [sA sB];
{suite.Name}'

ans =

    3×1 cell array

    'ExampleTagTest/testE'
    'ExampleTagTest/testD'
    'ExampleTagClassTest/testG'

```

Select Tests Using HasTag Selector

Create a suite of all the tests from the `ExampleTagTest` and `ExampleTagClassTest` classes.

```

import matlab.unittest.selectors.HasTag
sA = TestSuite.fromClass(?ExampleTagTest);
sB = TestSuite.fromFile('ExampleTagClassTest.m');
suite = [sA sB];

```

Select all the tests that do not have tags.

```
s1 = suite.selectIf(~HasTag)
```

```
s1 =
```

```
Test with properties:
```

```

        Name: 'ExampleTagTest/testA'
    ProcedureName: 'testA'
        TestClass: "ExampleTagTest"
        BaseFolder: 'C:\work'
    Parameterization: [0×0 matlab.unittest.parameters.EmptyParameter]
    SharedTestFixtures: [0×0 matlab.unittest.fixtures.EmptyFixture]
        Tags: {1×0 cell}

```

```
Tests Include:
```

```
    0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Select all the tests with the 'Unit' tag and display their names.

```

s2 = suite.selectIf(HasTag('Unit'));
{s2.Name}'

```

```
ans =  
  
3×1 cell array  
  
'ExampleTagTest/testD'  
'ExampleTagTest/testB'  
'ExampleTagTest/testC'
```

Select all the tests with the 'FeatureB' or 'System' tag using a constraint.

```
import matlab.unittest.constraints.IsEqualTo  
constraint = IsEqualTo('FeatureB') | IsEqualTo('System');  
s3 = suite.selectIf(HasTag(constraint));  
{s3.Name}'
```

```
ans =  
  
4×1 cell array  
  
'ExampleTagTest/testE'  
'ExampleTagClassTest/testH'  
'ExampleTagClassTest/testG'  
'ExampleTagClassTest/testF'
```

See Also

[matlab.unittest.TestCase](#) | [matlab.unittest.TestSuite](#) |
[matlab.unittest.constraints](#) | [matlab.unittest.selectors.HasTag](#) |
[runtests](#)

Write Tests Using Shared Fixtures

This example shows how to use shared fixtures when creating tests. You can share test fixtures across test classes using the `SharedTestFixtures` attribute of the `TestCase` class. To exemplify this attribute, create multiple test classes in a subdirectory of your current working folder. The test methods are shown only at a high level.

The two test classes used in this example test the `DocPolynom` class and the `BankAccount` class. You can access both classes in MATLAB, but you must add them to the MATLAB path. A path fixture adds the directory to the current path, runs the tests, and removes the directory from the path. Since both classes require the same addition to the path, the tests use a shared fixture.

Create a Test for the DocPolynom Class

Create a test file for the `DocPolynom` class. Create the shared fixture by specifying the `SharedTestFixtures` attribute for the `TestCase` and passing in a `PathFixture`.

DocPolynomTest Class Definition File

```
classdef (SharedTestFixtures={matlab.unittest.fixtures.PathFixture( ...
    fullfile(matlabroot,'help','techdoc','matlab_oop','examples')}}) ...
    DocPolynomTest < matlab.unittest.TestCase
    % Tests the DocPolynom class.

    properties
        msgEqn = 'Equation under test: ';
    end

    methods (Test)
        function testConstructor(testCase)
            p = DocPolynom([1, 0, 1]);
            testCase.verifyClass(p, ?DocPolynom)
        end

        function testAddition(testCase)
            p1 = DocPolynom([1, 0, 1]);
            p2 = DocPolynom([5, 2]);

            actual = p1 + p2;
            expected = DocPolynom([1, 5, 3]);

            msg = [testCase.msgEqn, ...
```

```

        '(x^2 + 1) + (5*x + 2) = x^2 + 5*x + 3'];
    testCase.verifyEqual(actual, expected, msg)
end

function testMultiplication(testCase)
    p1 = DocPolynom([1, 0, 3]);
    p2 = DocPolynom([5, 2]);

    actual = p1 * p2;
    expected = DocPolynom([5, 2, 15, 6]);

    msg = [testCase.msgEqn,...
        '(x^2 + 3) * (5*x + 2) = 5*x^3 + 2*x^2 + 15*x + 6'];
    testCase.verifyEqual(actual, expected, msg)
end

end
end

```

Create a Test for the BankAccount Class

Create a test file for the BankAccount class. Create the shared fixture by specifying the SharedTestFixtures attribute for the TestCase and passing in a PathFixture.

BankAccountTest Class Definition File

```

classdef (SharedTestFixtures={matlab.unittest.fixtures.PathFixture( ...
    fullfile(matlabroot, 'help', 'techdoc', 'matlab_oop', ...
    'examples')}}) BankAccountTest < matlab.unittest.TestCase
    % Tests the BankAccount class.

    methods (Test)
        function testConstructor(testCase)
            b = BankAccount(1234, 100);
            testCase.verifyEqual(b.AccountNumber, 1234, ...
                'Constructor failed to correctly set account number')
            testCase.verifyEqual(b.AccountBalance, 100, ...
                'Constructor failed to correctly set account balance')
        end

        function testConstructorNotEnoughInputs(testCase)
            import matlab.unittest.constraints.Throws
            testCase.verifyThat(@()BankAccount, ...
                Throws('MATLAB:minrhs'))
        end
    end
end

```

```

function testDeposit(testCase)
    b = BankAccount(1234, 100);
    b.deposit(25)
    testCase.verifyEqual(b.AccountBalance, 125)
end

function testWithdraw(testCase)
    b = BankAccount(1234, 100);
    b.withdraw(25)
    testCase.verifyEqual(b.AccountBalance, 75)
end

function testNotifyInsufficientFunds(testCase)
    callbackExecuted = false;
    function testCallback(~,~)
        callbackExecuted = true;
    end

    b = BankAccount(1234, 100);
    b.addlistener('InsufficientFunds', @testCallback);

    b.withdraw(50)
    testCase.assertFalse(callbackExecuted, ...
        'The callback should not have executed yet')
    b.withdraw(60)
    testCase.verifyTrue(callbackExecuted, ...
        'The listener callback should have fired')
end
end
end

```

Build the Test Suite

The classes `DocPolynomTest.m` and `BankAccountTest.m` are in your working directory. Create a test suite from your current working directory. If you have additional tests, they are included in the suite when you use the `TestSuite.fromFolder` method. Create the test suite at the command prompt.

```

import matlab.unittest.TestSuite;
suiteFolder = TestSuite.fromFolder(pwd);

```

Run the Tests

At the command prompt, run the tests in the test suite.

```
result = run(suiteFolder);
```

```
Setting up PathFixture.
```

```
Description: Adds 'C:\Program Files\MATLAB\R2013b\help\techdoc\matlab_oop\examples' to
```

```
Running BankAccountTest
```

```
.....
```

```
Done BankAccountTest
```

```
Running DocPolynomTest
```

```
...
```

```
Done DocPolynomTest
```

```
Tearing down PathFixture.
```

```
Description: Restores the path to its previous state.
```

The test framework sets up the test fixture, runs all the tests in each file, and then tears the fixture down. If the path fixture was set up and torn down using `TestClassSetup` methods, the fixture is set up and torn down twice—once for each test file.

See Also

`PathFixture` | `TestCase` | `matlab.unittest.fixtures`

Create Basic Custom Fixture

This example shows how to create a basic custom fixture that changes the display format to hexadecimal representation. The example also shows to use the fixture to test a function that displays a column of numbers as text. After the testing completes, the framework restores the display format to its pretest state.

Create FormatHexFixture Class Definition

In a file in your working folder, create a new class, `FormatHexFixture` that inherits from the `matlab.unittest.fixtures.Fixture` class. Since we want the fixture to restore the pretest state of the MATLAB display format, create an `OriginalFormat` property to keep track of the original display format.

```
classdef FormatHexFixture < matlab.unittest.fixtures.Fixture
    properties (Access=private)
        OriginalFormat
    end
end
```

Implement Setup and Teardown Methods

Subclasses of the `Fixture` class must implement the `setup` method. Use this method to record the pretest display format, and set the format to `'hex'`. Use the `teardown` method to restore the original display format. Define the `setup` and `teardown` methods in the `methods` block of the `FormatHexFixture.m` file.

```
    methods
        function setup(fixture)
            fixture.OriginalFormat = get(0, 'Format');
            set(0, 'Format', 'hex')
        end
        function teardown(fixture)
            set(0, 'Format', fixture.OriginalFormat)
        end
    end
end
```

Apply Custom Fixture

In a file in your working folder, create the following test class, `SampleTest.m`.

```
classdef SampleTest < matlab.unittest.TestCase
    methods (Test)
```

```
function test1(testCase)
    testCase.applyFixture(FormatHexFixture);
    actStr = getColumnForDisplay([1;2;3], 'Small Integers');
    expStr = ['Small Integers '
             '3ff0000000000000'
             '4000000000000000'
             '4008000000000000'];
    testCase.verifyEqual(actStr, expStr)
end
end
end

function str = getColumnForDisplay(values, title)
    elements = cell(numel(values)+1, 1);
    elements{1} = title;
    for idx = 1:numel(values)
        elements{idx+1} = displayNumber(values(idx));
    end
    str = char(elements);
end

function str = displayNumber(n)
    str = strtrim(evalc('disp(n);'));
end
```

This test applies the custom fixture and verifies that the displayed column of hexadecimal representation is as expected.

At the command prompt, run the test.

```
run(SampleTest);
Running SampleTest
.
Done SampleTest
_____
```

See Also

`matlab.unittest.fixtures.Fixture`

Related Examples

- “Create Advanced Custom Fixture” on page 32-76
- “Write Tests Using Shared Fixtures” on page 32-69

Create Advanced Custom Fixture

This example shows how to create a custom fixture that sets an environment variable. Prior to testing, this fixture will save the current `UserName` variable.

Create `UserNameEnvironmentVariableFixture` Class Definition

In a file in your working folder, create a new class, `UserNameEnvironmentVariableFixture` that inherits from the `matlab.unittest.fixtures.Fixture` class. Since you want to pass the fixture a user name, create a `UserName` property to pass the data between methods.

```
classdef UserNameEnvironmentVariableFixture < ...
    matlab.unittest.fixtures.Fixture

    properties (SetAccess=private)
        UserName
    end
```

Define Fixture Constructor

In the methods block of the `UserNameEnvironmentVariableFixture.m` file, create a constructor method that validates the input and defines the `SetupDescription`. Have the constructor accept a character vector and set the fixture's `UserName` property.

```
methods
    function fixture = UserNameEnvironmentVariableFixture(name)
        validateattributes(name, {'char'}, {'row'}, '', 'UserName')
        fixture.UserName = name;
        fixture.SetupDescription = sprintf( ...
            'Set the UserName environment variable to "%s".', ...
            fixture.UserName);
    end
```

Implement setup Method

Subclasses of the `Fixture` class must implement the `setup` method. Use this method to save the original `UserName` variable. This method also defines the `TeardownDescription` and registers the teardown task of setting the `UserName` to the original state after testing.

Define the `setup` method within the methods block of the `UserNameEnvironmentVariableFixture.m` file.

```
function setup(fixture)
    originalUserName = getenv('UserName');
    fixture.assertNotEmpty(originalUserName, ...
```

```

        'An existing UserName environment variable must be defined.')
    fixture.addTeardown(@setenv, 'UserName', originalUserName)
    fixture.TearDownDescription = sprintf(...
        'Restored the UserName environment variable to "%s".',...
        originalUserName);
    setenv('UserName', fixture.UserName)
end
end

```

Implement isCompatible Method

Classes that derive from `Fixture` must implement the `isCompatible` method if the constructor is configurable. Since you can configure the `UserName` property through the constructor, `UserNameEnvironmentVariableFixture` must implement `isCompatible`.

The `isCompatible` method is called with two instances of the same class. In this case, it is called with two instances of `UserNameEnvironmentVariableFixture`. The testing framework considers the two instances compatible if their `UserName` properties are equal.

In a new methods block within `UserNameEnvironmentVariableFixture.m`, define an `isCompatible` method which returns logical 1 (true) or logical 0 (false).

```

methods (Access=protected)
    function bool = isCompatible(fixture, other)
        bool = strcmp(fixture.UserName, other.UserName);
    end
end

```

Fixture Class Definition Summary

Below are the complete contents of `UserNameEnvironmentVariableFixture.m`.

```

classdef UserNameEnvironmentVariableFixture < ...
    matlab.unittest.fixtures.Fixture

    properties (SetAccess=private)
        UserName
    end

    methods
        function fixture = UserNameEnvironmentVariableFixture(name)
            validateattributes(name, {'char'}, {'row'}, '', 'UserName')
            fixture.UserName = name;
            fixture.SetupDescription = sprintf( ...
                'Set the UserName environment variable to "%s".',...
                fixture.UserName);
        end
    end
end

```

```
function setup(fixture)
    originalUserName = getenv('UserName');
    fixture.assertNotEmpty(originalUserName, ...
        'An existing UserName environment variable must be defined.')
    fixture.addTeardown(@setenv, 'UserName', originalUserName)
    fixture.TearDownDescription = sprintf(...
        'Restored the UserName environment variable to "%s".',...
        originalUserName);
    setenv('UserName', fixture.UserName)
end
end

methods (Access=protected)
function bool = isCompatible(fixture, other)
    bool = strcmp(fixture.UserName, other.UserName);
end
end
end
```

Apply Custom Fixture to Single Test Class

In a file in your working folder, create the following test class, `ExampleTest.m`.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods (TestMethodSetup)
        function mySetup(testCase)
            testCase.applyFixture(...
                UserNameEnvironmentVariableFixture('David'));
        end
    end
end

methods (Test)
    function t1(~)
        fprintf(1, 'Current UserName: "%s"', getenv('UserName'))
    end
end
end
```

This test uses the `UserNameEnvironmentVariableFixture` for each test in the `ExampleTest` class.

At the command prompt, run the test.

```
run(ExampleTest);
```

```
Running ExampleTest
Current UserName: "David".
Done ExampleTest
```

Apply Custom Fixture as Shared Fixture

In your working folder, create three test classes using a shared fixture. Using a shared fixture allows the `UserNameEnvironmentVariableFixture` to be shared across classes.

Create `testA.m` as follows.

```
classdef (SharedTestFixtures={...
    UserNameEnvironmentVariableFixture('David')}) ...
    testA < matlab.unittest.TestCase
    methods (Test)
        function t1(~)
            fprintf(1, 'Current UserName: "%s"', getenv('UserName'))
        end
    end
end
```

Create `testB.m` as follows.

```
classdef (SharedTestFixtures={...
    UserNameEnvironmentVariableFixture('Andy')}) ...
    testB < matlab.unittest.TestCase
    methods (Test)
        function t1(~)
            fprintf(1, 'Current UserName: "%s"', getenv('UserName'))
        end
    end
end
```

Create `testC.m` as follows.

```
classdef (SharedTestFixtures={...
    UserNameEnvironmentVariableFixture('Andy')}) ...
    testC < matlab.unittest.TestCase
    methods (Test)
        function t1(~)
            fprintf(1, 'Current UserName: "%s"', getenv('UserName'))
        end
    end
end
```

At the command prompt, run the tests.

```
runtests({'testA', 'testB', 'testC'});
```

```
Setting up UserNameEnvironmentVariableFixture
Done setting up UserNameEnvironmentVariableFixture: Set the UserName environment variable
-----

Running testA
Current UserName: "David".
Done testA
-----

Tearing down UserNameEnvironmentVariableFixture
Done tearing down UserNameEnvironmentVariableFixture: Restored the UserName environment
-----

Setting up UserNameEnvironmentVariableFixture
Done setting up UserNameEnvironmentVariableFixture: Set the UserName environment variable
-----

Running testB
Current UserName: "Andy".
Done testB
-----

Running testC
Current UserName: "Andy".
Done testC
-----

Tearing down UserNameEnvironmentVariableFixture
Done tearing down UserNameEnvironmentVariableFixture: Restored the UserName environment
-----
```

Recall that the fixtures are compatible if their `UserName` properties match. The tests in `testA` and `testB` use incompatible shared fixtures, since 'David' is not equal to 'Andy'. Therefore, the framework invokes the fixture `teardown` and `setup` methods between calls to `testA` and `testB`. However, the shared test fixture in `testC` is compatible with the fixture in `testB`, so the framework doesn't repeat fixture `teardown` and `setup` before `testC`.

Alternative Approach to Calling `addTeardown` in `setup` Method

An alternate approach to using the `addTeardown` method within the `setup` method is to implement a separate `teardown` method. Instead of the `setup` method described above, implement the following `setup` and `teardown` methods within `UserNameEnvironmentVariableFixture.m`.

Alternate UserNameEnvironmentVariableFixture Class Definition

```

classdef UserNameEnvironmentVariableFixture < ...
    matlab.unittest.fixtures.Fixture

    properties (Access=private)
        OriginalUser
    end
    properties (SetAccess=private)
        UserName
    end

    methods
        function fixture = UserNameEnvironmentVariableFixture(name)
            validateattributes(name, {'char'}, {'row'}, '', 'UserName')
            fixture.UserName = name;
            fixture.SetupDescription = sprintf( ...
                'Set the UserName environment variable to "%s".', ...
                fixture.UserName);
        end

        function setup(fixture)
            fixture.OriginalUser = getenv('UserName');
            fixture.assertNotEmpty(fixture.OriginalUser, ...
                'An existing UserName environment variable must be defined.')
            setenv('UserName', fixture.UserName)
        end

        function teardown(fixture)
            fixture.TearardownDescription = sprintf(...
                'Restored the UserName environment variable to "%s".', ...
                fixture.OriginalUser);
            setenv('UserName', fixture.OriginalUser)
        end

        end

    methods (Access=protected)
        function bool = isCompatible(fixture, other)
            bool = strcmp(fixture.UserName, other.UserName);
        end
    end
end

```

The `setup` method does not contain a call to `addTeardown` or a definition for `TearardownDescription`. These tasks are relegated to the `teardown` method. The alternative class definition contains an additional property, `OriginalUser`, which allows the information to be passed between methods.

See Also

`matlab.unittest.fixtures.Fixture`

Related Examples

- “Create Basic Custom Fixture” on page 32-73
- “Write Tests Using Shared Fixtures” on page 32-69

Create Basic Parameterized Test

This example shows how to create a basic parameterized test.

Create Function to Test

In your working folder, create a function in the file `sierpinski.m`. This function returns a matrix representing an image of a Sierpinski carpet fractal. It takes as input the fractal level and an optional data type.

```
function carpet = sierpinski(nLevels,classname)
if nargin == 1
    classname = 'single';
end

mSize = 3^nLevels;
carpet = ones(mSize,classname);

cutCarpet(1,1,mSize,nLevels) % begin recursion

function cutCarpet(x,y,s,cL)
    if cL
        ss = s/3; % define subsize
        for lx = 0:2
            for ly = 0:2
                if lx == 1 && ly == 1
                    % remove center square
                    carpet(x+ss:x+2*ss-1,y+ss:y+2*ss-1) = 0;
                else
                    % recurse
                    cutCarpet(x + lx*ss, y + ly*ss, ss, cL-1)
                end
            end
        end
    end
end
end
end
end
end
```

Create TestCarpet Test Class

In a file in your working folder, create a new class, `TestCarpet`, to test the `sierpinski` function.

```
classdef TestCarpet < matlab.unittest.TestCase
```

Define properties Block

Define the properties used for parameterized testing. In the `TestCarpet` class, define these properties in a property block with the `TestParameter` attribute.

```
properties (TestParameter)
    type = {'single', 'double', 'uint16'};
    level = struct('small', 2, 'medium', 4, 'large', 6);
    side = struct('small', 9, 'medium', 81, 'large', 729);
end
```

The `type` property contains the different data types you want to test. The `level` property contains the different fractal level you want to test. The `side` property contains the number of rows and columns in the Sierpinski carpet matrix and corresponds to the `level` property. To provide meaningful names for each parameterization value, `level` and `side` are defined as structs.

Define Test methods Block

Define the following test methods in the `TestCarpet` class.

```
methods (Test)
    function testRemainPixels(testCase, level)
        % expected number pixels equal to 1
        expPixelCount = 8^level;
        % actual number pixels equal to 1
        actPixels = find(sierpinski(level));
        testCase.verifyNumElements(actPixels, expPixelCount)
    end

    function testClass(testCase, type, level)
        testCase.verifyClass(...
            sierpinski(level, type), type);
    end

    function testDefaultL1Output(testCase)
        exp = single([1 1 1; 1 0 1; 1 1 1]);
        testCase.verifyEqual(sierpinski(1), exp)
    end
end
```

The `testRemainPixels` method tests the output of the `sierpinski` function by verifying that the number of nonzero pixels is the same as expected for a particular level.

This method uses the `level` property and, therefore, results in three test elements—one for each value in `level`. The `testClass` method tests the class of the output from the `sierpinski` function with each combination of the `type` and `level` properties. This approach results in nine test elements. The `testDefaultL1Output` test method does not use a `TestParameter` property and, therefore, is not parameterized. This test method verifies that the level 1 matrix contains the expected values. Since the test method is not parameterized, it results in a one test element.

In the test methods above, you did not define the `ParameterCombination` attribute of the `Test` methods block. This attribute is, by default, `'exhaustive'`. The test framework invokes a given test method once for every combination of the test parameters.

Define Test methods Block with ParameterCombination Attribute

Define the following test methods in the `TestCarpet` class to ensure that the matrix output by the `sierpinski` function has the correct number of elements. Set the `ParameterCombination` attribute to `'sequential'`.

```

methods (Test, ParameterCombination='sequential')
    function testNumel(testCase, level, side)
        import matlab.unittest.constraints.HasElementCount
        testCase.verifyThat(sierpinski(level),...
            HasElementCount(side^2))
    end
end
end

```

Test methods with the `ParameterCombination` attribute set to `'sequential'` are invoked once for each corresponding value of the parameter. The properties, `level` and `side`, must have the same number of values. Since these properties each have three values, the `testNumel` method is invoked three times.

TestCarpet Class Definition Summary

The complete contents of `TestCarpet.m` follows.

```

classdef TestCarpet < matlab.unittest.TestCase

    properties (TestParameter)
        type = {'single', 'double', 'uint16'};
        level = struct('small', 2, 'medium', 4, 'large', 6);
        side = struct('small', 9, 'medium', 81, 'large', 729);
    end
end

```

```
end

methods (Test)
    function testRemainPixels(testCase, level)
        % expected number pixels equal to 1
        expPixelCount = 8^level;
        % actual number pixels equal to 1
        actPixels = find(sierpinski(level));
        testCase.verifyNumElements(actPixels,expPixelCount)
    end

    function testClass(testCase, type, level)
        testCase.verifyClass(...
            sierpinski(level,type), type)
    end

    function testDefaultL1Output(testCase)
        exp = single([1 1 1; 1 0 1; 1 1 1]);
        testCase.verifyEqual(sierpinski(1), exp)
    end
end

methods (Test, ParameterCombination='sequential')
    function testNumel(testCase, level, side)
        import matlab.unittest.constraints.HasElementCount
        testCase.verifyThat(sierpinski(level),...
            HasElementCount(side^2))
    end
end
end
```

Run All Tests

At the command prompt, create a suite from `TestCarpet.m`.

```
suite = matlab.unittest.TestSuite.fromFile('TestCarpet.m');
{suite.Name}'
```

```
ans =
```

```
'TestCarpet/testNumel(level=small,side=small)'  
'TestCarpet/testNumel(level=medium,side=medium)'  
'TestCarpet/testNumel(level=large,side=large)'  
'TestCarpet/testRemainPixels(level=small)'  
'TestCarpet/testRemainPixels(level=medium)'
```

```
'TestCarpet/testRemainPixels(level=large) '
'TestCarpet/testClass(type=single,level=small) '
'TestCarpet/testClass(type=single,level=medium) '
'TestCarpet/testClass(type=single,level=large) '
'TestCarpet/testClass(type=double,level=small) '
'TestCarpet/testClass(type=double,level=medium) '
'TestCarpet/testClass(type=double,level=large) '
'TestCarpet/testClass(type=uint16,level=small) '
'TestCarpet/testClass(type=uint16,level=medium) '
'TestCarpet/testClass(type=uint16,level=large) '
'TestCarpet/testDefaultL1Output'
```

The suite had 16 test elements. The element's `Name` indicates any parameterization.

```
suite.run;
```

```
Running TestCarpet
.....
.....
Done TestCarpet
```

Run Tests with level Parameter Property Named small

Use the `selectIf` method of the `TestSuite` to select test elements that use a particular parameterization. Select all test elements that use the parameter name `small` in the `level` parameter property list.

```
s1 = suite.selectIf('ParameterName','small');
{s1.Name}'

ans =

'TestCarpet/testNumel(level=small,side=small) '
'TestCarpet/testRemainPixels(level=small) '
'TestCarpet/testClass(type=single,level=small) '
'TestCarpet/testClass(type=double,level=small) '
'TestCarpet/testClass(type=uint16,level=small) '
```

The suite has five elements.

```
s1.run;
```

```
Running TestCarpet
.....
```

Done TestCarpet

Alternatively, create the same test suite directly from the `fromFile` method of `TestSuite`.

```
import matlab.unittest.selectors.HasParameter
s1 = matlab.unittest.TestSuite.fromFile('TestCarpet.m', ...
    HasParameter('Name', 'small'));
```

See Also

`matlab.unittest.TestCase` | `matlab.unittest.TestSuite.selectIf` |
`matlab.unittest.selectors.HasParameter`

Related Examples

- “Create Advanced Parameterized Test” on page 32-89

Create Advanced Parameterized Test

This example shows how to create a test that is parameterized in the `TestClassSetup`, `TestMethodSetup`, and `Test` methods blocks. The example test class tests the random number generator.

Test Overview

The `TestRand` test class is parameterized at three different levels.

Parameterization Level	Parameterization Definition		Accessible Parameterization Properties
	Method Attribute	Property Attribute	
Test level	Test	TestParameter	TestParameter, MethodSetupParameter, and ClassSetupParameter
Method setup level	TestMethodSetup	MethodSetupParameter	MethodSetupParameter and ClassSetupParameter
Class setup level	TestClassSetup	ClassSetupParameter	ClassSetupParameter

At each test level, you can use the `ParameterCombination` method attribute to specify the test parameterization.

ParameterCombination Attribute	Method Invocation
'exhaustive' (default)	Methods are invoked for all combinations of parameters. The test framework uses this default combination if you do not specify the <code>ParameterCombination</code> attribute.
'sequential'	Methods are invoked with corresponding values from each parameter. Each parameter must contain the same number of values.

ParameterCombination Attribute	Method Invocation
'pairwise'	Methods are invoked for every pair of parameter values at least once. While the test framework guarantees that tests are created for every pair of values at least once, you should not rely on that size, ordering, or specific set of test suite elements.

For example, use the combined methods attribute `TestMethodSetup`, `ParameterCombination='sequential'` to specify sequential combination of the method setup-level parameters defined in the `MethodSetupParameter` properties block.

For this example, class setup-level parameterization defines the type of random number generator. The method setup-level parameterization defines the seed for the random number generator, and the test-level parameterization defines the data type and size of the random number output.

Create TestRand Test Class

In a file in your working folder, create a class that inherits from `matlab.unittest.TestCase`. This class tests various aspects of random number generation.

```
classdef TestRand < matlab.unittest.TestCase
```

Define properties Blocks

Define the properties used for parameterized testing. Each properties block corresponds to parameterization at a particular level.

```
properties (ClassSetupParameter)
    generator = {'twister', 'combRecursive', 'multFibonacci'};
end

properties (MethodSetupParameter)
    seed = {0, 123, 4294967295};
end

properties (TestParameter)
    dim1 = struct('small', 1, 'medium', 2, 'large', 3);
    dim2 = struct('small', 2, 'medium', 3, 'large', 4);
    dim3 = struct('small', 3, 'medium', 4, 'large', 5);
```

```

    type = {'single', 'double'};
end

```

Define Test Class and Test Method Setup Methods

Define the setup methods at the test class and test method level. These methods register the initial random number generator state. After the framework runs the tests, the methods restore the original state. The `ClassSetup` method defines the type of random number generator, and the `TestMethodSetup` seeds the generator.

```

methods (TestClassSetup)
    function ClassSetup(testCase, generator)
        orig = rng;
        testCase.addTeardown(@rng, orig)
        rng(0, generator)
    end
end

methods (TestMethodSetup)
    function MethodSetup(testCase, seed)
        orig = rng;
        testCase.addTeardown(@rng, orig)
        rng(seed)
    end
end

```

Define Sequential Parameterized Test Methods

Define a methods block with the `Test` and `ParameterCombination='sequential'` attributes. The test framework invokes these methods once for each corresponding property value.

```

methods (Test, ParameterCombination='sequential')
    function testSize(testCase, dim1, dim2, dim3)
        testCase.verifySize(rand(dim1, dim2, dim3), [dim1 dim2 dim3])
    end
end

```

The method tests the size of the output for each corresponding parameter in `dim1`, `dim2`, and `dim3`. For example, to test all the 'medium' values use: `testCase.verifySize(rand(2,3,4), [2 3 4]);`. For a given `TestClassSetup` and `TestMethodSetup` parameterization, the framework calls the `testSize` method three times—once each for the 'small', 'medium', and 'large' values.

Define Pairwise Parameterized Test Methods

Define a methods block with the `Test` and `ParameterCombination='pairwise'` attributes. The test framework invokes these methods at least once for every pair of property values.

```
methods (Test, ParameterCombination='pairwise')
    function testRepeatable(testCase,dim1,dim2,dim3)
        state = rng;
        firstRun = rand(dim1,dim2,dim3);
        rng(state)
        secondRun = rand(dim1,dim2,dim3);
        testCase.verifyEqual(firstRun,secondRun)
    end
end
```

The test method verifies that the random number generator results are repeatable. For a given `TestClassSetup` and `TestMethodSetup` parameterization, the framework calls the `testRepeatable` method 10 times to ensure testing of each pair of `dim1`, `dim2`, and `dim3`. However, if the parameter combination attribute is exhaustive, the framework calls the method $3^3=27$ times.

Define Exhaustive Parameterized Test Methods

Define a methods block with the `Test` attribute or no defined parameter combination. The parameter combination is exhaustive by default. The test framework invokes these methods once for every combination of property values.

```
methods (Test)
    function testClass(testCase,dim1,dim2,type)
        testCase.verifyClass(rand(dim1,dim2,type), type)
    end
end
```

The test method verifies that the class of the output from `rand` is the same as the expected class. For a given `TestClassSetup` and `TestMethodSetup` parameterization, the framework calls the `testClass` method $3*3*2=18$ times to ensure testing of each combination of `dim1`, `dim2`, and `type`.

TestRand Class Definition Summary

```
classdef TestRand < matlab.unittest.TestCase
    properties (ClassSetupParameter)
        generator = {'twister', 'combRecursive', 'multFibonacci'};
```

```
end

properties (MethodSetupParameter)
    seed = {0, 123, 4294967295};
end

properties (TestParameter)
    dim1 = struct('small', 1, 'medium', 2, 'large', 3);
    dim2 = struct('small', 2, 'medium', 3, 'large', 4);
    dim3 = struct('small', 3, 'medium', 4, 'large', 5);
    type = {'single', 'double'};
end

methods (TestClassSetup)
    function ClassSetup(testCase, generator)
        orig = rng;
        testCase.addTeardown(@rng, orig)
        rng(0, generator)
    end
end

methods (TestMethodSetup)
    function MethodSetup(testCase, seed)
        orig = rng;
        testCase.addTeardown(@rng, orig)
        rng(seed)
    end
end

methods (Test, ParameterCombination='sequential')
    function testSize(testCase, dim1, dim2, dim3)
        testCase.verifySize(rand(dim1, dim2, dim3), [dim1 dim2 dim3])
    end
end

methods (Test, ParameterCombination='pairwise')
    function testRepeatable(testCase, dim1, dim2, dim3)
        state = rng;
        firstRun = rand(dim1, dim2, dim3);
        rng(state)
        secondRun = rand(dim1, dim2, dim3);
        testCase.verifyEqual(firstRun, secondRun);
    end
end
```

```
        methods (Test)
            function testClass(testCase,dim1,dim2,type)
                testCase.verifyClass(rand(dim1,dim2,type), type)
            end
        end
    end
end
```

Create Suite from All Tests

At the command prompt, create a suite from `TestRand.m` class.

```
suite = matlab.unittest.TestSuite.fromClass(?TestRand)
```

```
suite =
```

```
1x279 Test array with properties:
```

```
Name
ProcedureName
TestClass
BaseFolder
Parameterization
SharedTestFixtures
Tags
```

```
Tests Include:
```

```
17 Unique Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

The test suite contains 279 test elements. For a given `TestClassSetup` and `TestMethodSetup` parameterization, the framework creates $3+10+18=31$ test elements. These 31 elements are called three times—once for each `TestMethodSetup` parameterization resulting in $3*31=93$ test elements for each `TestClassSetup` parameterization. There are three `TestClassSetup` parameterizations resulting in a total of $3*93=279$ test elements.

Examine the names of the first test element.

```
suite(1).Name
```

```
ans =
```

```
'TestRand[generator=twister]/[seed=value1]testClass(dim1=small,dim2=small,type=sing
```

The name of each element is constructed from the combination of the following:

- Test class: `TestRand`
- Class setup property and property name: `[generator=twister]`
- Method setup property and property name: `[seed=value1]`
- Test method name: `testClass`
- Test method properties and property names:
`(dim1=small,dim2=small,type=single)`

The name for the `seed` property isn't particularly meaningful (`value1`). The testing framework provided this name because the `seed` property values are numbers. For a more meaningful name, define the `seed` property as a struct with more descriptive field names.

Run Suite from Class Using Selector

At the command prompt, create a selector to select test elements that test the 'twister' generator for 'single' precision. Omit test elements that use properties with the 'large' name.

```
import matlab.unittest.selectors.HasParameter
s = HasParameter('Property','generator', 'Name','twister') & ...
    HasParameter('Property','type', 'Name','single') & ...
    ~HasParameter('Name','large');
```

```
suite2 = matlab.unittest.TestSuite.fromClass(?TestRand,s)
```

```
suite2 =
```

```
1×12 Test array with properties:
```

```
Name
ProcedureName
TestClass
BaseFolder
Parameterization
SharedTestFixtures
Tags
```

```
Tests Include:
```

```
9 Unique Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

If you first generate the full suite, construct the same test suite as above using the `selectIf` method.

```
suite = matlab.unittest.TestSuite.fromClass(?TestRand);  
suite2 = selectIf(suite,s);
```

Run the test suite.

```
suite2.run;
```

```
Running TestRand
```

```
.....  
..
```

```
Done TestRand
```

Run Suite from Method Using Selector

At the command prompt, create a selector that omits test elements that use properties with the 'large' or 'medium' name. Limit results to test elements from the `testRepeatable` method.

```
import matlab.unittest.selectors.HasParameter  
s = ~(HasParameter('Name','large') | HasParameter('Name','medium'));  
  
suite3 = matlab.unittest.TestSuite.fromMethod(?TestRand,'testRepeatable',s);  
{suite3.Name}'
```

```
ans =
```

```
9x1 cell array
```

```
'TestRand(generator=twister)/[seed=value1]testRepeatable(dim1=small,dim2=small,dim3=small)'  
'TestRand(generator=twister)/[seed=value2]testRepeatable(dim1=small,dim2=small,dim3=small)'  
'TestRand(generator=twister)/[seed=value3]testRepeatable(dim1=small,dim2=small,dim3=small)'  
'TestRand(generator=combRecursive)/[seed=value1]testRepeatable(dim1=small,dim2=small,dim3=small)'  
'TestRand(generator=combRecursive)/[seed=value2]testRepeatable(dim1=small,dim2=small,dim3=small)'  
'TestRand(generator=combRecursive)/[seed=value3]testRepeatable(dim1=small,dim2=small,dim3=small)'  
'TestRand(generator=multFibonacci)/[seed=value1]testRepeatable(dim1=small,dim2=small,dim3=small)'  
'TestRand(generator=multFibonacci)/[seed=value2]testRepeatable(dim1=small,dim2=small,dim3=small)'  
'TestRand(generator=multFibonacci)/[seed=value3]testRepeatable(dim1=small,dim2=small,dim3=small)'
```

Run the test suite.

```
suite3.run;
```

```
Running TestRand
```

```
.....  
Done TestRand
```

Run All Double Precision Tests

At the command prompt, run all the test elements from `TestRand.m` that use the parameter name `'double'`.

```
runtests('TestRand', 'ParameterName', 'double');
```

```
Running TestRand
```

```
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.
```

```
Done TestRand
```

See Also

`matlab.unittest.TestCase` | `matlab.unittest.TestSuite` |
`matlab.unittest.selectors`

Related Examples

- “Create Basic Parameterized Test” on page 32-83

Use External Parameters in Parameterized Test

You can inject variable inputs into your existing class-based test. To provide test data that is defined outside the test file and that should be used iteratively by the test (via parameterized testing), create a `Parameter` and use the `'ExternalParameters'` option to `TestSuite` creation methods such as `TestSuite.fromClass`.

Create the following function to test. The function accepts an array, vectorizes the array, removes 0, Nan, and Inf, and then sorts the array.

```
function Y = cleanData(X)
    Y = X(:);           % Vectorize array
    Y = rmmissing(Y); % Remove NaN
    % Remove 0 and Inf
    idx = (Y==0 | Y==Inf);
    Y = Y(~idx);
    % If array is empty, set to eps
    if isempty(Y)
        Y = eps;
    end
    Y = sort(Y);       % Sort vector
end
```

Create the following parameterized test for the `cleanData` function. The test repeats each of the four test methods for the two data sets that are defined in the `properties` block.

```
classdef TestClean < matlab.unittest.TestCase
    properties (TestParameter)
        Data = struct('clean',[5 3 9;1 42 5;32 5 2],...
            'needsCleaning',[1 13;NaN 0;Inf 42]);
    end
    methods (Test)
        function classCheck(testCase,Data)
            act = cleanData(Data);
            testCase.assertClass(act,'double')
        end
        function sortCheck(testCase,Data)
            act = cleanData(Data);
            testCase.verifyTrue(issorted(act))
        end
    end
end
```

```

end
function finiteCheck(testCase,Data)
    import matlab.unittest.constraints.IsFinite
    act = cleanData(Data);
    testCase.verifyThat(act,IsFinite)
end
function noZeroCheck(testCase,Data)
    import matlab.unittest.constraints.EveryElementOf
    import matlab.unittest.constraints.IsEqualTo
    act = cleanData(Data);
    testCase.verifyThat(EveryElementOf(act),~IsEqualTo(0))
end
end
end

```

Create and run a parameterized test suite. View the results. The framework runs the eight parameterized tests using the data defined in the test file.

```

import matlab.unittest.TestSuite
suite1 = TestSuite.fromClass(?TestClean);
results = suite1.run;
table(results)

```

Running TestClean

```

.....
Done TestClean

```

ans =

8×6 table

Name	Passed	Failed	Incomplete	Du
'TestClean/classCheck(Data=clean)'	true	false	false	
'TestClean/classCheck(Data=needsCleaning)'	true	false	false	0
'TestClean/sortCheck(Data=clean)'	true	false	false	0
'TestClean/sortCheck(Data=needsCleaning)'	true	false	false	0
'TestClean/finiteCheck(Data=clean)'	true	false	false	0
'TestClean/finiteCheck(Data=needsCleaning)'	true	false	false	0
'TestClean/noZeroCheck(Data=clean)'	true	false	false	

```
'TestClean/noZeroCheck(Data=needsCleaning)' true false false
```

Create a data set external to the test file.

```
A = [NaN 2 0;1 Inf 3];
```

Create a parameter with the external data set. The `fromData` method accepts the name of the parameterized property from the `properties` block in `TestClean` and the new data as a cell array (or struct).

```
import matlab.unittest.parameters.Parameter
newData = {A};
param = Parameter.fromData('Data',newData);
```

Create and a new test suite and view the suite element names. The `fromClass` method accepts the new parameter.

```
suite2 = TestSuite.fromClass(?TestClean,'ExternalParameters',param);
{suite2.Name}'
```

```
ans =
```

```
4×1 cell array
```

```
{'TestClean/classCheck(Data=value1#ext)' }
{'TestClean/sortCheck(Data=value1#ext)' }
{'TestClean/finiteCheck(Data=value1#ext)'}
{'TestClean/noZeroCheck(Data=value1#ext)'}
```

Using the external parameter, the framework creates four suite elements. Since the parameters are defined as a cell array, MATLAB generates the parameter name (`value1`). Also, it appends the characters `#ext` to the end of the parameter name, indicating the parameter is defined externally.

To assign meaningful parameter names (instead of `valueN`), define the parameter using a struct. View the suite element names and run the tests.

```
newData = struct('commandLineData',A);
param = Parameter.fromData('Data',newData);
suite2 = TestSuite.fromClass(?TestClean,'ExternalParameters',param);
{suite2.Name}'
results = suite2.run;
```

```
ans =

4x1 cell array

    {'TestClean/classCheck(Data=commandLineData#ext)'}
    {'TestClean/sortCheck(Data=commandLineData#ext)'}
    {'TestClean/finiteCheck(Data=commandLineData#ext)'}
    {'TestClean/noZeroCheck(Data=commandLineData#ext)'}
```

Running TestClean

```
....
Done TestClean
```

Create another data set that is stored in an ASCII-delimited file.

```
B = rand(3);
B(2,4) = 0;
dlmwrite('myFile.dat',B)
clear B
```

Create a parameter with the stored data set and A.

```
newData = struct('commandLineData',A,'storedData',dlmread('myFile.dat'));
param2 = Parameter.fromData('Data',newData);
suite3 = TestSuite.fromClass(?TestClean,'ExternalParameters',param2);
```

To run the tests using parameters defined in the test file and externally, concatenate test suites. View the suite element names and run the tests.

```
suite = [suite1 suite3];
{suite.Name}'
results = suite.run;
```

```
ans =

16x1 cell array

    {'TestClean/classCheck(Data=clean)'}
    {'TestClean/classCheck(Data=needsCleaning)'}
    {'TestClean/sortCheck(Data=clean)'}
    {'TestClean/sortCheck(Data=needsCleaning)'}
```

```
{ 'TestClean/finiteCheck(Data=clean)' }  
{ 'TestClean/finiteCheck(Data=needsCleaning)' }  
{ 'TestClean/noZeroCheck(Data=clean)' }  
{ 'TestClean/noZeroCheck(Data=needsCleaning)' }  
{ 'TestClean/classCheck(Data=commandLineData#ext)' }  
{ 'TestClean/classCheck(Data=storedData#ext)' }  
{ 'TestClean/sortCheck(Data=commandLineData#ext)' }  
{ 'TestClean/sortCheck(Data=storedData#ext)' }  
{ 'TestClean/finiteCheck(Data=commandLineData#ext)' }  
{ 'TestClean/finiteCheck(Data=storedData#ext)' }  
{ 'TestClean/noZeroCheck(Data=commandLineData#ext)' }  
{ 'TestClean/noZeroCheck(Data=storedData#ext)' }
```

Running TestClean

```
.....  
Done TestClean
```

Running TestClean

```
.....  
Done TestClean
```

See Also

`matlab.unittest.TestSuite` |
`matlab.unittest.parameters.Parameter.fromData`

Related Examples

- “Create Basic Parameterized Test” on page 32-83
- “Create Advanced Parameterized Test” on page 32-89

Create Simple Test Suites

This example shows how to combine tests into test suites, using the SolverTest test case. Use the static from* methods in the matlab.unittest.TestSuite class to create suites for combinations of your tests, whether they are organized in packages and classes or files and folders, or both.

Create Quadratic Solver Function

Create the following function that solves roots of the quadratic equation in a file, quadraticSolver.m, in your working folder.

```
function roots = quadraticSolver(a, b, c)
% quadraticSolver returns solutions to the
% quadratic equation a*x^2 + b*x + c = 0.

if ~isa(a,'numeric') || ~isa(b,'numeric') || ~isa(c,'numeric')
    error('quadraticSolver:InputMustBeNumeric', ...
        'Coefficients must be numeric.');
```

end

```
roots(1) = (-b + sqrt(b^2 - 4*a*c)) / (2*a);
roots(2) = (-b - sqrt(b^2 - 4*a*c)) / (2*a);
```

end

Create Test for Quadratic Solver Function

Create the following test class in a file, SolverTest.m, in your working folder.

```
classdef SolverTest < matlab.unittest.TestCase
    % SolverTest tests solutions to the quadratic equation
    % a*x^2 + b*x + c = 0

    methods (Test)
        function testRealSolution(testCase)
            actSolution = quadraticSolver(1,-3,2);
            expSolution = [2,1];
            testCase.verifyEqual(actSolution,expSolution);
        end
        function testImaginarySolution(testCase)
            actSolution = quadraticSolver(1,2,10);
            expSolution = [-1+3i, -1-3i];
            testCase.verifyEqual(actSolution,expSolution);
        end
    end
end
```

```
        end
    end

end
```

Import TestSuite Class

At the command prompt, add the `matlab.unittest.TestSuite` class to the current import list.

```
import matlab.unittest.TestSuite
```

Make sure the `SolverTest` class definition file is on your MATLAB path.

Create Suite from SolverTest Class

The `fromClass` method creates a suite from all `Test` methods in the `SolverTest` class.

```
suiteClass = TestSuite.fromClass(?SolverTest);
result = run(suiteClass);
```

Create Suite from SolverTest Class Definition File

The `fromFile` method creates a suite using the name of the file to identify the class.

```
suiteFile = TestSuite.fromFile('SolverTest.m');
result = run(suiteFile);
```

Create Suite from All Test Case Files in Current Folder

The `fromFolder` method creates a suite from all test case files in the specified folder. For example, the following files are in the current folder:

- `BankAccountTest.m`
- `DocPolynomTest.m`
- `FigurePropertiesTest.m`
- `IsSupportedTest.m`
- `SolverTest.m`

```
suiteFolder = TestSuite.fromFolder(pwd);
result = run(suiteFolder);
```

Create Suite from Single Test Method

The `fromMethod` method creates a suite from a single test method.

```
suiteMethod = TestSuite.fromMethod(?SolverTest, 'testRealSolution')  
result = run(suiteMethod);
```

See Also

TestSuite

Related Examples

- “Write Simple Test Case Using Classes” on page 32-52

Run Tests for Various Workflows

In this section...

“Set Up Example Tests” on page 32-106
“Run All Tests in Class or Function” on page 32-106
“Run Single Test in Class or Function” on page 32-107
“Run Test Suites by Name” on page 32-108
“Run Test Suites from Test Array” on page 32-108
“Run Tests with Customized Test Runner” on page 32-108

Set Up Example Tests

To explore different ways to run tests, create a class-based test and a function-based test in your current working folder. For the class-based test file use the `DocPolynomTest` example test presented in the `matlab.unittest.qualifications.Verifiable` example. For the function-based test file use the `axesPropertiesTest` example test presented in “Write Test Using Setup and Teardown Functions” on page 32-36.

Run All Tests in Class or Function

Use the `run` method of the `TestCase` class to directly run tests contained in a single test file. When running tests directly, you do not need to explicitly create a `Test` array.

```
% Directly run a single file of class-based tests
results1 = run(DocPolynomTest);
```

```
% Directly run a single file of function-based tests
results2 = run(axesPropertiesTest);
```

You can also assign the test file output to a variable and run the tests using the functional form or dot notation.

```
% Create Test or TestCase objects
t1 = DocPolynomTest;      % TestCase object from class-based test
t2 = axesPropertiesTest;  % Test object from function-based test
```

```
% Run tests using functional form
results1 = run(t1);
```

```
results2 = run(t2);  
  
% Run tests using dot notation  
results1 = t1.run;  
results2 = t2.run;
```

Alternatively, you can run tests contained in a single file by using `runtests` or from the Editor.

Run Single Test in Class or Function

Run a single test from within a class-based test file by specifying the test method as an input argument to the `run` method. For example, only run the test, `testMultiplication`, from the `DocPolynomTest` file.

```
results1 = run(DocPolynomTest, 'testMultiplication');
```

Function-based test files return an array of `Test` objects instead of a single `TestCase` object. You can run a particular test by indexing into the array. However, you must examine the `Name` field in the test array to ensure you run the correct test. For example, only run the test, `surfaceColorTest`, from the `axesPropertiesTest` file.

```
t2 = axesPropertiesTest; % Test object from function-based test  
t2(:).Name
```

```
ans =
```

```
axesPropertiesTest/testDefaultXLim
```

```
ans =
```

```
axesPropertiesTest/surfaceColorTest
```

The `surfaceColorTest` test corresponds to the second element in the array.

Only run the `surfaceColorTest` test.

```
results2 = t2(2).run; % or results2 = run(t2(2));
```

Alternatively, you can run a single test from the Editor.

Run Test Suites by Name

You can run a group, or suite, of tests together. To run the test suite using `runtests`, the suite is defined as a cell array of character vectors representing a test file, a test class, a package that contains tests or a folder that contains tests.

```
suite = {'axesPropertiesTest', 'DocPolynomTest'};  
runtests(suite);
```

Run all tests in the current folder using the `pwd` as input to the `runtests` function.

```
runtests(pwd);
```

Alternatively, you can explicitly create `Test` arrays and use the `run` method to run them.

Run Test Suites from Test Array

You can explicitly create `Test` arrays and use the `run` method in the `TestSuite` class to run them. Using this approach, you explicitly define `TestSuite` objects and, therefore, can examine the contents. The `runtests` function does not return the `TestSuite` object.

```
import matlab.unittest.TestSuite  
s1 = TestSuite.fromClass(?DocPolynomTest);  
s2 = TestSuite.fromFile('axesPropertiesTest.m');  
  
% generate test suite and then run  
fullSuite = [s1 s2];  
result = run(fullSuite);
```

Since the suite is explicitly defined, it is easy for you to perform further analysis on the suite, such as rerunning failed tests.

```
failedTests = fullSuite([result.Failed]);  
result2 = run(failedTests);
```

Run Tests with Customized Test Runner

You can specialize the test running by defining a custom test runner and adding plugins. The `run` method of the `TestRunner` class operates on a `TestSuite` object.

```
import matlab.unittest.TestRunner  
import matlab.unittest.TestSuite
```

```
import matlab.unittest.plugins.TestRunProgressPlugin

% Generate TestSuite.
s1 = TestSuite.fromClass(?DocPolynomTest);
s2 = TestSuite.fromFile('axesPropertiesTest.m');
suite = [s1 s2];

% Create silent test runner.
runner = TestRunner.withNoPlugins;

% Add plugin to display test progress.
runner.addPlugin(TestRunProgressPlugin.withVerbosity(2))

% Run tests using customized runner.
result = run(runner,[suite]);
```

See Also

[matlab.unittest.TestCase.run](#) | [matlab.unittest.TestRunner.run](#) | [matlab.unittest.TestSuite.run](#) | [runtests](#)

More About

- “Run Tests in Editor” on page 32-22

Programmatically Access Test Diagnostics

If you run tests with the `runtests` function or the `run` method of `TestSuite` or `TestCase`, the test framework uses a `DiagnosticsRecordingPlugin` plugin that records diagnostics on test results.

After you run tests, you can access recorded diagnostics via the `DiagnosticRecord` field in the `Details` property on `TestResult`. For example, if your test results are stored in the variable `results`, find the recorded diagnostics for the second test in the suite by invoking `records = result(2).Details.DiagnosticRecord`.

The recorded diagnostics are `DiagnosticRecord` objects. To access particular types of test diagnostics for a particular test, use the `selectFailed`, `selectPassed`, `selectIncomplete`, and `selectLogged` methods of the `DiagnosticRecord` class.

By default, the `DiagnosticsRecordingPlugin` plugin records qualification failures and events logged at a `Terse` level. To configure the plugin to record passing diagnostics or other logged messages at different verbosity levels, configure an instance of `DiagnosticsRecordingPlugin` and add it to the test runner.

See Also

`matlab.unittest.TestResult` |
`matlab.unittest.plugins.DiagnosticsRecordingPlugin` |
`matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord`

Related Examples

- “Add Plugin to Test Runner” on page 32-111

Add Plugin to Test Runner

This example shows how to add a plugin to the test runner. The `matlab.unittest.plugins.TestRunProgressPlugin` displays progress messages about a test case. This plugin is part of the `matlab.unittest` package. MATLAB® uses it for default test runners.

Create a Test for the BankAccount Class

In a file in your working folder, create a test file for the `BankAccount` class.

type `BankAccountTest.m`

```
classdef BankAccountTest < matlab.unittest.TestCase
    % Tests the BankAccount class.

    methods (TestClassSetup)
        function addBankAccountClassToPath(testCase)
            p = path;
            testCase.addTeardown(@path,p);
            addpath(fullfile(matlabroot,'help','techdoc','matlab_oop',...
                'examples'));
        end
    end

    methods (Test)
        function testConstructor(testCase)
            b = BankAccount(1234, 100);
            testCase.verifyEqual(b.AccountNumber, 1234, ...
                'Constructor failed to correctly set account number');
            testCase.verifyEqual(b.AccountBalance, 100, ...
                'Constructor failed to correctly set account balance');
        end

        function testConstructorNotEnoughInputs(testCase)
            import matlab.unittest.constraints.Throws;
            testCase.verifyThat(@()BankAccount, ...
                Throws('MATLAB:minrhs'));
        end

        function testDesposit(testCase)
            b = BankAccount(1234, 100);
            b.deposit(25);
            testCase.verifyEqual(b.AccountBalance, 125);
        end
    end
end
```

```
end

function testWithdraw(testCase)
    b = BankAccount(1234, 100);
    b.withdraw(25);
    testCase.verifyEqual(b.AccountBalance, 75);
end

function testNotifyInsufficientFunds(testCase)
    callbackExecuted = false;
    function testCallback(~,~)
        callbackExecuted = true;
    end

    b = BankAccount(1234, 100);
    b.addlistener('InsufficientFunds', @testCallback);

    b.withdraw(50);
    testCase.assertFalse(callbackExecuted, ...
        'The callback should not have executed yet');
    b.withdraw(60);
    testCase.verifyTrue(callbackExecuted, ...
        'The listener callback should have fired');
end
end
end
```

Create Test Suite

At the command prompt, create a test suite, `ts`, from the `BankAccountTest` test case.

```
ts = matlab.unittest.TestSuite.fromClass(?BankAccountTest);
```

Show Results with No Plugins

Create a test runner with no plugins.

```
runner = matlab.unittest.TestRunner.withNoPlugins;
res = runner.run(ts);
```

No output displayed.

Customize Test Runner

Add the custom plugin, `TestRunProgressPlugin`.

```
import matlab.unittest.plugins.TestRunProgressPlugin
runner.addPlugin(TestRunProgressPlugin.withVerbosity(2))
res = runner.run(ts);
```

```
Running BankAccountTest
```

```
.....
```

```
Done BankAccountTest
```

MATLAB displays progress messages about BankAccountTest.

See Also

`matlab.unittest.plugins`

Write Plugins to Extend TestRunner

In this section...

“Custom Plugins Overview” on page 32-114

“Extending Test Level Plugin Methods” on page 32-115

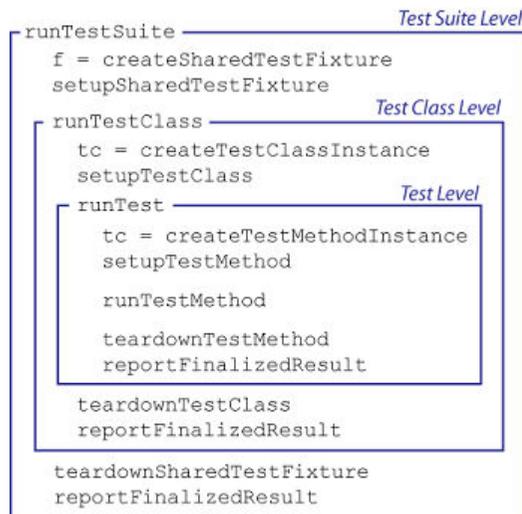
“Extending Test Class Level Plugin Methods” on page 32-115

“Extending Test Suite Level Plugin Methods” on page 32-116

Custom Plugins Overview

TestRunnerPlugin methods have three levels: Test Suite, Test Class, and Test. At each level, you implement methods to extend the creation, setup, run, and teardown of tests or test fixtures. The TestRunner runs these methods as shown in the figure.

Additionally, the `reportFinalizedResult` method enables the test runner to report finalized test results. A test result is finalized when no remaining test content can modify the results. The test runner determines if it invokes the `reportFinalizedResult` method at each level.



The creation methods are the only set of TestRunnerPlugin methods with an output argument. Typically, you extend the creation methods to listen for various events

originating from the test content at the corresponding level. Since both `TestCase` and `Fixture` instances inherit from the `handle` class, you add these listeners using the `addListener` method. The methods that set up, run and tear down test content extend the way the `TestRunner` evaluates the test content.

Extending Test Level Plugin Methods

The `TestRunnerPlugin` methods at the test level extend the creation, setup, run, and teardown of a single test suite element. A single test element consists of one test method or, if the test is parameterized, one instance of the test's parameterization.

Type of Method	Test Level Falls Within Scope of <code>runTest</code>
creation method	<code>createTestMethodInstance</code>
setup method	<code>setupTestMethod</code>
run method	<code>runTestMethod</code>
teardown method	<code>teardownTestMethod</code>

At this level, the `createTestMethodInstance` method is the only plugin method with an output argument. It returns the `TestCase` instances created for each `Test` element. The test framework passes each of these instances into corresponding `Test` methods, and into any methods with the `TestMethodSetup` or `TestMethodTeardown` attribute.

The test framework evaluates methods at the test level within the scope of the `runTest` method. Provided the test framework completes all `TestMethodSetup` work, it invokes the plugin methods in this level a single time per test element.

Extending Test Class Level Plugin Methods

The `TestRunnerPlugin` methods at the test class level extend the creation, setup, run, and teardown of test suite elements that belong to the same test class or the same function-based test. These methods apply to a subset of the full `TestSuite` that the `TestRunner` runs.

Type of Method	Test Class Level Falls Within Scope of <code>runTestClass</code>
creation method	<code>createTestClassInstance</code>
setup method	<code>setupTestClass</code>

Type of Method	Test Class Level Falls Within Scope of runTestClass
run method	runTest
teardown method	teardownTestClass

At this level, the `createTestClassInstance` method is the only plugin method with an output argument. It returns the `TestCase` instances created at the class level. For each class, the test framework passes the instance into any methods with the `TestClassSetup` or `TestClassTeardown` attribute.

A test class setup is parameterized if it contains properties with the `ClassSetupParameter` attribute. In this case, the test framework evaluates the `setupTestClass` and `teardownTestClass` methods as many times as the class setup parameterization dictates.

The run method at this level, `runTest`, extends the running of a single `TestSuite` element, and incorporates the functionality described for the test level plugin methods.

The test framework evaluates methods at the test class level within the scope of the `runTestClass` method. If `TestClassSetup` completes successfully, it invokes the `runTest` method one time for each element in the `Test` array. Each `TestClassSetup` parameterization invokes the creation, setup, and teardown methods a single time.

Extending Test Suite Level Plugin Methods

The `TestRunnerPlugin` methods at the test suite level extend the creation, setup, run, and teardown of shared test fixtures. These methods fall within the scope of `runTestSuite`.

Type of Method	Test Level Falls Within Scope of runTestSuite
creation method	<code>createSharedTestFixture</code>
setup method	<code>setupSharedTestFixture</code>
run method	<code>runTestClass</code>
teardown method	<code>teardownSharedTestFixture</code>

At this level, the `createSharedTestFixture` method is the only plugin method with an output argument. It returns the `Fixture` instances for each shared fixture required by a test class. These fixture instances are available to the test through the `getSharedTestFixtures` method of `TestCase`.

The run method at this level, `runTestClass`, extends the running of tests that belong to the same test class or the same function-based test, and incorporates the functionality described for the test class level plugin methods.

See Also

`addlistener` | `matlab.unittest.TestCase` | `matlab.unittest.TestRunner` |
`matlab.unittest.fixtures.Fixture` |
`matlab.unittest.plugins.OutputStream` |
`matlab.unittest.plugins.TestRunnerPlugin`

Related Examples

- “Create Custom Plugin” on page 32-118
- “Plugin to Generate Custom Test Output Format” on page 32-129
- “Write Plugin to Save Diagnostic Details” on page 32-124

Create Custom Plugin

This example shows how to create a custom plugin that counts the number of passing and failing assertions when running a specified test suite. The plugin prints a brief summary at the end of the testing.

Create AssertionCountingPlugin Class

In a file in your working folder, create a new class, `AssertionCountingPlugin`, that inherits from the `matlab.unittest.plugins.TestRunnerPlugin` class. For a complete version of the code for an `AssertionCountingPlugin`, see "[AssertionCountingPlugin Class Definition Summary](#)".

Keep track of the number of passing and failing assertions. Within a `properties` block, create `NumPassingAssertions` and `NumFailingAssertions` properties to pass the data between methods.

```
properties
    NumPassingAssertions = 0;
    NumFailingAssertions = 0;
end
```

Extend Running of TestSuite

Implement the `runTestSuite` method in a `methods` block with protected access.

```
methods (Access = protected)
    function runTestSuite(plugin, pluginData)
        suiteSize = numel(pluginData.TestSuite);
        fprintf('## Running a total of %d tests\n', suiteSize)

        plugin.NumPassingAssertions = 0;
        plugin.NumFailingAssertions = 0;

        runTestSuite@matlab.unittest.plugins.TestRunnerPlugin(...
            plugin, pluginData);

        fprintf('## Done running tests\n')
        plugin.printAssertionSummary()
    end
end
```

The test framework evaluates this method one time. It displays information about the total number of tests, initializes the assertion count, and invokes the superclass method.

After the framework completes evaluating the superclass method, the `runTestSuite` method displays the assertion count summary.

Extend Creation of Shared Test Fixtures and TestCase Instances

Add listeners to `AssertionPassed` and `AssertionFailed` events to count the assertions. To add these listeners, extend the methods that the test framework uses to create the test content. The test content comprises `TestCase` instances for each `Test` element, class-level `TestCase` instances for the `TestClassSetup` and `TestClassTeardown` methods, and `Fixture` instances that are used when a `TestCase` class has the `SharedTestFixtures` attribute.

Invoke the corresponding superclass method when you override the creation methods. The creation methods return the content that the test framework creates for each of their respective contexts. When implementing one of these methods, pass this argument out of your own implementation, and add the listeners required by this plugin.

Add these creation methods to a methods block with protected access.

```
methods (Access = protected)
    function fixture = createSharedTestFixture(plugin, pluginData)
        fixture = createSharedTestFixture@...
            matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);

        fixture.addListener('AssertionPassed', ...
            @(~,~)plugin.incrementPassingAssertionsCount);
        fixture.addListener('AssertionFailed', ...
            @(~,~)plugin.incrementFailingAssertionsCount);
    end

    function testCase = createTestClassInstance(plugin, pluginData)
        testCase = createTestClassInstance@...
            matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);

        testCase.addListener('AssertionPassed', ...
            @(~,~)plugin.incrementPassingAssertionsCount);
        testCase.addListener('AssertionFailed', ...
            @(~,~)plugin.incrementFailingAssertionsCount);
    end

    function testCase = createTestMethodInstance(plugin, pluginData)
        testCase = createTestMethodInstance@...
            matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);
```

```
        testCase.addListener('AssertionPassed', ...
            @(~,~)plugin.incrementPassingAssertionsCount);
        testCase.addListener('AssertionFailed', ...
            @(~,~)plugin.incrementFailingAssertionsCount);
    end
end
```

Extend Running of Single Test Suite Element

Extend `runTest` to display the name of each test at run time. Include this function in a methods block with protected access. Like all plugin methods, when you override this method you must invoke the corresponding superclass method.

```
methods (Access = protected)
    function runTest(plugin, pluginData)
        fprintf('### Running test: %s\n', pluginData.Name)

        runTest@matlab.unittest.plugins.TestRunnerPlugin(...
            plugin, pluginData);
    end
end
```

Define Helper Functions

In a methods block with private access, define three helper functions. These functions increment the number of passing or failing assertions, and print out the assertion count summary.

```
methods (Access = private)
    function incrementPassingAssertionsCount(plugin)
        plugin.NumPassingAssertions = plugin.NumPassingAssertions + 1;
    end

    function incrementFailingAssertionsCount(plugin)
        plugin.NumFailingAssertions = plugin.NumFailingAssertions + 1;
    end

    function printAssertionSummary(plugin)
        fprintf('%s\n', repmat('_', 1, 30))
        fprintf('Total Assertions: %d\n', ...
            plugin.NumPassingAssertions + plugin.NumFailingAssertions)
        fprintf('\t%d Passed, %d Failed\n', ...
            plugin.NumPassingAssertions, plugin.NumFailingAssertions)
    end
end
```

AssertionCountingPlugin Class Definition Summary

```

classdef AssertionCountingPlugin < ...
    matlab.unittest.plugins.TestRunnerPlugin

    properties
        NumPassingAssertions = 0;
        NumFailingAssertions = 0;
    end

    methods (Access = protected)
        function runTestSuite(plugin, pluginData)
            suiteSize = numel(pluginData.TestSuite);
            fprintf('## Running a total of %d tests\n', suiteSize)

            plugin.NumPassingAssertions = 0;
            plugin.NumFailingAssertions = 0;

            runTestSuite@matlab.unittest.plugins.TestRunnerPlugin(...
                plugin, pluginData);

            fprintf('## Done running tests\n')
            plugin.printAssertionSummary()
        end

        function fixture = createSharedTestFixture(plugin, pluginData)
            fixture = createSharedTestFixture@...
                matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);

            fixture.addListener('AssertionPassed', ...
                @(~,~)plugin.incrementPassingAssertionsCount);
            fixture.addListener('AssertionFailed', ...
                @(~,~)plugin.incrementFailingAssertionsCount);
        end

        function testCase = createTestClassInstance(plugin, pluginData)
            testCase = createTestClassInstance@...
                matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);

            testCase.addListener('AssertionPassed', ...
                @(~,~)plugin.incrementPassingAssertionsCount);
            testCase.addListener('AssertionFailed', ...
                @(~,~)plugin.incrementFailingAssertionsCount);
        end

        function testCase = createTestMethodInstance(plugin, pluginData)
            testCase = createTestMethodInstance@...
                matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);

            testCase.addListener('AssertionPassed', ...
                @(~,~)plugin.incrementPassingAssertionsCount);
            testCase.addListener('AssertionFailed', ...
                @(~,~)plugin.incrementFailingAssertionsCount);
        end
    end
end

```

```
function runTest(plugin, pluginData)
    fprintf('### Running test: %s\n', pluginData.Name)

    runTest@matlab.unittest.plugins.TestRunnerPlugin(...
        plugin, pluginData);
end
end

methods (Access = private)
function incrementPassingAssertionsCount(plugin)
    plugin.NumPassingAssertions = plugin.NumPassingAssertions + 1;
end

function incrementFailingAssertionsCount(plugin)
    plugin.NumFailingAssertions = plugin.NumFailingAssertions + 1;
end

function printAssertionSummary(plugin)
    fprintf('%s\n', repmat('_', 1, 30))
    fprintf('Total Assertions: %d\n', ...
        plugin.NumPassingAssertions + plugin.NumFailingAssertions)
    fprintf('\t%d Passed, %d Failed\n', ...
        plugin.NumPassingAssertions, plugin.NumFailingAssertions)
end
end
end
```

Create Example Test Class

In your working folder, create the file `ExampleTest.m` containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase) % Test fails
            testCase.assertEqual(5, 4)
        end
        function testTwo(testCase) % Test passes
            testCase.verifyEqual(5, 5)
        end
        function testThree(testCase) % Test passes
            testCase.assertEqual(7*2, 14)
        end
    end
end
```

Add Plugin to TestRunner and Run Tests

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestSuite
import matlab.unittest.TestRunner

suite = TestSuite.fromClass(?ExampleTest);
```

Create a test runner with no plugins. This code creates a silent runner and provides you with complete control over the installed plugins.

```
runner = TestRunner.withNoPlugins;
```

Run the tests.

```
result = runner.run(suite);
```

Add `AssertionCountingPlugin` to the runner and run the tests.

```
runner.addPlugin(AssertionCountingPlugin)
result = runner.run(suite);
```

```
## Running a total of 3 tests
### Running test: ExampleTest/testOne
### Running test: ExampleTest/testTwo
### Running test: ExampleTest/testThree
## Done running tests
```

```
Total Assertions: 2
    1 Passed, 1 Failed
```

See Also

`addlistener` | `matlab.unittest.TestCase` | `matlab.unittest.TestRunner` |
`matlab.unittest.fixtures.Fixture` |
`matlab.unittest.plugins.OutputStream` |
`matlab.unittest.plugins.TestRunnerPlugin`

Related Examples

- “Write Plugins to Extend TestRunner” on page 32-114
- “Write Plugin to Save Diagnostic Details” on page 32-124

Write Plugin to Save Diagnostic Details

This example shows how to create a custom plugin to save diagnostic details. The plugin listens for test failures and saves diagnostic information so you can access it after the framework completes the tests.

Create Plugin

In a file in your working folder, create a class, `myPlugin`, that inherits from the `matlab.unittest.plugins.TestRunnerPlugin` class. In the plugin class:

- Define a `FailedTestData` property on the plugin that stores information from failed tests.
- Override the default `createTestMethodInstance` method of `TestRunnerPlugin` to listen for assertion, fatal assertion, and verification failures, and to record relevant information.
- Override the default `runTestSuite` method of `TestRunnerPlugin` to initialize the `FailedTestData` property value. If you do not initialize value of the property, each time you run the tests using the same test runner, failed test information is appended to the `FailedTestData` property.
- Define a helper function, `recordData`, to save information about the test failure as a table.

The plugin saves information contained in the `PluginData` and `QualificationEventData` objects. It also saves the type of failure and timestamp.

```
classdef DiagnosticRecorderPlugin < matlab.unittest.plugins.TestRunnerPlugin
    properties
        FailedTestData
    end

    methods (Access = protected)
        function runTestSuite(plugin, pluginData)
            plugin.FailedTestData = [];
            runTestSuite@...
                matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);
        end

        function testCase = createTestMethodInstance(plugin, pluginData)
            testCase = createTestMethodInstance@...
```

```

        matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);

    testName = pluginData.Name;
    testCase.addlistener('AssertionFailed', ...
        @(~,event)plugin.recordData(event, testName, 'Assertion'));
    testCase.addlistener('FatalAssertionFailed', ...
        @(~,event)plugin.recordData(event, testName, 'Fatal Assertion'));
    testCase.addlistener('VerificationFailed', ...
        @(~,event)plugin.recordData(event, testName, 'Verification'));
    end
end

methods (Access = private)
    function recordData(plugin,eventData,name,failureType)
        s.Name = {name};
        s.Type = {failureType};
        if isempty(eventData.TestDiagnosticResult)
            s.TestDiagnostics = 'TestDiagnostics not provided';
        else
            s.TestDiagnostics = eventData.TestDiagnosticResult;
        end
        s.FrameworkDiagnostics = eventData.FrameworkDiagnosticResult;
        s.Stack = eventData.Stack;
        s.Timestamp = datetime;

        plugin.FailedTestData = [plugin.FailedTestData; struct2table(s)];
    end
end
end

```

Create Test Class

In your working folder, create the file `ExampleTest.m` containing the following test class.

```

classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)
            testCase.assertGreaterThan(5,10)
        end
        function testTwo(testCase)
            wrongAnswer = 'wrong';
            testCase.verifyEmpty(wrongAnswer, 'Not Empty')
            testCase.verifyClass(wrongAnswer, 'double', 'Not double')
        end
    end
end

```

```
function testThree(testCase)
    testCase.assertEqual(7*2,13, 'Values not equal')
end
function testFour(testCase)
    testCase.fatalAssertEqual(3+2,6);
end
end
end
```

The fatal assertion failure in `testFour` causes the framework to halt and throw an error. In this example, there are no subsequent tests. If there was a subsequent test, the framework would not run it.

Add Plugin to Test Runner and Run Tests

At the command prompt, create a test suite from the `ExampleTest` class, and create a test runner.

```
import matlab.unittest.TestSuite
import matlab.unittest.TestRunner

suite = TestSuite.fromClass(?ExampleTest);
runner = TestRunner.withNoPlugins;
```

Create an instance of `myPlugin` and add it to the test runner. Run the tests.

```
p = DiagnosticRecorderPlugin;
runner.addPlugin(p)
result = runner.run(suite);
```

```
Error using ExampleTest/testFour (line 16)
Fatal assertion failed.
```

With the failed fatal assertion, the framework throws an error, and the test runner does not return a `TestResult` object. However, the `DiagnosticRecorderPlugin` stores information about the tests preceding and including the test with the failed assertion.

Inspect Diagnostic Information

At the command prompt, view information about the failed tests. The information is saved in the `FailedTestData` property of the plugin.

```
T = p.FailedTestData
```

```
T =
```

```
5x6 table
```

Name	Type	TestDiagnostics	
'ExampleTest/testOne'	'Assertion'	'TestDiagnostics not provided'	'assertGreaterThan fail
'ExampleTest/testTwo'	'Verification'	'Not Empty'	'verifyEmpty failed.␣
'ExampleTest/testTwo'	'Verification'	'Not double'	'verifyClass failed.␣
'ExampleTest/testThree'	'Assertion'	'Values not equal'	'assertEqual failed.␣
'ExampleTest/testFour'	'Fatal Assertion'	'TestDiagnostics not provided'	'fatalAssertEqual fail

There are many options to archive or post-process this information. For example, you can save the variable as a MAT-file or use `writetable` to write the table to various file types, such as `.txt`, `.csv`, or `.xls`.

View the stack information for the third test failure

```
T.Stack(3)
```

```
ans =
```

```
struct with fields:
```

```
file: 'C:\Work\ExampleTest.m'
name: 'ExampleTest.testTwo'
line: 9
```

Display the diagnostics that the framework displayed for the fifth test failure.

```
celldisp(T.FrameworkDiagnostics(5))
```

```
ans{1} =
```

```
fatalAssertEqual failed.
```

```
--> The values are not equal using "isequaln".
```

```
--> Failure table:
```

Actual	Expected	Error	RelativeError
5	6	-1	-0.166666666666667

```
Actual Value:
```

```
5
```

Expected Value:
6

See Also

`addlistener` | `matlab.unittest.TestCase` | `matlab.unittest.TestRunner` | `matlab.unittest.plugins.TestRunnerPlugin`

Related Examples

- “Write Plugins to Extend TestRunner” on page 32-114
- “Create Custom Plugin” on page 32-118
- “Plugin to Generate Custom Test Output Format” on page 32-129

Plugin to Generate Custom Test Output Format

This example shows how to create a plugin that uses a custom format to write finalized test results to an output stream.

Create Plugin

In a file in your working folder, create a class, `ExampleCustomPlugin`, that inherits from the `matlab.unittest.plugins.TestRunnerPlugin` class. In the plugin class:

- Define a `Stream` property on the plugin that stores the `OutputStream` instance. By default, the plugin writes to standard output.
- Override the default `runTestSuite` method of `TestRunnerPlugin` to output text that indicates the test runner is running a new test session. This information is especially useful if you are writing to a single log file, as it allows you to differentiate the test runs.
- Override the default `reportFinalizedResult` method of `TestRunnerPlugin` to write finalized test results to the output stream. You can modify the `print` method to output the test results in a format that works for your test logs or continuous integration system.

```
classdef ExampleCustomPlugin < matlab.unittest.plugins.TestRunnerPlugin
    properties (Access=private)
        Stream
    end

    methods
        function p = ExampleCustomPlugin(stream)
            if ~nargin
                stream = matlab.unittest.plugins.ToStandardOutput;
            end
            validateattributes(stream,...
                {'matlab.unittest.plugins.OutputStream'},{})
            p.Stream = stream;
        end
    end

    methods (Access=protected)
        function runTestSuite(plugin,pluginData)
            plugin.Stream.print('\n--- NEW TEST SESSION at %s ---\n',...
                char(datetime))
            runTestSuite@...
        end
    end
end
```

```
        matlab.unittest.plugins.TestRunnerPlugin(plugin,pluginData);
    end

    function reportFinalizedResult(plugin,pluginData)
        thisResult = pluginData.TestResult;
        if thisResult.Passed
            status = 'PASSED';
        elseif thisResult.Failed
            status = 'FAILED';
        elseif thisResult.Incomplete
            status = 'SKIPPED';
        end
        plugin.Stream.print(...
            '### YPS Company - Test %s ### - %s in %f seconds.\n',...
            status,thisResult.Name,thisResult.Duration)

        reportFinalizedResult@...
        matlab.unittest.plugins.TestRunnerPlugin(plugin,pluginData)
    end
end
end
```

Create Test Class

In your working folder, create the file `ExampleTest.m` containing the following test class. In this test class, two of the tests pass and the others result in a verification or assumption failure.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)
            testCase.assertGreaterThan(5,1)
        end
        function testTwo(testCase)
            wrongAnswer = 'wrong';
            testCase.verifyEmpty(wrongAnswer, 'Not Empty')
            testCase.verifyClass(wrongAnswer, 'double', 'Not double')
        end
        function testThree(testCase)
            testCase.assumeEqual(7*2,13, 'Values not equal')
        end
        function testFour(testCase)
            testCase.verifyEqual(3+2,5);
        end
    end
end
```

```

    end
end

```

Add Plugin to Test Runner and Run Tests

At the command prompt, create a test suite from the `ExampleTest` class, and create a test runner.

```

import matlab.unittest.TestSuite
import matlab.unittest.TestRunner

suite = TestSuite.fromClass(?ExampleTest);
runner = TestRunner.withNoPlugins;

```

Create an instance of `ExampleCustomPlugin` and add it to the test runner. Run the tests.

```

import matlab.unittest.plugins.ToFile
fname = 'YPS_test_results.txt';
p = ExampleCustomPlugin(ToFile(fname));

runner.addPlugin(p)
result = runner.run(suite);

```

View the contents of the output file.

```
type(fname)
```

```

--- NEW TEST SESSION at 26-Jan-2015 10:41:24 ---
### YPS Company - Test PASSED ### - ExampleTest/testOne in 0.123284 seconds.
### YPS Company - Test FAILED ### - ExampleTest/testTwo in 0.090363 seconds.
### YPS Company - Test SKIPPED ### - ExampleTest/testThree in 0.518044 seconds.
### YPS Company - Test PASSED ### - ExampleTest/testFour in 0.020599 seconds.

```

Rerun the Incomplete tests using the same test runner. View the contents of the output file.

```

suiteFiltered = suite([result.Incomplete]);
result2 = runner.run(suiteFiltered);

```

```
type(fname)
```

```

--- NEW TEST SESSION at 26-Jan-2015 10:41:24 ---

```

```
### YPS Company - Test PASSED ### - ExampleTest/testOne in 0.123284 seconds.  
### YPS Company - Test FAILED ### - ExampleTest/testTwo in 0.090363 seconds.  
### YPS Company - Test SKIPPED ### - ExampleTest/testThree in 0.518044 seconds.  
### YPS Company - Test PASSED ### - ExampleTest/testFour in 0.020599 seconds.  
  
--- NEW TEST SESSION at 26-Jan-2015 10:41:58 ---  
### YPS Company - Test SKIPPED ### - ExampleTest/testThree in 0.007892 seconds.
```

See Also

`ToFile` | `ToStandardOutput` | `matlab.unittest.plugins.OutputStream` |
`matlab.unittest.plugins.TestRunnerPlugin`

Related Examples

- “Write Plugins to Extend TestRunner” on page 32-114
- “Write Plugin to Save Diagnostic Details” on page 32-124

Analyze Test Case Results

This example shows how to analyze the information returned by a test runner created from the SolverTest test case.

Create Quadratic Solver Function

Create the following function that solves roots of the quadratic equation in a file, `quadraticSolver.m`, in your working folder.

```
type quadraticSolver.m

function roots = quadraticSolver(a, b, c)
% quadraticSolver returns solutions to the
% quadratic equation a*x^2 + b*x + c = 0.

if ~isa(a,'numeric') || ~isa(b,'numeric') || ~isa(c,'numeric')
    error('quadraticSolver:InputMustBeNumeric', ...
        'Coefficients must be numeric.');
```

end

```
roots(1) = (-b + sqrt(b^2 - 4*a*c)) / (2*a);
roots(2) = (-b - sqrt(b^2 - 4*a*c)) / (2*a);

end
```

Create Test for Quadratic Solver Function

Create the following test class in a file, `SolverTest.m`, in your working folder.

```
type SolverTest.m

classdef SolverTest < matlab.unittest.TestCase
    % SolverTest tests solutions to the quadratic equation
    % a*x^2 + b*x + c = 0

    methods (Test)
        function testRealSolution(testCase)
            actSolution = quadraticSolver(1,-3,2);
            expSolution = [2,1];
            testCase.verifyEqual(actSolution,expSolution);
        end
        function testImaginarySolution(testCase)
            actSolution = quadraticSolver(1,2,10);
```

```
        expSolution = [-1+3i, -1-3i];
        testCase.verifyEqual(actSolution,expSolution);
    end
end
end
```

Run SolverTest Test Case

Create a test suite, quadTests.

```
quadTests = matlab.unittest.TestSuite.fromClass(?SolverTest);
result = run(quadTests);
```

```
Running SolverTest
..
Done SolverTest
```

—————
All tests passed.

Explore Output Argument, result

The output argument, `result`, is a `matlab.unittest.TestResult` object. It contains information of the two tests in `SolverTest`.

```
whos result
```

Name	Size	Bytes	Class	Attributes
result	1x2	5172	matlab.unittest.TestResult	

Display Information for One Test

To see the information for one value, type:

```
result(1)
```

```
ans =
```

```
TestResult with properties:
```

```
    Name: 'SolverTest/testRealSolution'
   Passed: 1
   Failed: 0
 Incomplete: 0
  Duration: 0.8787
```

```
Details: [1x1 struct]
```

```
Totals:
```

```
  1 Passed, 0 Failed, 0 Incomplete.  
  0.87871 seconds testing time.
```

Create Table of Test Results

To access functionality available to tables, create one from the `TestResult` object.

```
rt = table(result)
```

```
rt=2x6 table
```

Name	Passed	Failed	Incomplete	Duration
'SolverTest/testRealSolution'	true	false	false	0.87871
'SolverTest/testImaginarySolution'	true	false	false	0.022691

Sort the test results by duration.

```
sortrows(rt, 'Duration')
```

```
ans=2x6 table
```

Name	Passed	Failed	Incomplete	Duration
'SolverTest/testImaginarySolution'	true	false	false	0.022691
'SolverTest/testRealSolution'	true	false	false	0.87871

Export test results to a CSV file.

```
writetable(rt, 'myTestResults.csv', 'QuoteStrings', true)
```

See Also

Related Examples

- “Write Simple Test Case Using Classes” on page 32-52

Analyze Failed Test Results

This example shows how to identify and rerun failed tests.

Create an Incorrect Test Method

Using the `SolverTest` test case, add a method, `testBadRealSolution`. This test, based on `testRealSolution`, calls the `quadraticSolver` function with inputs 1, 3, 2, but tests the results against an incorrect solution, [2, 1].

```
function testBadRealSolution(testCase)
    actSolution = quadraticSolver(1,3,2);
    expSolution = [2,1];
    testCase.verifyEqual(actSolution,expSolution)
end
```

Run New Test Suite

Save the updated `SolverTest` class definition and rerun the tests.

```
quadTests = matlab.unittest.TestSuite.fromClass(?SolverTest);
result1 = run(quadTests);
```

```
Running SolverTest
```

```
..
```

```
=====
Verification failed in SolverTest/testBadRealSolution.
```

```
-----
Framework Diagnostic:
-----
verifyEqual failed.
--> The values are not equal using "isequaln".
--> Failure table:
      Index   Actual   Expected   Error   RelativeError
      -----
      1       -1        2         -3      -1.5
      2       -2        1         -3       -3

Actual Value:
      -1    -2
Expected Value:
      2     1
```

```
-----
Stack Information:
-----
```

```
In C:\work\SolverTest.m (SolverTest.testBadRealSolution) at 19
=====
```

```
.
Done SolverTest
-----
```

Failure Summary:

Name	Failed	Incomplete	Reason(s)
SolverTest/testBadRealSolution	X		Failed by verification.

Analyze Results

The output tells you `SolverTest/testBadRealSolution` failed. From the Framework Diagnostic you see the following:

```
Actual Value:
    -1  -2
Expected Value:
    2   1
```

At this point, you must decide if the error is in `quadraticSolver` or in your value for `expSolution`.

Correct Error

Edit the value of `expSolution` in `testBadRealSolution`:

```
expSolution = [-1 -2];
```

Rerun Tests

Save `SolverTest` and rerun only the failed tests.

```
failedTests = quadTests([result1.Failed]);
result2 = run(failedTests)
```

```
Running SolverTest
```

```
.
Done SolverTest
```

```
result2 =
```

```
  TestResult with properties:
```

```
      Name: 'SolverTest/testBadRealSolution'  
      Passed: 1  
      Failed: 0  
      Incomplete: 0  
      Duration: 0.0108  
      Details: [1x1 struct]
```

```
Totals:
```

```
  1 Passed, 0 Failed, 0 Incomplete.  
  0.010813 seconds testing time.
```

Alternatively, you can rerun failed tests using the ([rerun](#)) link in the test results.

See Also

More About

- “Rerun Failed Tests” on page 32-139

Rerun Failed Tests

If a test failure is caused by incorrect or incomplete code, it is useful to rerun failed tests quickly and conveniently. When you run a test suite, the test results include information about the test suite and the test runner. If there are test failures in the results, when MATLAB displays the test results there is a link to rerun the failed tests.

Totals:

```
1 Passed, 1 Failed (rerun), 0 Incomplete.
0.25382 seconds testing time.
```

This link allows you to modify your test code or your code under test and quickly rerun failed tests. However, if you make structural changes to your test class, using the rerun link does not pick up the changes. Structural changes include adding, deleting, or renaming a test method, and modifying a test parameter property and its value. In this case, recreate the entire test suite to pick up the changes.

Create the following function in your current working folder. The function is meant to compute the square and square root. However, in this example, the function computes the cube of the value instead of the square.

```
function [x,y] = exampleFunction(n)
    validateattributes(n,{'numeric'},{'scalar'})

    x = n^3;      % square (incorrect code, should be n^2)
    y = sqrt(n); % square root
end
```

Create the following test in a file `exampleTest.m`.

```
function tests = exampleTest
    tests = functiontests(localfunctions);
end

function testSquare(testCase)
    [sqrVal,sqrRootVal] = exampleFunction(3);
    verifyEqual(testCase,sqrVal,9);
end

function testSquareRoot(testCase)
    [sqrVal,sqrRootVal] = exampleFunction(100);
    verifyEqual(testCase,sqrRootVal,10);
end
```

Create a test suite and run the tests. The `testSquare` test fails because the implementation of `exampleFunction` is incorrect.

```
suite = testsuite('ExampleTest.m');
results = run(suite)
```

Running exampleTest

```
=====
Verification failed in exampleTest/testSquare.

-----
Framework Diagnostic:
-----
verifyEqual failed.
--> The values are not equal using "isequaln".
--> Failure table:
      Actual   Expected   Error   RelativeError
      -----   -
          27         9        18         2

Actual Value:
    27
Expected Value:
    9

-----
Stack Information:
-----
In C:\Work\exampleTest.m (testSquare) at 7
=====
..
Done exampleTest

-----
Failure Summary:

      Name                Failed  Incomplete  Reason(s)
      =====
exampleTest/testSquare    X                               Failed by verification.

results =

1x2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
    1 Passed, 1 Failed (rerun), 0 Incomplete.
    0.24851 seconds testing time.
```

Update the code in `exampleFunction` to fix the coding error.

```
function [x,y] = exampleFunction(n)
    validateattributes(n,{'numeric'},{'scalar'})

    x = n^2;    % square
    y = sqrt(n); % square root
end
```

Click the (rerun) link in the command window to rerun the failed test. You cannot rerun failed tests if the variable that stores the test results is overwritten. If the link is no longer in the Command Window, you can type `results` at the prompt to view it.

```
Running exampleTest
```

```
.
Done exampleTest
```

```
ans =
```

```
    TestResult with properties:
```

```
        Name: 'exampleTest/testSquare'
        Passed: 1
        Failed: 0
    Incomplete: 0
    Duration: 0.0034
    Details: [1x1 struct]
```

```
Totals:
```

```
    1 Passed, 0 Failed, 0 Incomplete.
    0.0033903 seconds testing time.
```

MATLAB stores the `TestResult` array associated with tests that you rerun in the `ans` variable. `results` is a 1x2 array that contains all the tests in `exampleTest.m`, and `ans` is a 1x1 array that contains the rerun results from the one failed test.

```
whos
```

Name	Size	Bytes	Class	Attributes
ans	1x1	664	matlab.unittest.TestResult	
results	1x2	1344	matlab.unittest.TestResult	
suite	1x2	96	matlab.unittest.Test	

To programmatically rerun failed tests, use the `Failed` property on the `TestResult` object to create and run a filtered test suite.

```
failedTests = suite([results.Failed]);  
result2 = run(failedTests);
```

```
Running exampleTest
```

```
.
```

```
Done exampleTest
```

To ensure that all passing tests continue to pass, rerun the full test suite.

See Also

More About

- “Analyze Failed Test Results” on page 32-136

Dynamically Filtered Tests

In this section...

“Test Methods” on page 32-143

“Method Setup and Teardown Code” on page 32-146

“Class Setup and Teardown Code” on page 32-148

Assumption failures produce filtered tests. In the `matlab.unittest.TestResult` class, such a test is marked `Incomplete`.

Since filtering test content through the use of assumptions does not produce test failures, it has the possibility of creating dead test code. Avoiding this requires monitoring of filtered tests.

Test Methods

If an assumption failure is encountered inside of a `TestCase` method with the `Test` attribute, the entire method is marked as filtered, but MATLAB runs the subsequent `Test` methods.

The following class contains an assumption failure in one of the methods in the `Test` block.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testA(testCase)
            testCase.verifyTrue(true)
        end
        function testB(testCase)
            testCase.assertEqual(0,1)
            % remaining test code is not exercised
        end
        function testC(testCase)
            testCase.verifyFalse(true)
        end
    end
end
```

Since the `testB` method contains an assumption failure, when you run the test, the testing framework filters that test and marks it as incomplete. After the assumption

failure in `testB`, the testing framework proceeds and executes `testC`, which contains a verification failure.

```
ts = matlab.unittest.TestSuite.fromClass(?ExampleTest);  
res = ts.run;
```

```
Running ExampleTest
```

```
.  
=====  
ExampleTest/testB was filtered.  
Details  
=====  
.   
=====  
Verification failed in ExampleTest/testC.  
  
-----  
Framework Diagnostic:  
-----  
verifyFalse failed.  
--> The value must evaluate to "false".  
  
Actual logical:  
    1  
  
-----  
Stack Information:  
-----  
In C:\work\ExampleTest.m (ExampleTest.testC) at 11  
=====  
.   
Done ExampleTest
```

```
-----  
Failure Summary:
```

Name	Failed	Incomplete	Reason(s)
ExampleTest/testB		X	Filtered by assumption.
ExampleTest/testC	X		Failed by verification.

If you examine the `TestResult`, you notice that there is a passed test, a failed test, and a test that did not complete due to an assumption failure.

```
res
res =
  1x3 TestResult array with properties:
    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
  1 Passed, 1 Failed, 1 Incomplete.
  2.4807 seconds testing time.
```

The testing framework keeps track of incomplete tests so that you can monitor filtered tests for nonexercised test code. You can see information about these tests within the `TestResult` object.

```
res([res.Incomplete])
ans =
  TestResult with properties:
    Name: 'ExampleTest/testB'
    Passed: 0
    Failed: 0
    Incomplete: 1
    Duration: 2.2578
    Details: [1x1 struct]

Totals:
  0 Passed, 0 Failed, 1 Incomplete.
  2.2578 seconds testing time.
```

To create a modified test suite from only the filtered tests, select incomplete tests from the original test suite.

```
tsFiltered = ts([res.Incomplete])
tsFiltered =
```

```
Test with properties:
    Name: 'ExampleTest/testB'
    ProcedureName: 'testB'
    TestClass: "ExampleTest"
    BaseFolder: 'C:\work'
    Parameterization: [0x0 matlab.unittest.parameters.EmptyParameter]
    SharedTestFixtures: [0x0 matlab.unittest.fixtures.EmptyFixture]
    Tags: {1x0 cell}
```

```
Tests Include:
    0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Method Setup and Teardown Code

If an assumption failure is encountered inside a `TestCase` method with the `TestMethodSetup` attribute, MATLAB filters the method which was to be run for that instance. If a test uses assumptions from within the `TestMethodSetup` block, consider instead using the assumptions in the `TestClassSetup` block, which likewise filters all `Test` methods in the class but is less verbose and more efficient.

One of the methods in the following `TestMethodSetup` block within `ExampleTest.m` contains an assumption failure.

```
methods(TestMethodSetup)
    function setupMethod1(testCase)
        testCase.assumeEqual(1,0)
        % remaining test code is not exercised
    end
    function setupMethod2(testCase)
        disp('* Running setupMethod2 *')
        testCase.assertEqual(1,1)
    end
end
```

Updated ExampleTest Class Definition

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(TestMethodSetup)
        function setupMethod1(testCase)
            testCase.assumeEqual(1,0)
            % remaining test code is not exercised
        end
        function setupMethod2(testCase)
```

```

        disp('* Running setupMethod2 *')
        testCase.assertEqual(1,1)
    end
end

methods(Test)
function testA(testCase)
    testCase.verifyTrue(true)
end
function testB(testCase)
    testCase.assumeEqual(0,1)
    % remaining test code is not exercised
end
function testC(testCase)
    testCase.verifyFalse(true)
end
end
end
end

```

When you run the test, you see that the framework completes executes all the methods in the TestMethodSetup block that do not contain the assumption failure, and it marks as incomplete all methods in the Test block.

```

ts = matlab.unittest.TestSuite.fromClass(?ExampleTest);
res = ts.run;

```

Running ExampleTest

```

=====
ExampleTest/testA was filtered.
  Details

```

```

=====
* Running setupMethod2 *
.

```

```

=====
ExampleTest/testB was filtered.
  Details

```

```

=====
* Running setupMethod2 *
.

```

```

=====
ExampleTest/testC was filtered.
  Details

```

```

=====
* Running setupMethod2 *

```

Done ExampleTest

Failure Summary:

Name	Failed	Incomplete	Reason(s)
ExampleTest/testA		X	Filtered by assumption.
ExampleTest/testB		X	Filtered by assumption.
ExampleTest/testC		X	Filtered by assumption.

The Test methods did not change but all 3 are filtered due to an assumption failure in the TestMethodSetup block. The testing framework executes methods in the TestMethodSetup block without assumption failures, such as `setupMethod2`. As expected, the testing framework executes `setupMethod2` 3 times, once before each Test method.

Class Setup and Teardown Code

If an assumption failure is encountered inside of a TestCase method with the TestClassSetup or TestClassTeardown attribute, MATLAB filters the entire TestCase class.

The methods in the following TestClassSetup block within `ExampleTest.m` contains an assumption failure.

```
methods(TestClassSetup)
    function setupClass(testCase)
        testCase.assumeEqual(1,0)
        % remaining test code is not exercised
    end
end
```

Updated ExampleTest Class Definition

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(TestClassSetup)
        function setupClass(testCase)
            testCase.assumeEqual(1,0)
            % remaining test code is not exercised
        end
    end
end
```

```

        end
    end

    methods(TestMethodSetup)
        function setupMethod1(testCase)
            testCase.assertEqual(1,0)
            % remaining test code is not exercised
        end
        function setupMethod2(testCase)
            disp('* Running setupMethod2 *')
            testCase.assertEqual(1,1)
        end
    end

    methods(Test)
        function testA(testCase)
            testCase.verifyTrue(true)
        end
        function testB(testCase)
            testCase.assertEqual(0,1)
            % remaining test code is not exercised
        end
        function testC(testCase)
            testCase.verifyFalse(true)
        end
    end
end
end

```

When you run the test, you see that the framework does not execute any of the methods in the `TestMethodSetup` or `Test`.

```
ts = matlab.unittest.TestSuite.fromClass(?ExampleTest);
res = ts.run;
```

Running ExampleTest

```
=====
All tests in ExampleTest were filtered.
  Details
=====
```

Done ExampleTest

Failure Summary:

Name	Failed	Incomplete	Reason(s)
ExampleTest/testA		X	Filtered by assumption.
ExampleTest/testB		X	Filtered by assumption.
ExampleTest/testC		X	Filtered by assumption.

The `Test` and `TestMethodSetup` methods did not change but everything is filtered due to an assumption failure in the `TestClassSetup` block.

See Also

`TestCase` | `TestResult` | `matlab.unittest.qualifications.Assumable`

Create Custom Constraint

This example shows how to create a custom constraint that determines if a given value is the same size as an expected value.

In a file in your working folder, create a `HasSameSizeAs.m`. The constructor accepts a value to compare to the actual size. This value is stored within the `ValueWithExpectedSize` property. Since, it is recommended that `Constraint` implementations are immutable, set the property `SetAccess=immutable`.

```
classdef HasSameSizeAs < matlab.unittest.constraints.Constraint
    properties(SetAccess=immutable)
        ValueWithExpectedSize
    end

    methods
        function constraint = HasSameSizeAs(value)
            constraint.ValueWithExpectedSize = value;
        end
    end
end
```

Classes that derive from `Constraint` must implement the `satisfiedBy` method. This method must contain the comparison logic and return a boolean value.

Include the `satisfiedBy` method in the `methods` block in `HasSameSizeAs.m`.

```
function bool = satisfiedBy(constraint, actual)
    bool = isequal(size(actual), size(constraint.ValueWithExpectedSize));
end
```

If the actual size and expected size are equal, this method returns `true`.

Classes deriving from `Constraint` must implement the `getDiagnosticFor` method. This method must evaluate the actual value against the constraint and provide a `Diagnostic` object. In this example, `getDiagnosticFor` returns a `StringDiagnostic`. Include the `getDiagnosticFor` method in the `methods` block in `HasSameSizeAs.m`.

```
function diag = getDiagnosticFor(constraint, actual)
    import matlab.unittest.diagnostics.StringDiagnostic
```

```

        if constraint.satisfiedBy(actual)
            diag = StringDiagnostic('HasSameSizeAs passed.');
```

```

        else
            diag = StringDiagnostic(sprintf(...
                'HasSameSizeAs failed.\nActual Size: [%s]\nExpectedSize: [%s]',...
                int2str(size(actual)),...
                int2str(size(constraint.ValueWithExpectedSize))));
        end
    end
end

```

HasSameSizeAs Class Definition Summary

```

classdef HasSameSizeAs < matlab.unittest.constraints.Constraint

    properties(SetAccess=immutable)
        ValueWithExpectedSize
    end

    methods
        function constraint = HasSameSizeAs(value)
            constraint.ValueWithExpectedSize = value;
        end
        function bool = satisfiedBy(constraint, actual)
            bool = isequal(size(actual), size(constraint.ValueWithExpectedSize));
        end
        function diag = getDiagnosticFor(constraint, actual)
            import matlab.unittest.diagnostics.StringDiagnostic

            if constraint.satisfiedBy(actual)
                diag = StringDiagnostic('HasSameSizeAs passed.');
```

```

            else
                diag = StringDiagnostic(sprintf(...
                    'HasSameSizeAs failed.\nActual Size: [%s]\nExpectedSize: [%s]',...
                    int2str(size(actual)),...
                    int2str(size(constraint.ValueWithExpectedSize))));
            end
        end
    end
end
end

```

At the command prompt, create a test case for interactive testing.

```
import matlab.unittest.TestCase
```

```
testCase = TestCase.forInteractiveUse;
```

Test a passing case.

```
testCase.verifyThat(zeros(5), HasSameSizeAs(repmat(1,5)))
```

```
Interactive verification passed.
```

Test a failing case.

```
testCase.verifyThat(zeros(5), HasSameSizeAs(ones(1,5)))
```

```
Interactive verification failed.
```

```
-----  
Framework Diagnostic:  
-----
```

```
HasSameSizeAs failed.
```

```
Actual Size: [5 5]
```

```
ExpectedSize: [1 5]
```

See Also

`matlab.unittest.constraints.Constraint`

Related Examples

- “Create Custom Boolean Constraint” on page 32-154

Create Custom Boolean Constraint

This example shows how to create a custom boolean constraint that determines if a given value is the same size as an expected value.

In a file in your working folder, create a file `HasSameSizeAs.m`. The constructor accepts a value to compare to the actual size. This value is stored within the `ValueWithExpectedSize` property. It is recommended that `BooleanConstraint` implementations be immutable, so set the property `SetAccess=immutable`.

```
classdef HasSameSizeAs < matlab.unittest.constraints.BooleanConstraint
    properties(SetAccess=immutable)
        ValueWithExpectedSize
    end

    methods
        function constraint = HasSameSizeAs(value)
            constraint.ValueWithExpectedSize = value;
        end
    end
end
```

Include these methods in the `methods` block in `HasSameSizeAs.m`. Since the `BooleanConstraint` class is a subclass of `Constraint`, classes that derive from it must implement the `satisfiedBy` and `getDiagnosticFor` methods. For more information about these methods, see `matlab.unittest.constraints.Constraint`.

```
methods
    function bool = satisfiedBy(constraint, actual)
        bool = isequal(size(actual), size(constraint.ValueWithExpectedSize));
    end
    function diag = getDiagnosticFor(constraint, actual)
        import matlab.unittest.diagnostics.StringDiagnostic

        if constraint.satisfiedBy(actual)
            diag = StringDiagnostic('HasSameSizeAs passed.');
```

```
        else
            diag = StringDiagnostic(sprintf(...
                'HasSameSizeAs failed.\nActual Size: [%s]\nExpectedSize: [%s]',...
                int2str(size(actual)),...
                int2str(size(constraint.ValueWithExpectedSize))));
        end
    end
end
```

Include the `getNegativeDiagnosticFor` method in the `methods` block with protected access in `HasSameSizeAs.m`. Classes that derive from `BooleanConstraint` must implement the `getNegativeDiagnosticFor` method. This method must provide a `Diagnostic` object that is expressed in the negative sense of the constraint.

```
methods(Access=protected)
    function diag = getNegativeDiagnosticFor(constraint, actual)
```

```

import matlab.unittest.diagnostics.StringDiagnostic

if constraint.satisfiedBy(actual)
    diag = StringDiagnostic(sprintf(...
        ['Negated HasSameSizeAs failed.\nSize [%s] of ' ...
        'Actual Value and Expected Value were the same ' ...
        'but should not have been.'], int2str(size(actual))));
else
    diag = StringDiagnostic('Negated HasSameSizeAs passed.');
```

In exchange for implementing the required methods, the constraint inherits the appropriate and, or, and not overloads so it can be combined with other BooleanConstraint objects or negated.

HasSameSizeAs Class Definition Summary

```

classdef HasSameSizeAs < matlab.unittest.constraints.BooleanConstraint
    properties(SetAccess=immutable)
        ValueWithExpectedSize
    end
    methods
        function constraint = HasSameSizeAs(value)
            constraint.ValueWithExpectedSize = value;
        end
        function bool = satisfiedBy(constraint, actual)
            bool = isequal(size(actual), size(constraint.ValueWithExpectedSize));
        end
        function diag = getDiagnosticFor(constraint, actual)
            import matlab.unittest.diagnostics.StringDiagnostic

            if constraint.satisfiedBy(actual)
                diag = StringDiagnostic('HasSameSizeAs passed.');
```

At the command prompt, create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasLength
```

```
testCase = TestCase.forInteractiveUse;
```

Test a passing case.

```
testCase.verifyThat(zeros(5), HasLength(5) | ~HasSameSizeAs(repmat(1,5)))
```

```
Interactive verification passed.
```

The test passes because one of the or conditions, `HasLength(5)`, is true.

Test a failing case.

```
testCase.verifyThat(zeros(5), HasLength(5) & ~HasSameSizeAs(repmat(1,5)))
```

```
Interactive verification failed.
```

```
-----
Framework Diagnostic:
-----
```

```
AndConstraint failed.
```

```
--> + [First Condition]:
```

```
    |   HasLength passed.
```

```
--> AND
```

```
    + [Second Condition]:
```

```
    |   Negated HasSameSizeAs failed.
```

```
    |   Size [5 5] of Actual Value and Expected Value were the same but should not ha
```

```
    -+-----
```

The test fails because one of the and conditions, `~HasSameSizeAs(repmat(1,5))`, is false.

See Also

`matlab.unittest.constraints.BooleanConstraint`

Related Examples

- “Create Custom Constraint” on page 32-151

Create Custom Tolerance

This example shows how to create a custom tolerance to determine if two DNA sequences have a Hamming distance within a specified tolerance. For two DNA sequences of the same length, the Hamming distance is the number of positions in which the nucleotides (letters) of one sequence differ from the other.

In a file, `DNA.m`, in your working folder, create a simple class for a DNA sequence.

```
classdef DNA
    properties (SetAccess=immutable)
        Sequence
    end

    methods
        function dna = DNA(sequence)
            validLetters = ...
                sequence == 'A' | ...
                sequence == 'C' | ...
                sequence == 'T' | ...
                sequence == 'G';

            if ~all(validLetters(:))
                error('Sequence contained a letter not found in DNA.')
            end
            dna.Sequence = sequence;
        end
    end
end
```

In a file in your working folder, create a tolerance class so that you can test that DNA sequences are within a specified Hamming distance. The constructor requires a `Value` property that defines the maximum Hamming distance.

```
classdef HammingDistance < matlab.unittest.constraints.Tolerance
    properties
        Value
    end

    methods
        function tolerance = HammingDistance(value)
            tolerance.Value = value;
        end
    end
end
```

```
end  
end
```

In a `methods` block with the `HammingDistance` class definition, include the following method so that the tolerance supports DNA objects. Tolerance classes must implement a `supports` method.

```
methods  
    function tf = supports(~, value)  
        tf = isa(value, 'DNA');  
    end  
end
```

In a `methods` block with the `HammingDistance` class definition, include the following method that returns `true` or `false`. Tolerance classes must implement a `satisfiedBy` method. The testing framework uses this method to determine if two values are within the tolerance.

```
methods  
    function tf = satisfiedBy(tolerance, actual, expected)  
        if ~isSameSize(actual.Sequence, expected.Sequence)  
            tf = false;  
            return  
        end  
        tf = hammingDistance(actual.Sequence, expected.Sequence) <= tolerance.Value;  
    end  
end
```

In the `HammingDistance.m` file, define the following helper functions outside of the `classdef` block. The `isSameSize` function returns `true` if two DNA sequences are the same size, and the `hammingDistance` function returns the Hamming distance between two sequences.

```
function tf = isSameSize(str1, str2)  
tf = isequal(size(str1), size(str2));  
end  
  
function distance = hammingDistance(str1, str2)  
distance = nnz(str1 ~= str2);  
end
```

The function returns a `Diagnostic` object with information about the comparison. In a `methods` block with the `HammingDistance` class definition, include the following method

that returns a `StringDiagnostic`. Tolerance classes must implement a `getDiagnosticFor` method.

```

methods
    function diag = getDiagnosticFor(tolerance, actual, expected)
        import matlab.unittest.diagnostics.StringDiagnostic

        if ~isSameSize(actual.Sequence, expected.Sequence)
            str = 'The DNA sequences must be the same length.';
        else
            str = sprintf('%s%d.\n%s%d.', ...
                'The DNA sequences have a Hamming distance of ', ...
                hammingDistance(actual.Sequence, expected.Sequence), ...
                'The allowable distance is ', ...
                tolerance.Value);
        end
        diag = StringDiagnostic(str);
    end
end

```

HammingDistance Class Definition Summary

```

classdef HammingDistance < matlab.unittest.constraints.Tolerance
    properties
        Value
    end

    methods
        function tolerance = HammingDistance(value)
            tolerance.Value = value;
        end

        function tf = supports(~, value)
            tf = isa(value, 'DNA');
        end

        function tf = satisfiedBy(tolerance, actual, expected)
            if ~isSameSize(actual.Sequence, expected.Sequence)
                tf = false;
            return
            end
            tf = hammingDistance(actual.Sequence, expected.Sequence) <= tolerance.Value;
        end

        function diag = getDiagnosticFor(tolerance, actual, expected)

```

```
import matlab.unittest.diagnostics.StringDiagnostic

if ~isSameSize(actual.Sequence, expected.Sequence)
    str = 'The DNA sequences must be the same length.';
else
    str = sprintf('%s%d.\n%s%d.', ...
        'The DNA sequences have a Hamming distance of ', ...
        hammingDistance(actual.Sequence, expected.Sequence), ...
        'The allowable distance is ', ...
        tolerance.Value);
end
diag = StringDiagnostic(str);
end
end
end

function tf = isSameSize(str1, str2)
tf = isequal(size(str1), size(str2));
end

function distance = hammingDistance(str1, str2)
distance = nnz(str1 ~= str2);
end
```

At the command prompt, create a TestCase for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
```

```
testCase = TestCase.forInteractiveUse;
```

Create two DNA objects.

```
sampleA = DNA('ACCTGAGTA');
sampleB = DNA('ACCACAGTA');
```

Verify that the DNA sequences are equal to each other.

```
testCase.verifyThat(sampleA, IsEqualTo(sampleB))
```

```
Interactive verification failed.
```

```
-----
Framework Diagnostic:
-----
```

```

IsEqualTo failed.
--> ObjectComparator failed.
    --> The objects are not equal using "isequal".

```

```

Actual Object:
    DNA with properties:
        Sequence: 'ACCTGAGTA'
Expected Object:
    DNA with properties:
        Sequence: 'ACCACAGTA'

```

Verify that the DNA sequences are equal to each other within a Hamming distance of 1.

```

testCase.verifyThat(sampleA, IsEqualTo(sampleB, ...
    'Within', HammingDistance(1)))

```

Interactive verification failed.

```

-----
Framework Diagnostic:
-----
IsEqualTo failed.
--> ObjectComparator failed.
    --> The objects are not equal using "isequal".
    --> The DNA sequences have a Hamming distance of 2.
        The allowable distance is 1.

```

```

Actual Object:
    DNA with properties:
        Sequence: 'ACCTGAGTA'
Expected Object:
    DNA with properties:
        Sequence: 'ACCACAGTA'

```

The sequences are not equal to each other within a tolerance of 1. The testing framework displays additional diagnostics from the `getDiagnosticFor` method.

Verify that the DNA sequences are equal to each other within a Hamming distance of 2.

```

testCase.verifyThat(sampleA, IsEqualTo(sampleB, ...
    'Within', HammingDistance(2)))

```

Interactive verification passed.

See Also

`matlab.unittest.constraints.Tolerance`

Overview of App Testing Framework

Use the MATLAB app testing framework to test App Designer apps, or apps built programmatically using the `uifigure` function. The app testing framework lets you author a test class that programmatically performs a gesture on a UI component, such as pressing a button or dragging a slider, and verifies the behavior of the app.

App Testing

Test Creation - Class-based tests can use the app testing framework by subclassing `matlab.uitest.TestCase`. Because `matlab.uitest.TestCase` is a subclass of `matlab.unittest.TestCase`, your test has access to the features of the unit testing framework, such as qualifications, fixtures, and plugins. To experiment with the app testing framework at the command prompt, create a test case instance using `matlab.uitest.TestCase.forInteractiveUse`.

Test Content - Typically, a test of an app programmatically interacts with app components using a gesture method of `matlab.uitest.TestCase`, such as `press` or `drag`, and then performs a qualification on the result. For example, a test might press one check box and verify that the other check boxes are disabled. Or it might type a number into a text box and verify the app correctly computes a result. These types of tests require understanding of the properties of the app being tested. To verify a button press, you must know where in the app object MATLAB stores the status of a button. To verify the result of a computation, you must know how to access the result within the app.

Test Clean Up - It is a best practice to include a tear-down action to delete the app after the test. Typically, the test method adds this action using the `addTeardown` method of `matlab.unittest.TestCase`.

App Locking - When an app test creates a figure, the framework locks the figure immediately to prevent external interactions with the components. The app testing framework does not lock UI components if you create an instance of `matlab.uitest.TestCase.forInteractiveUse` for experimentation at the command prompt.

To unlock a figure for debugging purposes, use the `matlab.uitest.unlock` function.

Gesture Support of UI Components

The gesture methods of `matlab.uitest.TestCase` support various UI components.

Component	matlab.uitest.TestCase Gesture Meth		
	press	choose	drag
Button	✓		
State button	✓	✓	
Check box	✓	✓	
Switch	✓	✓	
Discrete knob		✓	
Knob		✓	✓
Drop-down		✓	
Edit field			
Text area			
Spinner	✓		
Slider		✓	✓
List box		✓	
Button group		✓	
Tab group		✓	
Tab		✓	
Tree node		✓	
Menu	✓		
Axes	✓		

Write a Test for an App

This example shows how to write a test for an app that provides options to change the sample size and colormap of a plot. To programmatically interact with the app and qualify the results, use the app testing framework combined with the unit testing framework.

To explore the properties of this app prior to testing, create an instance of the app at the command prompt.

```
app = ConfigurePlotAppExample;
```

This step is not necessary for the tests, but it is helpful to explore the properties used by the app tests. For example, use `app.UpdatePlotButton` to access the **Update Plot** button within the app object.

Create a test class that inherits from `matlab.uitest.TestCase`.

```
classdef testConfigurePlotAppExample < matlab.uitest.TestCase
    methods (Test)
    end
end
```

Create a test method `test_SampleSize` to test the sample size. The test method modifies the sample size, updates the plot, and verifies that the surface uses the specified sample size. The call to `addTeardown` deletes the app after the test is complete.

```
classdef testConfigurePlotAppExample < matlab.uitest.TestCase
    methods (Test)
        function test_SampleSize(testCase)
            app = ConfigurePlotAppExample;
            testCase.addTeardown(@delete, app);

            testCase.type(app.SampleSizeEditField, 12);
            testCase.press(app.UpdatePlotButton);

            ax = app.UIAxes;
            surfaceObj = ax.Children;
            testCase.verifySize(surfaceObj.ZData, [12 12]);
        end
    end
end
```

Create a second test method `test_Colormap` to test the colormap. The test method selects a colormap, updates the plot, and verifies that the plot uses the specified colormap. The full code is now as follows.

```
classdef testConfigurePlotAppExample < matlab.uitest.TestCase
    methods (Test)
        function test_SampleSize(testCase)
```

```
        app = ConfigurePlotAppExample;
        testCase.addTeardown(@delete, app);

        testCase.type(app.SampleSizeEditField, 12);
        testCase.press(app.UpdatePlotButton);

        ax = app.UIAxes;
        surfaceObj = ax.Children;
        testCase.verifySize(surfaceObj.ZData, [12 12]);
    end

    function test_Colormap(testCase)
        app = ConfigurePlotAppExample;
        testCase.addTeardown(@delete, app);

        testCase.choose(app.ColormapDropDown, 'Winter');
        testCase.press(app.UpdatePlotButton);

        expectedMap = winter;
        ax = app.UIAxes;
        testCase.verifyEqual(ax.Colormap, expectedMap);
    end
end

end
```

At the command prompt, run the tests.

```
results = runtests('testConfigurePlotAppExample')
```

```
Running testConfigurePlotAppExample
```

```
..
```

```
Done testConfigurePlotAppExample
```

```
results =
```

```
1x2 TestResult array with properties:
```

```
    Name
    Passed
    Failed
    Incomplete
    Duration
```

Details

Totals:

2 Passed, 0 Failed, 0 Incomplete.
4.7551 seconds testing time.

See Also

`matlab.uitest.TestCase`

More About

- “Write Test for App” on page 32-168
- “Write Test That Uses App Testing and Mocking Frameworks” on page 32-173

Write Test for App

This example shows how to write tests for an App Designer app. To interact with the app programmatically and qualify the results, use the app testing framework and the unit testing framework.

To explore the properties of this app prior to testing, create an instance of the app at the command prompt.

```
app = PatientsDisplay;
```

This step is not necessary for the tests, but it is helpful to explore the properties used by the app tests. For example, use `app.BloodPressureSwitch` to access the **Blood Pressure** switch within the app object.

Create a test class that inherits from `matlab.uiunit.TestCase`. To test the tab switching functionality, create a test method `test_tab`. The test method chooses the **Data** tab and then verifies that the selected tab has the correct title. The `TestMethodSetup` method creates an app for each test and deletes it after the test is complete.

```
classdef TestPatientsDisplay < matlab.uiunit.TestCase
    properties
        App
    end

    methods (TestMethodSetup)
        function launchApp(testCase)
            testCase.App = PatientsDisplay;
            testCase.addTeardown(@delete, testCase.App);
        end
    end

    methods (Test)
        function test_tab(testCase)
            % Choose Data Tab
            dataTab = testCase.App.Tab2;
            testCase.choose(dataTab)

            % Verify Data Tab is selected
            testCase.verifyEqual(testCase.App.TabGroup.SelectedTab.Title, 'Data')
        end
    end
end
```

Create a `test_plottingOptions` method that tests various plotting options. The test method presses the **Histogram** radio button and verifies that the x-label changed. Then, it changes the **Bin Width** slider and verifies the number of bins.

```
classdef TestPatientsDisplay < matlab.uiunit.TestCase
    properties
```

```

        App
    end

    methods (TestMethodSetup)
        function launchApp(testCase)
            testCase.App = PatientsDisplay;
            testCase.addTeardown(@delete, testCase.App);
        end
    end

    methods (Test)
        function test_plottingOptions(testCase)
            % Press the histogram radio button
            testCase.press(testCase.App.HistogramButton)

            % Verify xlabel updated from 'Weight' to 'Systolic'
            testCase.verifyEqual(testCase.App.UIAxes.XLabel.String, 'Systolic')

            % Change the Bin Width to 9
            testCase.choose(testCase.App.BinWidthSlider, 9)

            % Verify the number of bins is now 4
            testCase.verifyEqual(testCase.App.UIAxes.Children.NumBins, 4)
        end

        function test_tab(testCase) ...
    end
end

```

Create a `test_bloodPressure` method that tests blood pressure data and display. The test method verifies the y-axis label and the values of the scatter points. Then it changes to Diastolic readings, and verifies the label and data again.

```

classdef TestPatientsDisplay < matlab.uitest.TestCase
    properties
        App
    end

    methods (TestMethodSetup)
        function launchApp(testCase)
            testCase.App = PatientsDisplay;
            testCase.addTeardown(@delete, testCase.App);
        end
    end

    methods (Test)
        function test_bloodPressure(testCase)
            % Extract blood pressure data from app
            t = cell2table(testCase.App.Tab2.Children.Data);
            t.Var2 = categorical(t.Var2);
            allMales = t(t.Var2=='Male',:);
            maleDiastolicData = allMales.Var7';
            maleSystolicData = allMales.Var8';

            % Verify ylabel and that male Systolic data shows
            ax = testCase.App.UIAxes;
            testCase.verifyEqual(ax.YLabel.String, 'Systolic')
            testCase.verifyEqual(ax.Children.YData, maleSystolicData)

            % Switch to 'Diastolic' reading
        end
    end
end

```

```

        testCase.choose(testCase.App.BloodPressureSwitch, 'Diastolic')

        % Verify ylabel changed and male Diastolic data shows
        testCase.verifyEqual(ax.YLabel.String, 'Diastolic')
        testCase.verifyEqual(ax.Children.YData, maleDiastolicData);
    end

    function test_plottingOptions(testCase) ...

    function test_tab(testCase) ...

end
end
end

```

Create a `test_gender` method that tests gender data and display. The test method verifies the number of male scatter points and then presses the check box to include female data. It verifies that two data sets are plotted and the color of the female data is red. Finally, it clears the male data check box and verifies the number of plotted data sets and scatter points. This test fails because there are 53 female scatter points instead of 50. To take a screen shot when the test fails, use a `ScreenshotDiagnostic` with the `onFailure` method.

```

classdef TestPatientsDisplay < matlab.uiunitest.TestCase
    properties
        App
    end

    methods (TestMethodSetup)
        function launchApp(testCase)
            testCase.App = PatientsDisplay;
            testCase.addTeardown(@delete, testCase.App);
        end
    end

    methods (Test)
        function test_gender(testCase)
            import matlab.unittest.diagnostics.ScreenshotDiagnostic
            testCase.onFailure(ScreenshotDiagnostic);

            % Verify 47 male scatter points
            ax = testCase.App.UIAxes;
            testCase.verifyNumElements(ax.Children.XData, 47);

            % Enable the checkbox for female data
            testCase.choose(testCase.App.FemaleCheckBox);

            % Verify two data sets display and the female data is red
            testCase.assertNumElements(ax.Children, 2);
            testCase.verifyEqual(ax.Children(1).CData, [1 0 0]);

            % Disable the male data
            testCase.choose(testCase.App.MaleCheckBox, false);

            % Verify one data set displays and number of scatter points
            testCase.verifyNumElements(ax.Children, 1);
            testCase.verifyNumElements(ax.Children.XData, 50);
        end

        function test_bloodPressure(testCase)

```

```

% Extract blood pressure data from app
t = cell2table(testCase.App.Tab2.Children.Data);
t.Var2 = categorical(t.Var2);
allMales = t(t.Var2=='Male',:);
maleDiastolicData = allMales.Var7';
maleSystolicData = allMales.Var8';

% Verify ylabel and that male Systolic data shows
ax = testCase.App.UIAxes;
testCase.verifyEqual(ax.YLabel.String,'Systolic')
testCase.verifyEqual(ax.Children.YData,maleSystolicData)

% Switch to 'Diastolic' reading
testCase.choose(testCase.App.BloodPressureSwitch,'Diastolic')

% Verify ylabel changed and male Diastolic data shows
testCase.verifyEqual(ax.YLabel.String,'Diastolic')
testCase.verifyEqual(ax.Children.YData,maleDiastolicData);
end

function test_plottingOptions(testCase)
% Press the histogram radio button
testCase.press(testCase.App.HistogramButton)

% Verify xlabel updated from 'Weight' to 'Systolic'
testCase.verifyEqual(testCase.App.UIAxes.XLabel.String,'Systolic')

% Change the Bin Width to 9
testCase.choose(testCase.App.BinWidthSlider,9)

% Verify the number of bins is now 4
testCase.verifyEqual(testCase.App.UIAxes.Children.NumBins,4)
end

function test_tab(testCase)
% Choose Data Tab
dataTab = testCase.App.Tab2;
testCase.choose(dataTab)

% Verify Data Tab is selected
testCase.verifyEqual(testCase.App.TabGroup.SelectedTab.Title,'Data')
end

end
end

```

Run the tests.

```
results = runtests('TestPatientsDisplay');
```

Running TestPatientsDisplay

```
=====
Verification failed in TestPatientsDisplay/test_gender.
```

```

-----
Framework Diagnostic:
-----
verifyNumElements failed.
--> The value did not have the correct number of elements.
```

```
Actual Number of Elements:
    53
Expected Number of Elements:
    50

Actual Value:
Columns 1 through 49

    131    133    119    142    142    132    128    137    129    131    133    117    137    146    123    143    114

Columns 50 through 53

    141    129    124    134

-----
Additional Diagnostic:
-----
Screenshot captured to:
--> C:\Temp\54fd8dc0-0637-4926-9c4f-f217fe195fe1\Screenshot_daba8870-adb3-4a1c-ba11-df3d9b51d36f.png

-----
Stack Information:
-----
In C:\Work\TestPatientsDisplay.m (TestPatientsDisplay.test_gender) at 34
=====
....
Done TestPatientsDisplay
-----

Failure Summary:

Name                                     Failed Incomplete Reason(s)
-----
TestPatientsDisplay/test_gender          X                                     Failed by verification.
```

See Also

matlab.uitest.TestCase

More About

- “Overview of App Testing Framework” on page 32-163
- “Write Test That Uses App Testing and Mocking Frameworks” on page 32-173

Write Test That Uses App Testing and Mocking Frameworks

This example shows how to write a test that uses the app testing framework and the mocking framework. The app contains a file selection dialog box and a label indicating the selected file. To test the app programmatically, use a mock object to define the behavior of the file selector.

Create App

Create the `launchApp` app in your current working folder. The app allows a user to select an input file and displays the name of the file in the app. The file selection dialog box is a blocking modal dialog box that waits for user input.

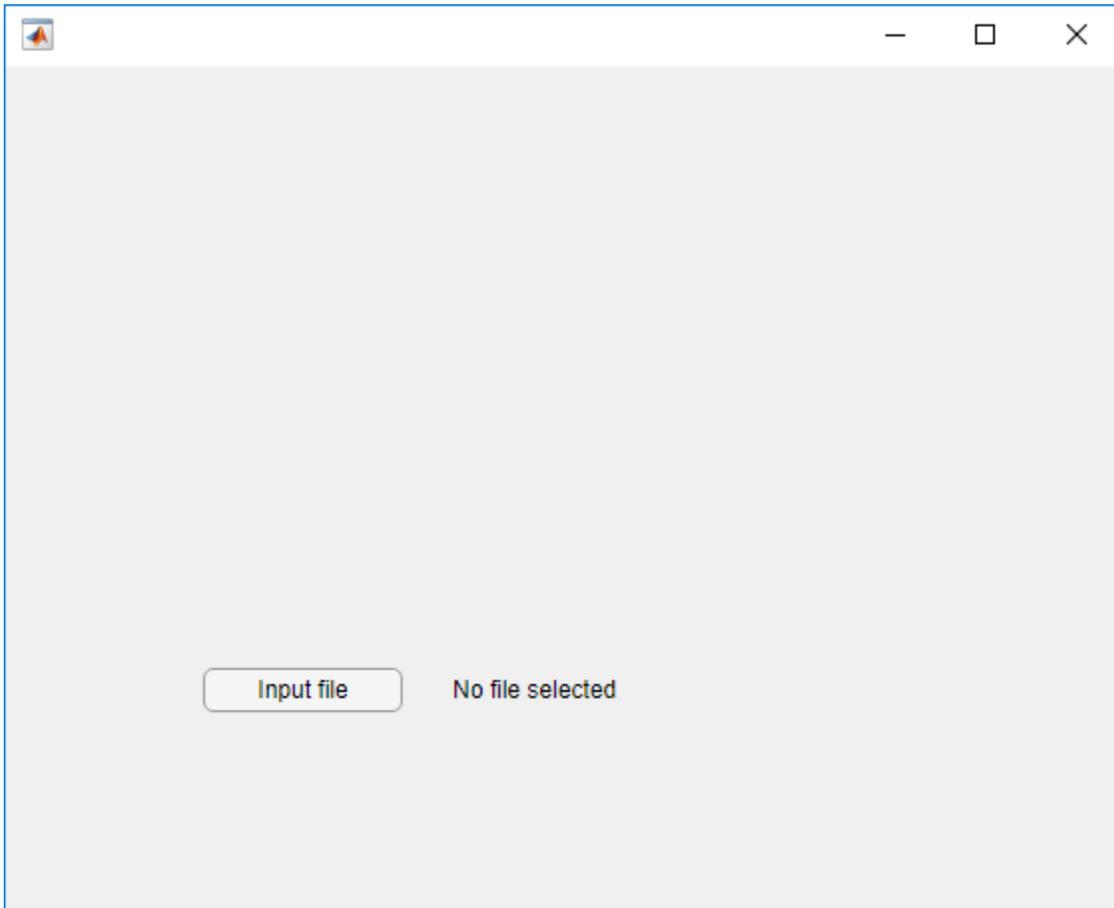
```
function app = launchApp
    f = uifigure;
    button = uibutton(f,'Text','Input file');
    button.ButtonPushedFcn = @(src,evt)pickFile;
    label = uilabel(f,'Text','No file selected');
    label.Position(1) = button.Position(1) + button.Position(3) + 25;
    label.Position(3) = 200;

    % Add components to an App struct for output
    app.UIFigure = f;
    app.Button = button;
    app.Label = label;

    function file = pickFile()
        [file,folder,status] = uigetfile('*.');
        if status
            label.Text = file;
        end
    end
end
```

To explore the properties of this app prior to testing, create an instance of the app at the command prompt. This step is not necessary for the tests, but it is helpful to explore the properties used by the app tests. For example, use `app.Button` to access the **Input file** button within the app object.

```
app = launchApp;
```



Test App With Manual Intervention

Create the `LaunchAppTest` class without using mocks. The test assumes the file `input2.txt` exists in your current working folder. If it does not exist, create it. The test presses the **Input file** button programmatically and verifies that the label matches `'input2.txt'`. You must manually select the file.

```
classdef LaunchAppTest < matlab.uitest.TestCase
    properties
        TestFile = 'input2.txt';
    end
```

```

methods(TestClassSetup)
    function checkTestFiles(tc)
        import matlab.unittest.constraints.IsFile
        tc.assumeThat(tc.TestFile,IsFile)
    end
end
methods (Test)
    function testInputButton(tc)
        app = launchApp;
        tc.addTeardown(@close,app.UIFigure);

        tc.press(app.Button);

        tc.verifyEqual(app.Label.Text,tc.TestFile)
    end
end
end

```

Run the test. When the file selection dialog box appears, select `input2.txt` to allow MATLAB to proceed with the test. Selecting any other file results in a test failure.

```
results = runtests('LaunchAppTest');
```

```
Running LaunchAppTest
```

```
Done LaunchAppTest
```

Create Fully Automated Test

To test the app without manual intervention, use the mocking framework. Modify the app to accept a file-choosing service instead of implementing it in the app (dependency injection).

Create a `FileChooser` service with an `Abstract` method that implements the file selection functionality.

```

classdef FileChooser
    % Interface to choose a file
    methods (Abstract)
        [file,folder,status] = chooseFile(chooser,varargin)
    end
end

```

Create a default FileChooser that uses the `uigetfile` function for file selection.

```
classdef DefaultFileChooser < FileChooser
    methods
        function [file, folder, status] = chooseFile(chooser, varargin)
            [file, folder, status] = uigetfile(varargin{:});
        end
    end
end
```

Change the app to accept an optional FileChooser object. When called with no inputs, the app uses an instance of DefaultFileChooser.

```
function app = launchApp(fileChooser)
    if nargin==0
        fileChooser = DefaultFileChooser;
    end
    f = uifigure;
    button = uibutton(f, 'Text', 'Input file');
    button.ButtonPushedFcn = @(src, evt)pickFile(fileChooser);
    label = uilabel(f, 'Text', 'No file selected');
    label.Position(1) = button.Position(1) + button.Position(3) + 25;
    label.Position(3) = 200;

    % Add components to an App struct for output
    app.UIFigure = f;
    app.Button = button;
    app.Label = label;

    function file = pickFile(fileChooser)
        [file, folder, status] = fileChooser.chooseFile('*.');
        if status
            label.Text = file;
        end
    end
end
```

Make the following modifications to `LaunchAppTest`.

- Change the test to inherit from both `matlab.uitest.TestCase` and `matlab.mock.TestCase`.
- Remove the properties block and the `TestClassSetup` block. Because the mock defines the output of the `chooseFile` method call, the test does not rely on the existence of an external file.

- Change the `testInputButton` test method so that it will do these things.
 - Create a mock object of the `FileChooser`.
 - Define mock behavior such that when the `chooseFile` method is called with the input `'*.*'`, the outputs are the test file name (`'input2.txt'`), the current working folder, and a selected filter index of 1. These outputs are analogous to the outputs from the `uigetfile` function.
 - Press the button and verify the selected file name. These steps are the same as in the original test, but the mock assigns the output values, so you do not need to interact with the app to continue testing.
- To test the **Cancel** button, add a test method `testInputButton_Cancel` so that it will do these things.
 - Create a mock object of the `FileChooser`.
 - Define mock behavior such that when the `chooseFile` method is called with the input `'*.*'`, the outputs are the test file name (`'input2.txt'`), the current working folder, and a selected filter index of 0. These outputs are analogous to the outputs from the `uigetfile` function if a user selects a file and then chooses to cancel.
 - Press the button and verify that the test calls the `chooseFile` method and that the label indicates that no file was selected.

```

classdef LaunchAppTest < matlab.uitest.TestCase & matlab.mock.TestCase
    methods (Test)
        function testInputButton(tc)
            import matlab.mock.actions.AssignOutputs
            fname = 'myFile.txt';

            [mockChooser,behavior] = tc.createMock(?FileChooser);
            when(behavior.chooseFile('*.*'),AssignOutputs(fname,pwd,1))

            app = launchApp(mockChooser);
            tc.addTeardown(@close,app.UIFigure);

            tc.press(app.Button);

            tc.verifyEqual(app.Label.Text,fname);
        end

        function testInputButton_Cancel(tc)
            import matlab.mock.actions.AssignOutputs

            [mockChooser,behavior] = tc.createMock(?FileChooser);
            when(behavior.chooseFile('*.*'),AssignOutputs('myFile.txt',pwd,0))

            app = launchApp(mockChooser);
            tc.addTeardown(@close,app.UIFigure);

            tc.press(app.Button);
    end
end

```

```
        tc.verifyCalled(behavior.chooseFile('*.*'));
        tc.verifyEqual(app.Label.Text, 'No file selected');
    end
end
end
```

Run the tests. The tests run to completion without manual file selection.

```
results = runtests('LaunchAppTest');
```

```
Running LaunchAppTest
..
Done LaunchAppTest
```

See Also

`matlab.mock.TestCase` | `matlab.uitest.TestCase`

More About

- “Overview of App Testing Framework” on page 32-163
- “Write Test for App” on page 32-168
- “Create Mock Object” on page 32-202

Overview of Performance Testing Framework

In this section...

“Determine Bounds of Measured Code” on page 32-179

“Types of Time Experiments” on page 32-180

“Write Performance Tests with Measurement Boundaries” on page 32-181

“Run Performance Tests” on page 32-182

“Understand Invalid Test Results” on page 32-182

The performance test interface leverages the script, function, and class-based unit testing interfaces. You can perform qualifications within your performance tests to ensure correct functional behavior while measuring code performance. Also, you can run your performance tests as standard regression tests to ensure that code changes do not break performance tests.

Determine Bounds of Measured Code

This table indicates what code is measured for the different types of tests.

Type of Test	What Is Measured	What Is Excluded
Script-based	Code in each section of the script	<ul style="list-style-type: none"> Code in the shared variables section Measured estimate of the framework overhead
Function-based	Code in each test function	<ul style="list-style-type: none"> Code in the following functions: <code>setup</code>, <code>setupOnce</code>, <code>teardown</code>, and <code>teardownOnce</code> Measured estimate of the framework overhead

Type of Test	What Is Measured	What Is Excluded
Class-based	Code in each method tagged with the <code>Test</code> attribute	<ul style="list-style-type: none"> Code in the methods with the following attributes: <code>TestMethodSetup</code>, <code>TestMethodTeardown</code>, <code>TestClassSetup</code>, and <code>TestClassTeardown</code> Shared fixture setup and teardown Measured estimate of the framework overhead
Class-based deriving from <code>matlab.perftest.TestCase</code> and using <code>startMeasuring</code> and <code>stopMeasuring</code> methods	Code between calls to <code>startMeasuring</code> and <code>stopMeasuring</code> in each method tagged with the <code>Test</code> attribute	<ul style="list-style-type: none"> Code outside of the <code>startMeasuring/stopMeasuring</code> boundary Measured estimate of the framework overhead
Class-based deriving from <code>matlab.perftest.TestCase</code> and using the <code>keepMeasuring</code> method	Code inside each <code>keepMeasuring-while</code> loop in each method tagged with the <code>Test</code> attribute	<ul style="list-style-type: none"> Code outside of the <code>keepMeasuring-while</code> boundary Measured estimate of the framework overhead

Types of Time Experiments

You can create two types of time experiments.

- A frequentist time experiment collects a variable number of measurements to achieve a specified margin of error and confidence level. Use a frequentist time experiment to define statistical objectives for your measurement samples. Generate this experiment using the `runperf` function or the `limitingSamplingError` static method of the `TimeExperiment` class.
- A fixed time experiment collects a fixed number of measurements. Use a fixed time experiment to measure first-time costs of your code or to take explicit control of your sample size. Generate this experiment using the `withFixedSampleSize` static method of the `TimeExperiment` class.

This table summarizes the differences between the frequentist and fixed time experiments.

	Frequentist time experiment	Fixed time experiment
Warm-up measurements	4 by default, but configurable through <code>TimeExperiment.limitingSamplingError</code>	0 by default, but configurable through <code>TimeExperiment.withFixedSampleSize</code>
Number of samples	Between 4 and 32 by default, but configurable through <code>TimeExperiment.limitingSamplingError</code>	Defined during experiment construction
Relative margin of error	5% by default, but configurable through <code>TimeExperiment.limitingSamplingError</code>	Not applicable
Confidence level	95% by default, but configurable through <code>TimeExperiment.limitingSamplingError</code>	Not applicable
Framework behavior for invalid test result	Stops measuring a test and moves to the next one	Collects specified number of samples

Write Performance Tests with Measurement Boundaries

If your class-based tests derive from `matlab.perftest.TestCase` instead of `matlab.unittest.TestCase`, then you can use the `startMeasuring` and `stopMeasuring` methods or the `keepMeasuring` method multiple times to define boundaries for performance test measurements. If a test method has multiple calls to `startMeasuring`, `stopMeasuring` and `keepMeasuring`, then the performance framework accumulates and sums the measurements. The performance framework does not support nested measurement boundaries. If you use these methods incorrectly in a `Test` method and run the test as a `TimeExperiment`, then the framework marks the measurement as invalid. Also, you still can run these performance tests as unit tests. For more information, see “Test Performance Using Classes” on page 32-190.

Run Performance Tests

There are two ways to run performance tests:

- Use the `runperf` function to run the tests. This function uses a variable number of measurements to reach a sample mean with a 0.05 relative margin of error within a 0.95 confidence level. It runs the tests four times to warm up the code and between 4 and 32 times to collect measurements that meet the statistical objectives.
- Generate an explicit test suite using the `testsuite` function or the methods in the `TestSuite` class, and then create and run a time experiment.
 - Use the `withFixedSampleSize` method of the `TimeExperiment` class to construct a time experiment with a fixed number of measurements. You can specify a fixed number of warm-up measurements and a fixed number of samples.
 - Use the `limitingSamplingError` method of the `TimeExperiment` class to construct a time experiment with specified statistical objectives, such as margin of error and confidence level. Also, you can specify the number of warm-up measurements and the minimum and maximum number of samples.

You can run your performance tests as regression tests. For more information, see “Run Tests for Various Workflows” on page 32-106.

Understand Invalid Test Results

In some situations, the `MeasurementResult` for a test result is marked invalid. A test result is marked invalid when the performance testing framework sets the `Valid` property of the `MeasurementResult` to false. This invalidation occurs if your test fails or is filtered. Also, if your test incorrectly uses the `startMeasuring` and `stopMeasuring` methods of `matlab.perftest.TestCase`, then the `MeasurementResult` for that test is marked invalid.

When the performance testing framework encounters an invalid test result, it behaves differently depending on the type of time experiment:

- If you create a frequentist time experiment, then the framework stops measuring for that test and moves to the next test.
- If you create a fixed time experiment, then the framework continues collecting the specified number of samples.

See Also

`matlab.perftest.TimeExperiment` |
`matlab.unittest.measurement.MeasurementResult` | `runperf` | `testsuite`

Related Examples

- “Test Performance Using Scripts or Functions” on page 32-184
- “Test Performance Using Classes” on page 32-190

Test Performance Using Scripts or Functions

This example shows how to create a script or function-based performance test that times the preallocation of a vector using four different approaches.

Write Performance Test

Create a performance test in a file, `preallocationTest.m`, in your current working folder. In this example, you can choose to use either the following script-based test or the function-based test. The output in this example is for the function-based test. If you use the script-based test, then your test names will be different.

Script-Based Performance Test	Function-Based Performance Test
<pre> vectorSize = 1e7; %% Ones Function x = ones(1,vectorSize); %% Indexing With Variable id = 1:vectorSize; x(id) = 1; %% Indexing On LHS x(1:vectorSize) = 1; %% For Loop for i=1:vectorSize x(i) = 1; end </pre>	<pre> function tests = preallocationTest tests = functiontests(localfunctions); end function testOnes(testCase) vectorSize = getSize(); x = ones(1,vectorSize()); end function testIndexingWithVariable(testCase) vectorSize = getSize(); id = 1:vectorSize; x(id) = 1; end function testIndexingOnLHS(testCase) vectorSize = getSize(); x(1:vectorSize) = 1; end function testForLoop(testCase) vectorSize = getSize(); for i=1:vectorSize x(i) = 1; end end function vectorSize = getSize() vectorSize = 1e7; end </pre>

Run Performance Test

Run the performance test. Depending on your system, the warnings you see might vary. In this example output, the performance testing framework ran the `preallocationTest/testOnes` test the maximum number of times, but it did not achieve a 0.05 relative margin of error with a 0.95 confidence level.

```
results = runperf('preallocationTest.m')
```

```
Running preallocationTest
```

```
.....
.....
```

```
.....  
.....Warning: The target Relative Margin of Error was not met after running the MaxSa  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
Done preallocationTest
```

```
results =
```

```
1x4 MeasurementResult array with properties:
```

```
    Name  
    Valid  
    Samples  
    TestActivity
```

```
Totals:
```

```
4 Valid, 0 Invalid.
```

The `results` variable is a `1x4 MeasurementResult` array. Each element in the array corresponds to one of the tests defined in the code section in `preallocationTest.m`.

Display Test Results

Display the measurement results for the second test. Your results might vary.

```
results(2)
```

```
ans =
```

```
MeasurementResult with properties:
```

```
    Name: 'preallocationTest/testIndexingWithVariable'  
    Valid: 1  
    Samples: [17x7 table]  
    TestActivity: [21x12 table]
```

```
Totals:
```

```
1 Valid, 0 Invalid.
```

As indicated by the size of the `TestActivity` property, the performance testing framework collected 21 measurements. This number of measurements includes four measurements to warm up the code. The `Samples` property excludes warm-up measurements.

Display the sample measurements for the second test.

```
results(2).Samples
```

```
ans =
```

Name	MeasuredTime	Timestamp
preallocationTest/testIndexingWithVariable	0.12496	31-Dec-2015 06:29:38
preallocationTest/testIndexingWithVariable	0.16411	31-Dec-2015 06:29:39
preallocationTest/testIndexingWithVariable	0.13467	31-Dec-2015 06:29:39
preallocationTest/testIndexingWithVariable	0.14919	31-Dec-2015 06:29:39
preallocationTest/testIndexingWithVariable	0.13663	31-Dec-2015 06:29:39
preallocationTest/testIndexingWithVariable	0.12597	31-Dec-2015 06:29:39
preallocationTest/testIndexingWithVariable	0.13036	31-Dec-2015 06:29:39
preallocationTest/testIndexingWithVariable	0.17423	31-Dec-2015 06:29:40
preallocationTest/testIndexingWithVariable	0.13087	31-Dec-2015 06:29:40
preallocationTest/testIndexingWithVariable	0.13951	31-Dec-2015 06:29:40
preallocationTest/testIndexingWithVariable	0.12493	31-Dec-2015 06:29:40
preallocationTest/testIndexingWithVariable	0.12613	31-Dec-2015 06:29:40
preallocationTest/testIndexingWithVariable	0.15276	31-Dec-2015 06:29:40
preallocationTest/testIndexingWithVariable	0.16414	31-Dec-2015 06:29:41
preallocationTest/testIndexingWithVariable	0.13791	31-Dec-2015 06:29:41
preallocationTest/testIndexingWithVariable	0.12533	31-Dec-2015 06:29:41
preallocationTest/testIndexingWithVariable	0.12339	31-Dec-2015 06:29:41

Compute Statistics for Single Test Element

Display the mean measured time for the second test. To exclude data collected in the warm-up runs, use the values in the `Samples` field.

```
sampleTimes = results(2).Samples.MeasuredTime;
meanTest2 = mean(sampleTimes)
```

```
meanTest2 =
```

```
0.1391
```

The performance testing framework collected 17 sample measurements for the second test. The test took an average of 0.1391 second.

Compute Statistics for All Test Elements

Determine the average time for all the test elements. The `preallocationTest` test includes four different methods for allocating a vector of ones. Compare the time for each method (test element).

Since the performance testing framework returns a `Samples` table for each test element, concatenate all these tables into one table. Then group the rows by test element `Name`, and compute the mean `MeasuredTime` for each group.

```
fullTable = vertcat(results.Samples);
summaryStats = varfun(@mean,fullTable,...
    'InputVariables','MeasuredTime','GroupingVariables','Name')
```

```
summaryStats =
```

Name	GroupCount	mean_MeasuredTime
preallocationTest/testOnes	32	0.031445
preallocationTest/testIndexingWithVariable	17	0.13912
preallocationTest/testIndexingOnLHS	23	0.071286
preallocationTest/testForLoop	4	0.80677

Recall that the performance testing framework issued a warning stating that the measurements for the `preallocationTest/testOnes` test did not meet the statistical objectives. The testing framework collected the maximum number of samples, which is 32, and then it stopped the test. By contrast, the measurements for the `preallocationTest/testForLoop` test met statistical objectives in the minimum number of samples, which is four.

Change Statistical Objectives and Rerun Tests

Change the statistical objectives defined by the `runperf` function by constructing and running a time experiment. Construct a time experiment with measurements that reach a sample mean with an 8% relative margin of error within a 97% confidence level.

Construct an explicit test suite.

```
suite = testsuite('preallocationTest');
```

Construct a time experiment with a variable number of sample measurements, and run the tests.

```
import matlab.perftest.TimeExperiment
experiment = TimeExperiment.limitingSamplingError('NumWarmups',2,...
'RelativeMarginOfError',0.08, 'ConfidenceLevel', 0.97);
resultsTE = run(experiment,suite);
```

```
Running preallocationTest
.....
.....
.....
Done preallocationTest
```

Compute the statistics for all the test elements.

```
fullTableTE = vertcat(resultsTE.Samples);
summaryStatsTE = varfun(@mean,fullTableTE,...
'InputVariables','MeasuredTime','GroupingVariables','Name')
summaryStatsTE =
```

Name	GroupCount	mean_MeasuredTime
preallocationTest/testOnes	4	0.025568
preallocationTest/testIndexingWithVariable	6	0.12898
preallocationTest/testIndexingOnLHS	5	0.066603
preallocationTest/testForLoop	4	0.78484

See Also

matlab.perftest.TimeExperiment |
matlab.unittest.measurement.MeasurementResult | runperf | testsuite

Test Performance Using Classes

This example shows how to create a performance test and regression test for the `fprintf` function.

Write Performance Test

Consider the following unit (regression) test. You can run this test as a performance test using `runperf('fprintfTest')` instead of `runtests('fprintfTest')`.

```
classdef fprintfTest < matlab.unittest.TestCase
    properties
        file
        fid
    end
    methods(TestMethodSetup)
        function openFile(testCase)
            testCase.file = tempname;
            testCase.fid = fopen(testCase.file, 'w');
            testCase.assertNotEqual(testCase.fid, -1, 'IO Problem')

            testCase.addTeardown(@delete, testCase.file);
            testCase.addTeardown(@fclose, testCase.fid);
        end
    end

    methods(Test)
        function testPrintingToFile(testCase)
            textToWrite = repmat('abcdef', 1, 5000000);
            fprintf(testCase.fid, '%s', textToWrite);
            testCase.verifyEqual(fileread(testCase.file), textToWrite)
        end

        function testBytesToFile(testCase)
            textToWrite = repmat('tests_', 1, 5000000);
            nbytes = fprintf(testCase.fid, '%s', textToWrite);
            testCase.verifyEqual(nbytes, length(textToWrite))
        end
    end
end
```

The measured time does not include the time to open and close the file or the assertion because these activities take place inside a `TestMethodSetup` block, and not inside a

Test block. However, the measured time includes the time to perform the verifications. Best practice is to measure a more accurate performance boundary.

Create a performance test in a file, `fprintfTest.m`, in your current working folder. This test is similar to the regression test with the following modifications:

- The test inherits from `matlab.perftest.TestCase` instead of `matlab.unittest.TestCase`.
- The test calls the `startMeasuring` and `stopMeasuring` methods to create a boundary around the `fprintf` function call.

```
classdef fprintfTest < matlab.perftest.TestCase
    properties
        file
        fid
    end
    methods(TestMethodSetup)
        function openFile(testCase)
            testCase.file = tempname;
            testCase.fid = fopen(testCase.file, 'w');
            testCase.assertNotEqual(testCase.fid, -1, 'IO Problem')

            testCase.addTeardown(@delete, testCase.file);
            testCase.addTeardown(@fclose, testCase.fid);
        end
    end

    methods(Test)
        function testPrintingToFile(testCase)
            textToWrite = repmat('abcdef', 1, 5000000);

            testCase.startMeasuring();
            fprintf(testCase.fid, '%s', textToWrite);
            testCase.stopMeasuring();

            testCase.verifyEqual(fileread(testCase.file), textToWrite)
        end

        function testBytesToFile(testCase)
            textToWrite = repmat('tests_', 1, 5000000);

            testCase.startMeasuring();
            nbytes = fprintf(testCase.fid, '%s', textToWrite);
            testCase.stopMeasuring();
        end
    end
end
```

```
        testCase.verifyEqual(nbytes,length(textToWrite))
    end
end
end
```

The measured time for this performance test includes only the call to `fprintf`, and the testing framework still evaluates the qualifications.

Run Performance Test

Run the performance test. Depending on your system, you might see warnings that the performance testing framework ran the test the maximum number of times, but did not achieve a 0.05 relative margin of error with a 0.95 confidence level.

```
results = runperf('fprintfTest');
```

```
Running fprintfTest
```

```
.....
.....
.....
.....
.....
```

```
Done fprintfTest
```

```
results =
```

```
1x2 MeasurementResult array with properties:
```

```
    Name
    Valid
    Samples
    TestActivity
```

```
Totals:
```

```
    2 Valid, 0 Invalid.
```

The `results` variable is a 1x2 `MeasurementResult` array. Each element in the array corresponds to one of the tests defined in the test file.

Display Test Results

Display the measurement results for the first test. Your results might vary.

```

results(1)
ans =
    MeasurementResult with properties:
        Name: 'fprintfTest/testPrintingToFile'
        Valid: 1
        Samples: [10x7 table]
        TestActivity: [14x12 table]

Totals:
    1 Valid, 0 Invalid.

```

As indicated by the size of the `TestActivity` property, the performance testing framework collected 14 measurements. This number includes 4 measurements to warm up the code. The `Samples` property excludes warm-up measurements.

Display the sample measurements for the first test.

```

results(1).Samples
ans =

```

Name	MeasuredTime	Timestamp	Host
fprintfTest/testPrintingToFile	0.067772	02-Jan-2016 18:24:52	MY-HOSTNA
fprintfTest/testPrintingToFile	0.085359	02-Jan-2016 18:24:53	MY-HOSTNA
fprintfTest/testPrintingToFile	0.075863	02-Jan-2016 18:24:53	MY-HOSTNA
fprintfTest/testPrintingToFile	0.068161	02-Jan-2016 18:24:53	MY-HOSTNA
fprintfTest/testPrintingToFile	0.067606	02-Jan-2016 18:24:53	MY-HOSTNA
fprintfTest/testPrintingToFile	0.073692	02-Jan-2016 18:24:54	MY-HOSTNA
fprintfTest/testPrintingToFile	0.070815	02-Jan-2016 18:24:54	MY-HOSTNA
fprintfTest/testPrintingToFile	0.067791	02-Jan-2016 18:24:54	MY-HOSTNA
fprintfTest/testPrintingToFile	0.077599	02-Jan-2016 18:24:54	MY-HOSTNA
fprintfTest/testPrintingToFile	0.07438	02-Jan-2016 18:24:55	MY-HOSTNA

Compute Statistics for Single Test Element

Display the mean measured time for the first test. To exclude data collected in the warm-up runs, use the values in the `Samples` field.

```

sampleTimes = results(1).Samples.MeasuredTime;
meanTest = mean(sampleTimes)

```

```
meanTest =
    0.0729
```

Compute Statistics for All Test Elements

Determine the average time for all the test elements. The `fprintfTest` test includes two different methods. Compare the time for each method (test element).

Since the performance testing framework returns a `Samples` table for each test element, concatenate all these tables into one table. Then group the rows by test element `Name`, and compute the mean `MeasuredTime` for each group.

```
fullTable = vertcat(results.Samples);
summaryStats = varfun(@mean,fullTable,...
    'InputVariables','MeasuredTime','GroupingVariables','Name')
summaryStats =
```

Name	GroupCount	mean_MeasuredTime
fprintfTest/testPrintingToFile	10	0.072904
fprintfTest/testBytesToFile	27	0.079338

Both test methods write the same amount of data to a file. Therefore, some of the difference between the mean values is attributed to calling the `fprintf` function with an output argument.

Change Statistical Objectives and Rerun Tests

Change the statistical objectives defined by the `runperf` function by constructing and running a time experiment. Construct a time experiment with measurements that reach a sample mean with a 3% relative margin of error within a 97% confidence level. Collect eight warm-up measurements.

Construct an explicit test suite.

```
suite = testsuite('fprintfTest');
```

Construct a time experiment with a variable number of sample measurements, and run the tests.

```
import matlab.perftest.TimeExperiment
experiment = TimeExperiment.limitingSamplingError('NumWarmups',8,...
```

```

    'RelativeMarginOfError',0.03, 'ConfidenceLevel', 0.97);
resultsTE = run(experiment,suite);

```

```
Running fprintfTest
```

```

.....
.....
.....

```

```

.....Warning: The target Relative Margin of Error was not met after running the M
fprintfTest/testPrintingToFile.

```

```

.....
.....
.....

```

```

.....Warning: The target Relative Margin of Error was not met after running the M
fprintfTest/testBytesToFile.

```

```
Done fprintfTest
```

In this example output, the performance testing framework is not able to meet the stricter statistical objectives with the default number of maximum samples. Your results might vary.

Compute the statistics for all the test elements.

```

fullTableTE = vertcat(resultsTE.Samples);
summaryStatsTE = varfun(@mean,fullTableTE,...
    'InputVariables','MeasuredTime','GroupingVariables','Name')

summaryStatsTE =

```

Name	GroupCount	mean_MeasuredTime
fprintfTest/testPrintingToFile	32	0.081782
fprintfTest/testBytesToFile	32	0.076378

Increase the maximum number of samples to 100 and rerun the time experiment.

```

experiment = TimeExperiment.limitingSamplingError('NumWarmups',2,...
    'RelativeMarginOfError',0.03,'ConfidenceLevel',0.97,'MaxSamples',100);
resultsTE = run(experiment,suite);

```

```
Running fprintfTest
```

```
.....
```


Construct and run the time experiment.

```
import matlab.perftest.TimeExperiment
experiment = TimeExperiment.withFixedSampleSize(1);
results = run(experiment,suite);
```

```
Running fprintfTest
```

```
.
```

```
Done fprintfTest
```

Display the results. Observe the `TestActivity` table to ensure there are no warm-up samples.

```
fullTable = results.TestActivity
```

```
fullTable =
```

Name	Passed	Failed	Incomplete	MeasuredTime
fprintfTest/testPrintingToFile	true	false	false	0.065501

The performance testing framework collects one sample for each test.

See Also

`matlab.perftest.TestCase` | `matlab.perftest.TimeExperiment` |
`matlab.unittest.measurement.MeasurementResult` | `runperf` | `testsuite`

Measure Fast Executing Test Code

Performance tests that execute too quickly for MATLAB to measure accurately are filtered with an assumption failure. With the `matlab.perftest.TestCase.keepMeasuring` method, the testing framework can measure significantly faster code by automatically determining the number of times to iterate through code and measuring the average performance.

In your current working folder, create a class-based test, `PreallocationTest.m`, that compares different methods of preallocation. Since the test methods include qualifications, use the `startMeasuring` and `stopMeasuring` methods to define boundaries for the code you want to measure.

```
classdef PreallocationTest < matlab.perftest.TestCase
    methods(Test)
        function testOnes(testCase)
            testCase.startMeasuring
            x = ones(1,1e5);
            testCase.stopMeasuring
            testCase.verifyEqual(size(x),[1 1e5])
        end

        function testIndexingWithVariable(testCase)
            import matlab.unittest.constraints.IsSameSetAs
            testCase.startMeasuring
            id = 1:1e5;
            x(id) = 1;
            testCase.stopMeasuring
            testCase.verifyThat(x,IsSameSetAs(1))
        end

        function testIndexingOnLHS(testCase)
            import matlab.unittest.constraints.EveryElementOf
            import matlab.unittest.constraints.IsEqualTo
            testCase.startMeasuring
            x(1:1e5) = 1;
            testCase.stopMeasuring
            testCase.verifyThat(EveryElementOf(x),IsEqualTo(1))
        end

        function testForLoop(testCase)
            testCase.startMeasuring
            for i=1:1e5
```

```

        x(i) = 1;
    end
    testCase.stopMeasuring
    testCase.verifyNumElements(x, 1e5)
end
end
end
end

```

Run `PreallocationTest` as a performance test. Three tests are filtered because the measurements are too close to the precision of the framework.

```
results = runperf('PreallocationTest');
```

```

Running PreallocationTest
.....
=====
PreallocationTest/testOnes was filtered.
  Test Diagnostic: The MeasuredTime should not be too close to the precision of the framework.
Details
=====
.. .....
=====
PreallocationTest/testIndexingWithVariable was filtered.
  Test Diagnostic: The MeasuredTime should not be too close to the precision of the framework.
Details
=====
.... ....
=====
PreallocationTest/testIndexingOnLHS was filtered.
  Test Diagnostic: The MeasuredTime should not be too close to the precision of the framework.
Details
=====
..... ..
Done PreallocationTest

```

Failure Summary:

Name	Failed	Incomplete	Reason(s)
PreallocationTest/testOnes		X	Filtered by assumption.
PreallocationTest/testIndexingWithVariable		X	Filtered by assumption.
PreallocationTest/testIndexingOnLHS		X	Filtered by assumption.

To instruct the framework to automatically loop through the measured code and average the measurement results, modify `PreallocationTest` to use a `keepMeasuring-while` loop instead of `startMeasuring` and `stopMeasuring`.

```

classdef PreallocationTest < matlab.perftest.TestCase
    methods(Test)
        function testOnes(testCase)
            while(testCase.keepMeasuring)

```

```
        x = ones(1,1e5);
    end
    testCase.verifyEqual(size(x),[1 1e5])
end

function testIndexingWithVariable(testCase)
    import matlab.unittest.constraints.IsSameSetAs
    while(testCase.keepMeasuring)
        id = 1:1e5;
        x(id) = 1;
    end
    testCase.verifyThat(x,IsSameSetAs(1))
end

function testIndexingOnLHS(testCase)
    import matlab.unittest.constraints.EveryElementOf
    import matlab.unittest.constraints.IsEqualTo
    while(testCase.keepMeasuring)
        x(1:1e5) = 1;
    end
    testCase.verifyThat(EveryElementOf(x),IsEqualTo(1))
end

function testForLoop(testCase)
    while(testCase.keepMeasuring)
        for i=1:1e5
            x(i) = 1;
        end
    end
    testCase.verifyNumElements(x,1e5)
end
end
end
```

Rerun the tests. All the tests complete.

```
results = runperfn('PreallocationTest');
```

```
Running PreallocationTest
.....
Done PreallocationTest
```

View the results.

```
sampleSummary(results)
```

```
ans =
```

```
4x7 table
```

Name	SampleSize	Mean	StandardDeviation	Min
PreallocationTest/testOnes	4	2.8644e-05	7.5098e-07	2.7967e-0
PreallocationTest/testIndexingWithVariable	4	0.00042645	3.877e-06	0.0004238
PreallocationTest/testIndexingOnLHS	4	4.2269e-05	2.5913e-07	4.1886e-0
PreallocationTest/testForLoop	4	0.008932	0.00023528	0.008770

See Also

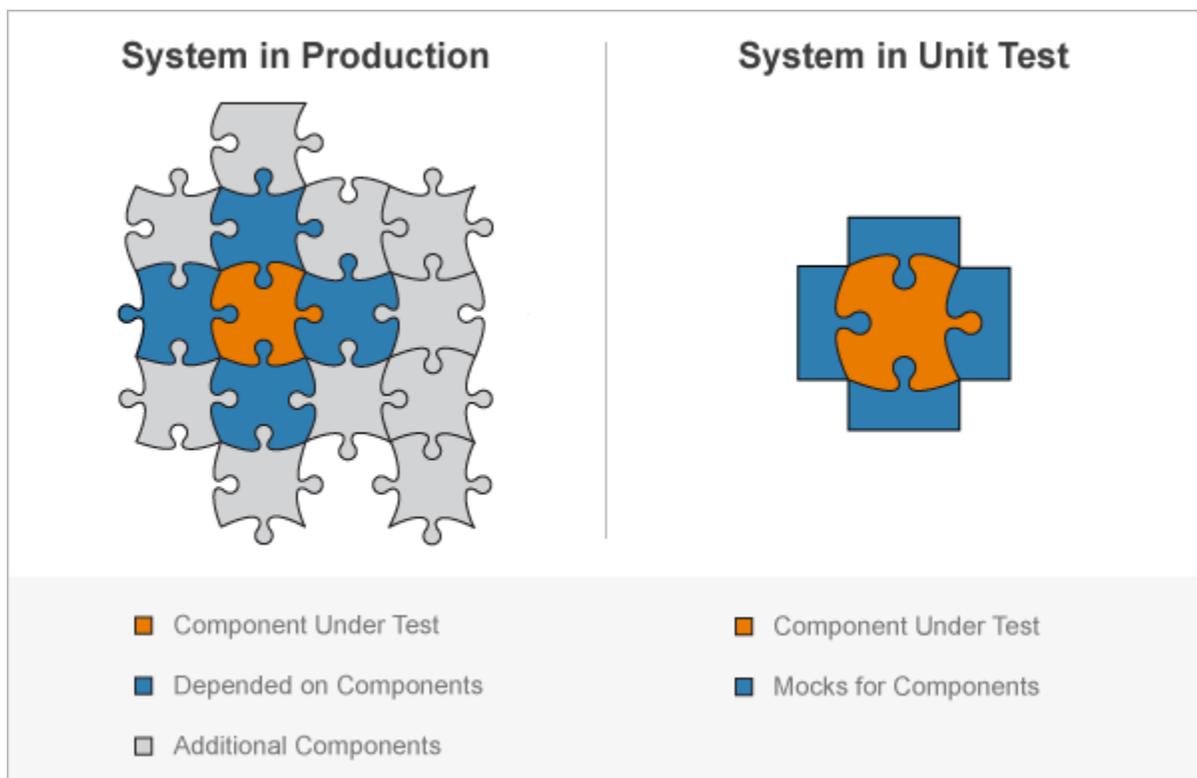
`matlab.perftest.TestCase.keepMeasuring` | `runperf`

Related Examples

- “Overview of Performance Testing Framework” on page 32-179
- “Test Performance Using Classes” on page 32-190

Create Mock Object

When unit testing, you are often interested in testing a portion of a complete system, isolated from the components it depends on. To test a portion of the system, we can substitute mock objects to replace the depended-on components. A mock object implements at least part of the same interface as the production object, but often in a manner that is simple, efficient, predictable, and controllable. When you use the mocking framework, the component under test is unaware of whether its collaborator is a "real" object or a mock object.



For example, suppose you want to test an algorithm for buying stock, but you do not want to test the entire system. You could use a mock object to replace the functionality of looking up the stock price, and another mock object to verify that the trader purchased the stock. The algorithm you are testing does not know that it is operating on mock objects, and you can test the algorithm isolated from the rest of the system.

Using a mock object, you can define behavior (a process known as *stubbing*). For example, you can specify that an object produces predefined responses to queries. You can also intercept and remember messages sent from the component under test to the mock object (a process known as *spying*). For example, you can verify that a particular method was called or a property was set.

The typical workflow to test a component in isolation is as follows:

- 1 Create mocks for the depended-on components.
- 2 Define behaviors of the mocks. For example, define the outputs that a mocked method or property returns when it is called with a particular set of inputs.
- 3 Test the component of interest.
- 4 Qualify interactions between the component of interest and the mocked components. For example, verify that a mocked method was called with particular inputs, or that a property was set.

Depended on Components

In this example, the component under test is a simple day-trading algorithm. It is the part of the system you want to test independent of other components. The day-trading algorithm has two dependencies: a data service to retrieve the stock price data and a broker to purchase the stock.

In a file `DataService.m` in your current working folder, create an abstract class that includes a `lookupPrice` method.

```
classdef DataService
    methods (Abstract,Static)
        price = lookupPrice(ticker,date)
    end
end
```

In production code, there could be several concrete implementations of the `DataService` class, such as a `BloombergDataService` class. This class uses the Datafeed Toolbox™. However, since we create a mock of the `DataService` class, you do not need to have the toolbox installed to run the tests for the trading algorithm.

```
classdef BloombergDataService < DataService
    methods (Static)
```

```
function price = lookupPrice(ticker,date)
    % This method assumes you have installed and configured the
    % Bloomberg software.
    conn = blp;
    data = history(conn,ticker,'LAST_PRICE',date-1,date);
    price = data(end);
    close(conn)
end
end
end
```

In this example, assume that the broker component has not been developed yet. Once it is implemented, it will have a `buy` method that accepts a ticker symbol and a specified number of shares to buy, and returns a status code. The mock for the broker component uses an implicit interface, and does not derive from a superclass.

Component Under Test

In a file `trader.m` in your current working folder, create a simple day trading algorithm. The `trader` function accepts as inputs a data service object that looks up the price of the stock, a broker object that defines how the stock is bought, a ticker symbol, and a number of shares to purchase. If the price from yesterday is less than the price two days ago, instruct the broker to buy the specified number of shares.

```
function trader(dataService,broker,ticker,numShares)
    yesterday = datetime('yesterday');
    priceYesterday = dataService.lookupPrice(ticker,yesterday);
    price2DaysAgo = dataService.lookupPrice(ticker,yesterday-days(1));

    if priceYesterday < price2DaysAgo
        broker.buy(ticker,numShares);
    end
end
```

Mock Objects and Behavior Objects

The mock object is an implementation of the abstract methods and properties of the interface specified by a superclass. You can also construct a mock without a superclass, in which case the mock has an implicit interface. The mock object carries out the actions for the component under test, such as calling a method or accessing a property.

When you create a mock, you also create an associated behavior object. The behavior object defines the same methods as the mock object and controls mock behavior. Use the behavior object to define mock actions and qualify interactions. For example, use it to define values a mocked method returns, or verify that a property was accessed.

At the command prompt, create a mock test case for interactive use. Using the mock in a test class instead of at the command prompt is presented later in this example.

```
import matlab.mock.TestCase
testCase = TestCase.forInteractiveUse;
```

Create Stub to Define Behavior

Create a mock for the data service dependency and examine the methods on it. The data service mock returns predefined values, replacing the implementation of the service that provides actual stock prices. Therefore, it exhibits stubbing behavior.

```
[stubDataService,dataServiceBehavior] = createMock(testCase,?DataService);
methods(stubDataService)
```

```
Methods for class matlab.mock.classes.DataServiceMock:
```

```
Static methods:
```

```
lookupPrice
```

In the `DataService` class, the `lookupPrice` method is abstract and static. The mocking framework implements this method as concrete and static.

Define behavior for the data service mock. For ticker symbol "F00", it returns the price yesterday as \$123 and anything before yesterday is \$234. Therefore, according to the `trader` function, the broker always buys stock "F00". For the ticker symbol "BAR", it returns the price yesterday as \$765 and anything before yesterday is \$543. Therefore, the broker never buys stock "BAR".

```
import matlab.unittest.constraints.IsLessThan
yesterday = datetime('yesterday');

testCase.assignOutputsWhen(dataServiceBehavior.lookupPrice(...
    "F00", yesterday), 123);
```

```
testCase.assignOutputsWhen(dataServiceBehavior.lookupPrice(...
    "F00", IsLessThan(yesterday)), 234);

testCase.assignOutputsWhen(dataServiceBehavior.lookupPrice(...
    "BAR", yesterday), 765);
testCase.assignOutputsWhen(dataServiceBehavior.lookupPrice(...
    "BAR", IsLessThan(yesterday)), 543);
```

You can now call the mocked `lookupPrice` method.

```
p1 = stubDataService.lookupPrice("F00", yesterday)
p2 = stubDataService.lookupPrice("BAR", yesterday-days(5))
```

```
p1 =
    123
```

```
p2 =
    543
```

While the `assignOutputsWhen` method on `testCase` is convenient to specify behavior, there is more functionality if you use the `AssignOutputs` action. For more information, see “Specify Mock Object Behavior” on page 32-211.

Create Spy to Intercept Messages

Create a mock for the broker dependency and examine the methods on it. Since the broker mock is used to verify interactions with the component under test (the `trader` function), it exhibits spying behavior. The broker mock has an implicit interface. While the `buy` method is not currently implemented, you can create a mock with it.

```
[spyBroker, brokerBehavior] = createMock(testCase, 'AddedMethods', {'buy'});
methods(spyBroker)
```

```
Methods for class matlab.mock.classes.Mock:
```

```
buy
```

Call the `buy` method of the mock. By default it returns empty.

```
s1 = spyBroker.buy  
s2 = spyBroker.buy("inputs",[13 42])
```

```
s1 =  
    []
```

```
s2 =  
    []
```

Since the `trader` function does not use the status return code, the default mock behavior of returning empty is acceptable. The broker mock is a pure spy, and does not need to implement any stubbing behavior.

Call Component Under Test

Call the `trader` function. In addition to the ticker symbol and the number of shares to buy, the `trader` function takes as inputs the data service and the broker. Instead of passing in actual data service and broker objects, pass in the `spyBroker` and `stubDataService` mocks.

```
trader(stubDataService,spyBroker,"F00",100)  
trader(stubDataService,spyBroker,"F00",75)  
trader(stubDataService,spyBroker,"BAR",100)
```

Verify Function Interactions

Use the broker behavior object (the spy) to verify that the `trader` function calls the `buy` method, as expected.

Use the `TestCase.verifyCalled` method to verify that the `trader` function instructed the `buy` method to buy 100 shares of the F00 stock.

```
import matlab.mock.constraints.WasCalled;  
testCase.verifyCalled(brokerBehavior.buy("F00",100))
```

```
Interactive verification passed.
```

Verify that F00 stock was purchased two times, regardless of the specified number of shares. While the `verifyCalled` method is convenient to specify behavior, there is more

functionality if you use the `WasCalled` constraint. For example, you can verify that a mocked method was called a specified number of times.

```
import matlab.unittest.constraints.IsAnything
testCase.verifyThat(brokerBehavior.buy("FOO",IsAnything), ...
    WasCalled('WithCount',2))
```

Interactive verification passed.

Verify that the `buy` method was not called requesting 100 shares of the `BAR` stock.

```
testCase.verifyNotCalled(brokerBehavior.buy("BAR",100))
```

Interactive verification passed.

Although the `trader` function was called requesting 100 shares of `BAR` stock, the stub defined yesterday's price for `BAR` to return a higher value than all days prior to yesterday. Therefore, the broker never buys stock `"BAR"`.

Test Class for trader Function

The interactive test case is convenient to experiment with at the command prompt. However, it is typical to create and use mocks within a test class. In a file in your current working folder, create the following test class that incorporates the interactive testing from this example.

```
classdef TraderTest < matlab.mock.TestCase
    methods(Test)
        function buysStockWhenDrops(testCase)
            import matlab.unittest.constraints.IsLessThan
            import matlab.unittest.constraints.IsAnything
            import matlab.mock.constraints.WasCalled
            yesterday = datetime('yesterday');

            % Create mocks
            [stubDataService,dataServiceBehavior] = createMock(testCase, ...
                ?DataService);
            [spyBroker,brokerBehavior] = createMock(testCase, ...
                'AddedMethods',{'buy'});

            % Set up behavior
            testCase.assignOutputsWhen(dataServiceBehavior.lookupPrice(...
                "FOO",yesterday),123);
            testCase.assignOutputsWhen(dataServiceBehavior.lookupPrice(...
```

```

        "FOO", IsLessThan(yesterday)), 234);

    % Call function under test
    trader(stubDataService, spyBroker, "FOO", 100)
    trader(stubDataService, spyBroker, "FOO", 75)

    % Verify interactions
    testCase.verifyCalled(brokerBehavior.buy("FOO", 100))
    testCase.verifyThat(brokerBehavior.buy("FOO", IsAnything), ...
        WasCalled('WithCount', 2))
end
function doesNotBuyStockWhenIncreases(testCase)
    import matlab.unittest.constraints.IsLessThan
    yesterday = datetime('yesterday');

    % Create mocks
    [stubDataService, dataServiceBehavior] = createMock(testCase, ...
        ?DataService);
    [spyBroker, brokerBehavior] = createMock(testCase, ...
        'AddedMethods', {'buy'});

    % Set up behavior
    testCase.assignOutputsWhen(dataServiceBehavior.lookupPrice(...
        "BAR", yesterday), 765);
    testCase.assignOutputsWhen(dataServiceBehavior.lookupPrice(...
        "BAR", IsLessThan(yesterday)), 543);

    % Call function under test
    trader(stubDataService, spyBroker, "BAR", 100)

    % Verify interactions
    testCase.verifyNotCalled(brokerBehavior.buy("BAR", 100))
end
end
end

```

Run the tests and view a table of the results.

```

results = runtests('TraderTest');
table(results)

```

```

Running TraderTest
..
Done TraderTest

```

ans =

2x6 table

Name	Passed	Failed	Incomplete	Dura
'TraderTest/buysStockWhenDrops'	true	false	false	0.1
'TraderTest/doesNotBuyStockWhenIncreases'	true	false	false	0.0

See Also

Specify Mock Object Behavior

In this section...

“Define Mock Method Behavior” on page 32-211

“Define Mock Property Behavior” on page 32-213

“Define Repeating and Subsequent Behavior” on page 32-214

“Summary of Behaviors” on page 32-216

When you create a mock, you create an associated behavior object that controls mock behavior. Use this object to define mock method and property behavior (*stub*). For more information on creating a mock, see “Create Mock Object” on page 32-202.

The mock object is an implementation of the abstract methods and properties of the interface specified by a superclass. You can also construct a mock without a superclass, in which case the mock has an implicit interface.

Create a mock with an implicit interface. The interface includes `Name` and `ID` properties and a `findUser` method that accepts an identifier and returns a name. While the interface is not currently implemented, you can create a mock with it.

```
testCase = matlab.mock.TestCase.forInteractiveUse;  
[mock,behaviorObj] = testCase.createMock('AddedProperties', ...  
    {'Name','ID'}, 'AddedMethods',{'findUser'});
```

Define Mock Method Behavior

You can specify that a mock method returns specific values or throws an exception in different situations.

Specify that when the `findUser` method is called with any inputs, it returns "Unknown". By default, MATLAB returns an empty array when you call the `findUser` method.

- The `assignOutputsWhen` method defines return values for the method call.
- The mocked method call (`behaviorObj.findUser`) implicitly creates a `MethodCallBehavior` object.
- The `withAnyInputs` method of the `MethodCallBehavior` object specifies that the behavior applies to a method call with any number of inputs with any value.

```
testCase.assignOutputsWhen(withAnyInputs(behaviorObj.findUser), "Unknown")
n = mock.findUser(1)
```

```
n =
    "Unknown"
```

Specify that when the input value is 1701, the mock method returns "Jim". This behavior supersedes the return of "Unknown" for the input value of 1701 only because it was defined after that specification.

```
testCase.assignOutputsWhen(behaviorObj.findUser(1701), "Jim")
n = mock.findUser(1701)
```

```
n =
    "Jim"
```

Specify that when the `findUser` method is called with only the object as input, the mock method returns "Unspecified ID". The `withExactInputs` method of the `MethodCallBehavior` object specifies that the behavior applies to a method call with the object as the only input value.

```
testCase.assignOutputsWhen(withExactInputs(behaviorObj.findUser), ...
    "Unspecified ID")
n = mock.findUser % equivalent to n = findUser(mock)
```

```
n =
    "Unspecified ID"
```

You can use classes in the `matlab.unittest.constraints` package to help define behavior. Specify that `findUser` throws an exception when it is called with an ID greater than 5000.

```
import matlab.unittest.constraints.IsGreaterThan
testCase.throwExceptionWhen(behaviorObj.findUser(IsGreaterThan(5000)));
n = mock.findUser(5001)
```

```
Error using
matlab.mock.internal.MockContext/createMockObject/mockMethodCallback (line 323)
The following method call was specified to throw an exception:
    findUser([1x1 matlab.mock.classes.Mock], 5001)
```

You can define behavior based on the number of outputs requested in a method call. If the method call requests two output values, return "??" for the name and -1 for the ID.

```
testCase.assignOutputsWhen(withNargout(2, ...
    withAnyInputs(behaviorObj.findUser)), "??", -1)
[n,id] = mock.findUser(13)

n =
    "??"

id =
    -1
```

Define Mock Property Behavior

When a mock property is accessed, you can specify that it returns specific or stored property values. When it is set, you can specify when the mock stores the property value. You can also define when the testing framework throws an exception for mock property set or access activities.

When defining mock property behavior, keep in mind that displaying a property value in the command window is a property access (get) operation.

Similar to defining mock method behavior, defining mock property behavior requires an instance of the `PropertyBehavior` class. The framework returns an instance of this class when you access a mock property. To define access behavior, use an instance of `PropertyGetBehavior` by calling the `get` method of the `PropertyBehavior` class. To define set behavior, use an instance of the `PropertySetBehavior` by calling the `set` or `setToValue` method of the `PropertyBehavior` class.

Specify that when the `Name` property is set to any value, the testing framework throws an exception.

- The `throwExceptionWhen` method instructs the framework to throw an exception for a specified behavior.
- Accessing a property on the behavior object `PropertyBehavior` class (`behaviorObj.Name`) creates a `PropertyBehavior` class instance.
- The call to the `set` method of the `PropertyBehavior` class creates a `PropertySetBehavior`.

```
testCase.throwExceptionWhen(set(behaviorObj.Name))
mock.Name = "Sue";
```

```
Error using matlab.mock.internal.MockContext/createMockObject/mockPropertySetCallback
The following property set was specified to throw an exception:
    <Mock>.Name = "Sue"
```

Allow the mock to store the value when the property is set to "David".

```
testCase.storeValueWhen(setToValue(behaviorObj.Name, "David"));
mock.Name = "David"
```

```
mock =
```

```
Mock with properties:
```

```
    Name: "David"
       ID: []
```

Define Repeating and Subsequent Behavior

The `matlab.mock.TestCase` methods are convenient for defining behavior. However, there is more functionality when you use a class in the `matlab.mock.actions` package instead. Using these classes, you can define behavior that repeats the same action multiple times and specify subsequent actions. To define repeating or subsequent behavior, pass an instance of a class in the `matlab.mock.actions` package to the `when` method of the behavior class.

Assign the value of 1138 to the ID property and then throw an exception for property access.

```
import matlab.mock.actions.AssignOutputs
import matlab.mock.actions.ThrowException
when(get(behaviorObj.ID), then(AssignOutputs(1138), ThrowException))
id = mock.ID
id = mock.ID

id =
```

```
1138
```

```
Error using matlab.mock.internal.MockContext/createMockObject/mockPropertyGetCallback
The following property access was specified to throw an exception:
    <Mock>.ID
```

Assign the value of 1138 and then 237 to the ID property. Then, throw an exception for property access. Each call to the `then` method accepts up to two actions. To specify more subsequent actions, use multiple calls to `then`.

```
when(get(behaviorObj.ID), then(AssignOutputs(1138), ...
    then(AssignOutputs(237), ThrowException)))
```

```
id = mock.ID
id = mock.ID
id = mock.ID
```

```
id =
```

```
    1138
```

```
id =
```

```
    237
```

```
Error using matlab.mock.internal.MockContext/createMockObject/mockPropertyGetCallback
The following property access was specified to throw an exception:
    <Mock>.ID
```

If the object is the only input value, specify the `findUser` function return the value of "Phil" twice.

```
when(withExactInputs(behaviorObj.findUser), repeat(2, AssignOutputs("Phil")))
n = mock.findUser
n = mock.findUser
```

```
n = mock.findUser
```

```
n =
```

```
    "Phil"
```

```
n =
```

```
    "Phil"
```

Call the function a third time. If you repeat an action, and do not follow it with a call to the `then` method, the mock continues to return the repeated value.

```
n = mock.findUser
```

```
n =
    "Phil"
```

Define behavior for setting the value of Name. Throw an exception the first two times and then store the value.

```
import matlab.mock.actions.StoreValue
when(set(behaviorObj.Name), then(repeat(2, ThrowException), StoreValue))
mock.Name = "John"
```

```
Error using matlab.mock.internal.MockContext/createMockObject/mockPropertySetCallback
The following property set was specified to throw an exception:
    <Mock>.Name = "John"
```

```
mock.Name = "Penny"
```

```
Error using matlab.mock.internal.MockContext/createMockObject/mockPropertySetCallback
The following property set was specified to throw an exception:
    <Mock>.Name = "Penny"
```

```
mock.Name = "Tommy"
```

```
mock =
    Mock with properties:
        Name: "Tommy"
```

Summary of Behaviors

Behavior	TestCase Method	matlab.mock.Actions Class (Allows for Definition of Repeat and Subsequent Behavior)
Return specified values for method call and property access.	assignOutputsWhen	AssignOutputs
Return stored value when property is accessed.	returnStoredValueWhen	ReturnStoredValue

Behavior	TestCase Method	matlab.mock.Actions Class (Allows for Definition of Repeat and Subsequent Behavior)
Store value when property is set.	storeValueWhen	StoreValue
Throw exception when method is called or when property is set or accessed.	throwExceptionWhen	ThrowException

See Also

matlab.mock.TestCase | matlab.mock.actions.AssignOutputs |
matlab.mock.actions.ReturnStoredValue | matlab.mock.actions.StoreValue
| matlab.mock.actions.ThrowException

Related Examples

- “Create Mock Object” on page 32-202

Qualify Mock Object Interaction

When you create a mock, you create an associated behavior object that controls mock behavior. Use this object to access intercepted messages sent from the component under test to the mock object (a process known as *spying*). For more information on creating a mock, see “Create Mock Object” on page 32-202.

In the mocking framework, qualifications are functions used to test interactions with the object. There are four types of qualifications:

- Verifications — Produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Assumptions — Ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as Incomplete.
- Assertions — Ensure that a failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.
- Fatal Assertions — Abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

The mock object is an implementation of the abstract methods and properties of the interface specified by a superclass. You can also construct a mock without a superclass, in which case the mock has an implicit interface. Create a mock with an implicit interface for a dice class. The interface includes `Color` and `NumSides` properties and a `roll` method that accepts a number of dice and returns a value. While the interface is not currently implemented, you can create a mock with it.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behaviorObj] = testCase.createMock('AddedProperties', ...
    {'NumSides', 'Color'}, 'AddedMethods', {'roll'});
```

Qualify Mock Method Interaction

Since the mock records interactions sent to it, you can qualify that a mock method was called. Roll one die.

```
val = mock.roll(1);
```

Verify that the `roll` method was called with 1 die.

```
testCase.verifyCalled(behaviorObj.roll(1))
```

Interactive verification passed.

Verify that the `roll` method was called with 3 dice. This test fails.

```
testCase.verifyCalled(behaviorObj.roll(3), ...
    'roll method should have been called with input 3.')
```

Interactive verification failed.

```
-----
Test Diagnostic:
```

```
-----
roll method should have been called with input 3.
```

```
-----
Framework Diagnostic:
```

```
-----
verifyCalled failed.
--> Method 'roll' was not called with the specified signature.
--> Observed method call(s) with any signature:
    out = roll([1x1 matlab.mock.classes.Mock], 1)
```

```
Specified method call:
    MethodCallBehavior
    [...] = roll(<Mock>, 3)
```

Verify that the `roll` method was not called with 2 dice.

```
testCase.verifyNotCalled(behaviorObj.roll(2))
```

Interactive verification passed.

Since the `withAnyInputs`, `withExactInputs`, and `withNargout` methods of the `MethodCallBehavior` class return `MethodCallBehavior` objects, you can use them in qualifications. Verify that the `roll` method was called at least once with any inputs.

```
testCase.verifyCalled(withAnyInputs(behaviorObj.roll))
```

Interactive verification passed.

Verify that the `roll` method was not called with 2 outputs and any inputs.

```
testCase.verifyNotCalled(withNargout(2,withAnyInputs(behaviorObj.roll)))
```

Interactive verification passed.

Qualify Mock Property Interaction

Similar to method calls, the mock records property set and access operations. Set the color of the dice.

```
mock.Color = "red"
```

```
mock =
```

```
    Mock with properties:
```

```
    NumSides: []  
    Color: "red"
```

Verify that the color was set.

```
testCase.verifySet(behaviorObj.Color)
```

Interactive verification passed.

Verify the color was accessed. This test passes because there is an implicit property access when MATLAB displays the object.

```
testCase.verifyAccessed(behaviorObj.Color)
```

Interactive verification passed.

Assert that the number of sides was not set.

```
testCase.assertNotSet(behaviorObj.NumSides)
```

Interactive assertion passed.

Use Mock Object Constraints

The `matlab.mock.TestCase` methods are convenient for spying on mock interactions. However, there is more functionality when you use a class in the `matlab.mock.constraints` package instead. To use a constraint, pass the behavior object and constraint to the `verifyThat`, `assumeThat`, `assertThat` or `fatalAssertThat` method.

Create a new mock object.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behaviorObj] = testCase.createMock('AddedProperties', ...
    {'NumSides', 'Color'}, 'AddedMethods', {'roll'});
```

Roll 2 dice. Then use a constraint to verify that the `roll` method was called at least once with two dice.

```
val = mock.roll(2);

import matlab.mock.constraints.WasCalled
testCase.verifyThat(behaviorObj.roll(2),WasCalled)
```

Interactive verification passed.

Roll one die. Then verify that the `roll` method was called at least twice with any inputs.

```
val = mock.roll(1);

testCase.verifyThat(withAnyInputs(behaviorObj.roll), ...
    WasCalled('WithCount',2))
```

Interactive verification passed.

Verify that `NumSides` was not accessed.

```
import matlab.mock.constraints.WasAccessed
testCase.verifyThat(behaviorObj.NumSides,~WasAccessed)
```

Interactive verification passed.

Set the color of the dice. Then verify the property was set once.

```
mock.Color = "blue";
```

```
import matlab.mock.constraints.WasSet
testCase.verifyThat(behaviorObj.Color,WasSet('WithCount',1))
```

```
Interactive verification passed.
```

Access the Color property. Then verify that it was not accessed exactly once. This test fails.

```
c = mock.Color
```

```
testCase.verifyThat(behaviorObj.Color,~WasAccessed('WithCount',1))
```

```
c =
```

```
    "blue"
```

```
Interactive verification failed.
```

```
-----  
Framework Diagnostic:  
-----
```

```
Negated WasAccessed failed.
```

```
--> Property 'Color' was accessed the prohibited number of times.
```

```
    Actual property access count:
```

```
        1
```

```
    Prohibited property access count:
```

```
        1
```

```
Specified property access:
```

```
    PropertyGetBehavior
```

```
    <Mock>.Color
```

Set the number of sides. Then, verify that the number of sides was set to 22.

```
mock.NumSides = 22;
```

```
testCase.verifyThat(behaviorObj.NumSides,WasSet('ToValue',22))
```

```
Interactive verification passed.
```

Use a constraint from the `matlab.unittest.constraints` package to assert that the number of dice sides isn't set to more than 20. This test fails.

```
import matlab.unittest.constraints.IsLessThanOrEqualTo
testCase.verifyThat(behaviorObj.NumSides, ...
    WasSet('ToValue', IsLessThanOrEqualTo(20)))
```

Interactive verification failed.

```
-----
Framework Diagnostic:
-----
```

```
WasSet failed.
--> Property 'NumSides' was not set to the specified value.
--> Observed property set(s) to any value:
    <Mock>.NumSides = 22
```

```
Specified property set:
    PropertySetBehavior
    <Mock>.NumSides = <IsLessThanOrEqualTo constraint>
```

Summary of Qualifications

Type of Qualification	TestCase Method	matlab.mock.constraints Class	
		Use matlab.unittest.TestCase Method	With matlab.mock.constraints Class
Method was called	verifyCalled or verifyNotCalled	verifyThat	WasCalled or Occurred
	assumeCalled or assumeNotCalled	assumeThat	
	assertCalled or assertNotCalled	assertThat	
	fatalAssertCalled or fatalAssertNotCalled	fatalAssertThat	
Method was called a certain number of times	Not applicable	verifyThat, assumeThat, assertThat, or fatalAssertThat	WasCalled

Type of Qualification	TestCase Method	matlab.mock.constraints Class	
		Use matlab.unittest.TestCase Method	With matlab.mock.constraints Class
Property was accessed	verifyAccessed or verifyNotAccessed	verifyThat	WasAccessed or Occurred
	assumeAccessed or assumeNotAccessed	assumeThat	
	assertAccessed or assertNotAccessed	assertThat	
	fatalAssertAccessed or fatalAssertNotAccessed	fatalAssertThat	
Property was accessed a certain number of times	Not applicable	verifyThat, assumeThat, assertThat, or fatalAssertThat	WasAccessed
Property was set	verifySet or verifyNotSet	verifyThat	WasSet or Occurred
	assumeSet or assumeNotSet	assumeThat	
	assertSet or assertNotSet	assertThat	
	fatalAssertSet or fatalAssertNotSet	fatalAssertThat	

Type of Qualification	TestCase Method	matlab.mock.constraints Class	
		Use matlab.unittest.TestCase Method	With matlab.mock.constraints Class
Property was set a certain number of times	Not applicable	verifyThat, assumeThat, assertThat, or fatalAssertThat	WasSet
Property was set to a certain value	Not applicable	verifyThat, assumeThat, assertThat, or fatalAssertThat	WasSet or Occurred
Methods were called and properties were accessed or set in a particular order	Not applicable	verifyThat, assumeThat, assertThat, or fatalAssertThat	Occurred

See Also

System object Usage and Authoring

- “What Are System Objects?” on page 33-2
- “System Objects vs MATLAB Functions” on page 33-6
- “System Design in MATLAB Using System Objects” on page 33-9
- “Define Basic System Objects” on page 33-15
- “Change the Number of Inputs” on page 33-18
- “Validate Property and Input Values” on page 33-22
- “Initialize Properties and Setup One-Time Calculations” on page 33-25
- “Set Property Values at Construction Time” on page 33-28
- “Reset Algorithm and Release Resources” on page 33-30
- “Define Property Attributes” on page 33-32
- “Hide Inactive Properties” on page 33-36
- “Limit Property Values to Finite List” on page 33-38
- “Process Tuned Properties” on page 33-43
- “Define Composite System Objects” on page 33-45
- “Define Finite Source Objects” on page 33-48
- “Save and Load System Object” on page 33-50
- “Define System Object Information” on page 33-54
- “Handle Input Specification Changes” on page 33-56
- “Summary of Call Sequence” on page 33-59
- “Detailed Call Sequence” on page 33-63
- “Tips for Defining System Objects” on page 33-66
- “Insert System Object Code Using MATLAB Editor” on page 33-69
- “Analyze System Object Code” on page 33-77
- “Use Global Variables in System Objects” on page 33-80

What Are System Objects?

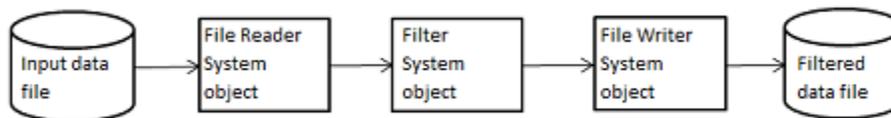
In this section...

“Running a System Object” on page 33-3

“System Object Functions” on page 33-4

A System object is a specialized MATLAB object. Many toolboxes include System objects. System objects are designed specifically for implementing and simulating dynamic systems with inputs that change over time. Many signal processing, communications, and controls systems are dynamic. In a dynamic system, the values of the output signals depend on both the instantaneous values of the input signals and on the past behavior of the system. System objects use internal states to store that past behavior, which is used in the next computational step. As a result, System objects are optimized for iterative computations that process large streams of data in segments, such as video and audio processing systems. This ability to process streaming data provides the advantage of not having to hold large amounts of data in memory. Use of streaming data also allows you to use simplified programs that use loops efficiently.

For example, you could use System objects in a system that reads data from a file, filters that data and then writes the filtered output to another file. Typically, a specified amount of data is passed to the filter in each loop iteration. The file reader object uses a state to track where in the file to begin the next data read. Likewise, the file writer object tracks where it last wrote data to the output file so that data is not overwritten. The filter object maintains its own internal states to ensure that the filtering is performed correctly. This diagram represents a single loop of the system.



These advantages make System objects well suited for processing streaming data.

Many System objects support:

- Fixed-point arithmetic (requires a Fixed-Point Designer™ license)
- C code generation (requires a MATLAB Coder™ or Simulink Coder license)

- HDL code generation (requires an HDL Coder™ license)
- Executable files or shared libraries generation (requires a MATLAB Compiler license)

Note Check the product documentation to confirm fixed-point, code generation, and MATLAB Compiler support for the specific System objects you want to use.

System objects use a minimum of two commands to process data:

- Creation of the object (such as, `fft256 = dsp.FFT`)
- Running data through the object (such as, `fft256(x)`)

This separation of creation from execution lets you create multiple, persistent, reusable objects, each with different settings. Using this approach avoids repeated input validation and verification, allows for easy use within a programming loop, and improves overall performance. In contrast, MATLAB functions must validate parameters every time you call the function.

In addition to the System objects provided with System Toolboxes, you can create your own System objects. See “Create System Objects”.

Running a System Object

To run a System object and perform the operation defined by its algorithm, you call the object as if it were a function. For example, to create an FFT object that uses the `dsp.FFT` System object, specifies a length of 1024, and names it `dft`, use:

```
dft = dsp.FFT('FFTLengthSource', 'Property', 'FFTLength', 1024);
```

To run this object with the input `x`, use:

```
dft(x);
```

If you run a System object without any input arguments, you must include empty parentheses. For example, `asysobj()`.

When you run a System object, it also performs other important tasks related to data processing, such as initialization and handling object states.

Note An alternative way to run a System object is to use the `step` function. For example, for an object created using `dft = dsp.FFT`, you can run it using `step(dft, x)`.

System Object Functions

After you create a System object, you use various object functions to process data or obtain information from or about the object. The syntax for using functions is `<object function name>(<system object name>)`, plus possible extra input arguments. For example, for `txfourier = dsp.FFT`, where `txfourier` is a name you assign, you call the `reset` function using `reset(txfourier)`.

Common Object Functions

All System objects support the following object functions. In cases where a function is not applicable to a particular object, calling that function has no effect on the object.

Function	Description
Run the object function, or step	<p>Runs the object to process data using the algorithm defined by that object.</p> <p><i>Example:</i> For the object <code>dft = dsp.FFT;</code>, run the object via:</p> <ul style="list-style-type: none"> • <code>y = dft(x)</code> • <code>y = step(dft,x)</code> <p>As part of this processing, the object initializes resources, returns outputs, and updates the object states as necessary. During execution, you can change only tunable properties. Both ways of running a System object return regular MATLAB variables.</p>
<code>release</code>	Release resources and allow changes to System object property values and additional characteristics that are limited while the System object is in use.
<code>reset</code>	Resets the System object to the initial values for that object.
<code>nargin</code>	Returns the number of inputs accepted by the System object algorithm definition. If the algorithm definition includes <code>varargin</code> , the <code>nargin</code> output is negative.
<code>nargout</code>	Returns the number of outputs accepted by the System object algorithm definition. If the algorithm definition includes <code>varargout</code> , the <code>nargout</code> output is negative.

Function	Description
<code>getDiscreteState</code>	Returns a structure of the object discrete states. If the object has no discrete states, <code>getDiscreteState</code> returns an empty structure.
<code>clone</code>	Creates another object of the same type with the same property values
<code>isLocked</code>	Returns a logical value indicating whether the object has been called and you have not yet called <code>release</code> on the object.
<code>isDone</code>	Applies only to source objects that inherit from <code>matlab.system.mixin.FiniteSource</code> . Returns a logical value indicating whether the end of the data file has been reached. If a particular object does not have end-of-data capability, this function value always returns <code>false</code> .
<code>info</code>	Returns a structure containing characteristic information about the object. The fields of this structure vary depending on the object. If a particular object does not have characteristic information, the structure is empty.

See Also

`matlab.System`

Related Examples

- “System Objects vs MATLAB Functions” on page 33-6
- “System Design in MATLAB Using System Objects” on page 33-9
- “System Design in Simulink Using System Objects” (Simulink)

System Objects vs MATLAB Functions

In this section...

“System Objects vs. MATLAB Functions” on page 33-6

“Process Audio Data Using Only MATLAB Functions Code” on page 33-6

“Process Audio Data Using System Objects” on page 33-7

System Objects vs. MATLAB Functions

Many System objects have MATLAB function counterparts. For simple, one-time computations, use MATLAB functions. However, if you need to design and simulate a system with many components, use System objects. Using System objects is also appropriate if your computations require managing internal states, have inputs that change over time or process large streams of data.

Building a dynamic system with different execution phases and internal states using only MATLAB functions would require complex programming. You would need code to initialize the system, validate data, manage internal states, and reset and terminate the system. System objects perform many of these managerial operations automatically during execution. By combining System objects in a program with other MATLAB functions, you can streamline your code and improve efficiency.

Process Audio Data Using Only MATLAB Functions Code

This example shows how to write MATLAB® function-only code for reading audio data.

The code reads audio data from a file, filters it, and plays the filtered audio data. The audio data is read in frames. This code produces the same result as the System objects code in the next example, allowing you to compare approaches.

Locate source audio file.

```
fname = 'speech_dft_8kHz.wav';
```

Obtain the total number of samples and the sampling rate from the source file.

```
audioInfo = audioinfo(fname);  
maxSamples = audioInfo.TotalSamples;  
fs = audioInfo.SampleRate;
```

Define the filter to use.

```
b = fir1(160, .15);
```

Initialize the filter states.

```
z = zeros(1, numel(b)-1);
```

Define the amount of audio data to process at one time, and initialize the while loop index.

```
frameSize = 1024;
nIdx = 1;
```

Define the while loop to process the audio data.

```
while nIdx <= maxSamples(1)-frameSize+1
    audio = audioread(fname,[nIdx nIdx+frameSize-1]);
    [y,z] = filter(b,1,audio,z);
    sound(y,fs);
    nIdx = nIdx+frameSize;
end
```

The loop uses explicit indexing and state management, which can be a tedious and error-prone approach. You must have detailed knowledge of the states, such as, sizes and data types. Another issue with this MATLAB-only code is that the sound function is not designed to run in real time. The resulting audio is choppy and barely audible.

Process Audio Data Using System Objects

This example shows how to write System objects code for reading audio data.

The code uses System objects from the DSP System Toolbox™ software to read audio data from a file, filter it, and then play the filtered audio data. This code produces the same result as the MATLAB® code shown previously, allowing you to compare approaches.

Locate source audio file.

```
fname = "speech_dft_8kHz.wav";
```

Define the System object to read the file.

```
audioIn = dsp.AudioFileReader(fname, 'OutputDataType', 'single');
```

Define the System object to filter the data.

```
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
```

Define the System object to play the filtered audio data.

```
audioOut = audioDeviceWriter('SampleRate',audioIn.SampleRate);
```

Define the while loop to process the audio data.

```
while ~isDone(audioIn)
    audio = audioIn();    % Read audio source file
    y = filtLP(audio);    % Filter the data
    audioOut(y);         % Play the filtered data
end
```

This System objects code avoids the issues present in the MATLAB-only code. Without requiring explicit indexing, the file reader object manages the data frame sizes while the filter manages the states. The audio device writer object plays each audio frame as it is processed.

System Design in MATLAB Using System Objects

In this section...

“System Design and Simulation in MATLAB” on page 33-9

“Create Individual Components” on page 33-10

“Configure Components” on page 33-10

“Create and Configure Components at the Same Time” on page 33-11

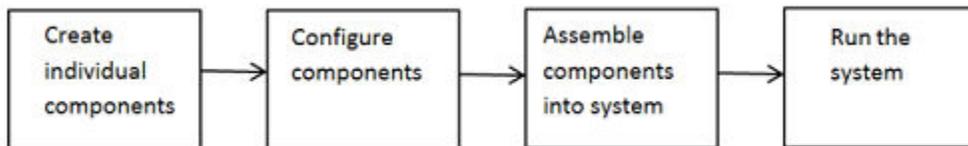
“Assemble Components Into System” on page 33-12

“Run Your System” on page 33-12

“Reconfiguring Objects” on page 33-13

System Design and Simulation in MATLAB

System objects allow you to design and simulate your system in MATLAB. You use System objects in MATLAB as shown in this diagram.



- 1 “Create Individual Components” on page 33-10 — Create the System objects to use in your system. “Create Individual Components” on page 33-10. In addition to the System objects provided with System Toolboxes, you can also create your own System objects. See “Create System Objects”.
- 2 “Configure Components” on page 33-10 — If necessary, change the objects’ property values to model your particular system. All System object properties have default values that you may be able to use without changing them. See “Configure Components” on page 33-10.
- 3 “Assemble Components Into System” on page 33-12 — Write a MATLAB program that includes those System objects, connecting them using MATLAB variables as inputs and outputs to simulate your system. See “Connecting System Objects” on page 33-12.

- 4 “Run Your System” on page 33-12 — Run your program. You can change tunable properties while your system is running. See “Run Your System” on page 33-12 and “Reconfiguring Objects” on page 33-13.

Create Individual Components

The example in this section shows how to use System objects that are predefined in the software. If you use a function to create and use a System object, specify the object creation using conditional code. Conditionalizing the creation prevents errors if that function is called within a loop. You can also create your own System objects, see “Create System Objects”.

This section shows how to set up your system using predefined components from DSP System Toolbox™ and Audio System Toolbox™:

- `dsp.AudioFileReader` — Read the file of audio data
- `dsp.FIRFilter` — Filter the audio data
- `audioDeviceWriter` — Play the filtered audio data

First, create the component objects, using default property settings.

```
audioIn = dsp.AudioFileReader;  
filtLP = dsp.FIRFilter;  
audioOut = audioDeviceWriter;
```

Configure Components

When to Configure Components

If you did not set an object's properties when you created it and do not want to use default values, you must explicitly set those properties. Some properties allow you to change their values while your system is running. See “Reconfiguring Objects” on page 33-13 for information.

Most properties are independent of each other. However, some System object properties enable or disable another property or limit the values of another property. To avoid errors or warnings, you should set the controlling property before setting the dependent property.

Display Component Property Values

To display the current property values for an object, type that object's handle name at the command line (such as `audioIn`). To display the value of a specific property, type `objecthandle.propertyname` (such as `audioIn.FileName`).

Configure Component Property Values

This section shows how to configure the components for your system by setting the component objects' properties.

Use this procedure if you have created your components separately from configuring them. You can also create and configure your components at the same time, as described in a later example.

For the file reader object, specify the file to read and set the output data type.

For the filter object, specify the filter numerator coefficients using the `fir1` function, which specifies the lowpass filter order and the cutoff frequency.

For the audio device writer object, specify the sample rate. In this case, use the same sample rate as the input data.

```
audioIn.FileName = "speech_dft_8kHz.wav";  
audioIn.OutputDataType = "single";  
filtLP.Numerator = fir1(160,.15);  
audioOut.SampleRate = audioIn.SampleRate;
```

Create and Configure Components at the Same Time

This example shows how to create your System object components and configure the desired properties at the same time. Specify each property with a 'Name', Value argument pair.

Create the file reader object, specify the file to read, and set the output data type.

```
audioIn = dsp.AudioFileReader("speech_dft_8kHz.wav",...  
                             'OutputDataType',"single");
```

Create the filter object and specify the filter numerator using the `fir1` function. Specify the lowpass filter order and the cutoff frequency of the `fir1` function.

```
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
```

Create the audio player object and set the sample rate to the same rate as the input data.

```
audioOut = audioDeviceWriter('SampleRate',audioIn.SampleRate);
```

Assemble Components Into System

Connecting System Objects

After you have determined the components you need and have created and configured your System objects, assemble your system. You use the System objects like other MATLAB variables and include them in MATLAB code. You can pass MATLAB variables into and out of System objects.

The main difference between using System objects and using functions is that System objects use a two-step process. First you create the object and set its parameters and then, you run the object. Running the object initializes it and controls the data flow and state management of your system. You typically call a System object within a code loop.

You use the output from an object as the input to another object. For some System objects, you can use properties of those objects to change the inputs or outputs. To verify that the appropriate number of inputs and outputs are being used, you can use `nargin` and `nargout` on any System object. For information on all available System object functions, see “System Object Functions” on page 33-4.

Connect Components in a System

This section shows how to connect the components together to read, filter, and play a file of audio data. The while loop uses the `isDone` function to read through the entire file.

```
while ~isDone(audioIn)
    audio = audioIn();      % Read audio source file
    y = filtLP(audio);     % Filter the data
    audioOut(y);           % Play the filtered data
end
```

Run Your System

Run your code by either typing directly at the command line or running a file containing your program. When you run the code for your system, data is processed through your objects.

What You Cannot Change While Your System Is Running

The first call to a System object initializes and runs the object. When a System object has started processing data, you cannot change nontunable properties.

Depending on the System object, additional specifications might also be restricted:

- Input size
- Input complexity
- Input data type
- Tunable property data types
- Discrete state data types

If the System object author has restricted these specifications, you get an error if you try to change them while the System object is in use.

Reconfiguring Objects

Change Properties

When a System object has started processing data, you cannot change nontunable properties. You can use `isLocked` on any System object to verify whether the object is processing data. When processing is complete, you can use the `release` function to release resources and allow changes to nontunable properties.

Some object properties are tunable, which enables you to change them even if the object is in use. Most System object properties are nontunable. Refer to the object's reference page to determine whether an individual property is tunable.

Change Input Complexity, Dimensions, or Data Type

During object usage, after you have called the algorithm, some System objects do not allow changes in input complexity, size, or data type. If the System object restricts these specifications, you can call `release` to change these specifications. Calling `release` also resets other aspects of the System object, such as states and Discrete states.

Change a Tunable Property in Your System

This example shows how to change the filter type to a high-pass filter as the code is running by modifying the `Numerator` property of the filter object. The change takes effect the next time the object is called.

```
reset(audioIn);% Reset audio file
Wn = [0.05,0.1,0.15,0.2];
for x=1:4000
    Wn_X = ceil(x/1000);
    filtLP.Numerator = fir1(160,Wn(Wn_X),'high');
    audio = audioIn();    % Read audio source file
    y = filtLP(audio);    % Filter the data
    audioOut(y);         % Play the filtered data
end
```

Define Basic System Objects

This example shows how to create a basic System object that increments a number by one. The class definition file used in the example contains the minimum elements required to define a System object.

Create System Object Class

You can create and edit a MAT-file or use the MATLAB Editor to create your System object. This example describes how to use the **New** menu in the MATLAB Editor.

- 1 In MATLAB, on the Editor tab, select **New > System Object > Basic**. A simple System object template opens.
- 2 Subclass your object from `matlab.System`. Replace `Untitled` with `AddOne` in the first line of your file.

```
classdef AddOne < matlab.System
```

System objects are composed of a base class, `matlab.System` and may include one or more mixin classes. You specify the base class and mixin classes on the first line of your class definition file.

- 3 Save the file and name it `AddOne.m`.

Define Algorithm

The `stepImpl` method contains the algorithm to execute when you run your object. Define this method so that it contains the actions you want the System object to perform.

- 1 In the basic System object you created, inspect the `stepImpl` method template.

```
methods (Access = protected)
    function y = stepImpl(obj,u)
        % Implement algorithm. Calculate y as a function of input u and
        % discrete states.
        y = u;
    end
end
```

The `stepImpl` method access is always set to `protected` because it is an internal method that users do not directly call or run.

All methods, except static methods, require the System object handle as the first input argument. The default value, inserted by MATLAB Editor, is `obj`. You can use any name for your System object handle.

By default, the number of inputs and outputs are both one. Inputs and outputs can be added using **Inputs/Outputs**. You can also use a variable number of inputs or outputs, see “Change the Number of Inputs” on page 33-18.

Alternatively, if you create your System object by editing a MAT-file, you can add the `stepImpl` method using **Insert Method > Implement algorithm**.

- 2 Change the computation in the `stepImpl` method to add 1 to the value of `u`.

methods (Access = protected)

```
function y = stepImpl(~,u)
    y = u + 1;
end
```

Tip Instead of passing in the object handle, you can use the tilde (~) to indicate that the object handle is not used in the function. Using the tilde instead of an object handle prevents warnings about unused variables.

- 3 Remove unused methods that are included by default in the basic template.

You can modify these methods to add more System object actions and properties. You can also make no changes, and the System object still operates as intended.

The class definition file now has all the code necessary for this System object.

```
classdef AddOne < matlab.System
% ADDONE Compute an output value one greater than the input value

% All methods occur inside a methods declaration.
% The stepImpl method has protected access
methods (Access = protected)

    function y = stepImpl(~,u)
        y = u + 1;
    end
end
end
```

See Also

`getNumInputsImpl` | `getNumOutputsImpl` | `matlab.System` | `stepImpl`

Related Examples

- “Change the Number of Inputs” on page 33-18
- “System Design and Simulation in MATLAB” on page 33-9

Change the Number of Inputs

This example shows how to set the number of inputs for a System object™ with and without using `getNumInputsImpl`.

If you have a variable number of inputs or outputs and you intend to use the System object in Simulink®, you must include the `getNumInputsImpl` or `getNumOutputsImpl` method in your class definition.

These examples show modifications for the number of inputs. If you want to change the number of outputs, the same principles apply.

As with all System object `Impl` methods, you always set the `getNumInputsImpl` and `getNumOutputsImpl` method's access to `protected` because they are internal methods that are never called directly.

Allow up to Three Inputs

This example shows how to write a System object that allows the number of inputs to vary.

Update the `stepImpl` method to accept up to three inputs by adding code to handle one, two, or three inputs. If you are only using this System object in MATLAB, `getNumInputsImpl` and `getNumOutputsImpl` are not required.

Full Class Definition

```
classdef AddTogether < matlab.System
    % Add inputs together

    methods (Access = protected)
        function y = stepImpl(~,x1,x2,x3)
            switch nargin
                case 2
                    y = x1;
                case 3
                    y = x1 + x2;
                case 4
                    y = x1 + x2 + x3;
                otherwise
                    y = [];
            end
        end
    end
end
```

```

        end
    end
end

```

Run this System object with one, two, and three inputs.

```

addObj = AddTogether;
addObj(2)

```

```

ans =

```

```

    2

```

```

addObj(2,3)

```

```

ans =

```

```

    5

```

```

addObj(2,3,4)

```

```

ans =

```

```

    9

```

Control the Number of Inputs and Outputs with a Property

This example shows how to write a System object that allows changes to the number of inputs and outputs before running the object. Use this method when your System object will be included in Simulink:

- Add a nontunable property `NumInputs` to control the number of inputs.
- Implement the associated `getNumInputs` and `getNumOutputsImpl` methods to specify the number of inputs.

Full Class Definition

```

classdef AddTogether2 < matlab.System

```

```
% Add inputs together. The number of inputs is controlled by the
% nontunable property |NumInputs|.

properties (Nontunable)
    NumInputs = 3; % Default value
end
methods (Access = protected)
    function y = stepImpl(obj,x1,x2,x3)
        switch obj.NumInputs
            case 1
                y = x1;
            case 2
                y = x1 + x2;
            case 3
                y = x1 + x2 + x3;
            otherwise
                y = [];
        end
    end
    function validatePropertiesImpl(obj)
        if ((obj.NumInputs < 1) || ...
            (obj.NumInputs > 3))
            error("Only 1, 2, or 3 inputs allowed.");
        end
    end
    function numIn = getNumInputsImpl(obj)
        numIn = obj.NumInputs;
    end
end
```

Run this System object with one, two, and three inputs.

```
addObj = AddTogether2;
addObj.NumInputs = 1;
addObj(2)
```

```
ans =
```

```
2
```

```
release(addObj);  
addObj.NumInputs = 2;  
addObj(2,3)
```

```
ans =
```

```
5
```

```
release(addObj);  
addObj.NumInputs = 3;  
addObj(2,3,4)
```

```
ans =
```

```
9
```

See Also

[getNumInputsImpl](#) | [getNumOutputsImpl](#)

Related Examples

- “Validate Property and Input Values” on page 33-22
- “Define Basic System Objects” on page 33-15
- “Using ~ as an Input Argument in Method Definitions” on page 33-66

Validate Property and Input Values

This example shows how to verify that the user's inputs and property values are valid.

Validate Properties

This example shows how to validate the value of a single property using `set.PropertyName` syntax. In this case, the *PropertyName* is `Increment`. Validate the value of two interdependent properties using the `validatePropertiesImpl` method. In this case, the `UseIncrement` property value must be true and the `WrapValue` property value must be less than the `Increment` property value.

```
methods
    % Validate the properties of the object
    function set.Increment(obj,val)
        if val >= 10
            error("The increment value must be less than 10");
        end
        obj.Increment = val;
    end
end
methods (Access = protected)
    function validatePropertiesImpl(obj)
        if obj.UseIncrement && obj.WrapValue > obj.Increment
            error("Wrap value must be less than increment value");
        end
    end
end
end
```

Validate Inputs

This example shows how to validate that the first input is a numeric value.

```
methods (Access = protected)
    function validateInputsImpl(~,x)
        if ~isnumeric(x)
            error("Input must be numeric");
        end
    end
end
end
```

Complete Class Definition File with Property and Input Validation

```
classdef AddOne < matlab.System
    % ADDONE Compute an output value by incrementing the input value
```

```
% All properties occur inside a properties declaration.
% These properties have public access (the default)
properties (Logical)
    UseIncrement = true
end

properties (PositiveInteger)
    Increment = 1
    WrapValue = 10
end

methods
    % Validate the properties of the object
    function set.Increment(obj,val)
        if val >= 10
            error("The increment value must be less than 10");
        end
        obj.Increment = val;
    end
end

methods (Access = protected)
    function validatePropertiesImpl(obj)
        if obj.UseIncrement && obj.WrapValue > obj.Increment
            error("Wrap value must be less than increment value");
        end
    end

    % Validate the inputs to the object
    function validateInputsImpl(~,x)
        if ~isnumeric(x)
            error("Input must be numeric");
        end
    end

    function out = stepImpl(obj,in)
        if obj.UseIncrement
            out = in + obj.Increment;
        else
            out = in + 1;
        end
    end
end
```

```
end  
end
```

See Also

`validateInputsImpl` | `validatePropertiesImpl`

Related Examples

- “Define Basic System Objects” on page 33-15
- “Change Input Complexity, Dimensions, or Data Type” on page 33-13
- “Summary of Call Sequence” on page 33-59
- “Property Set Methods”
- “Using ~ as an Input Argument in Method Definitions” on page 33-66

Initialize Properties and Setup One-Time Calculations

This example shows how to write code to initialize and set up a System object.

In this example, you allocate file resources by opening the file so the System object can write to that file. You do these initialization tasks one time during setup, rather than every time you run the object.

Define Public Properties to Initialize

In this example, you define the public `Filename` property and specify the value of that property as the nontunable character vector, `default.bin`. Users cannot change nontunable properties after the `setup` method has been called.

```
properties (Nontunable)
    Filename = "default.bin"
end
```

Define Private Properties to Initialize

Users cannot access private properties directly, but only through methods of the System object. In this example, you define the `pFileID` property as a private property. You also define this property as hidden to indicate it is an internal property that never displays to the user.

```
properties (Hidden,Access = private)
    pFileID;
end
```

Define Setup

You use the `setupImpl` method to perform setup and initialization tasks. You should include code in the `setupImpl` method that you want to execute one time only. The `setupImpl` method is called once the first time you run the object. In this example, you allocate file resources by opening the file for writing binary data.

```
methods
    function setupImpl(obj)
        obj.pFileID = fopen(obj.Filename,"wb");
        if obj.pFileID < 0
            error("Opening the file failed");
        end
    end
end
```

Although not part of setup, you should close files when your code is done using them. You use the `releaseImpl` method to release resources.

Complete Class Definition File with Initialization and Setup

```
classdef MyFile < matlab.System
% MyFile write numbers to a file

    % These properties are nontunable. They cannot be changed
    % after the setup method has been called or the object
    % is running.
    properties (Nontunable)
        Filename = "default.bin" % the name of the file to create
    end

    % These properties are private. Customers can only access
    % these properties through methods on this object
    properties (Hidden,Access = private)
        pFileID; % The identifier of the file to open
    end

    methods (Access = protected)
        % In setup allocate any resources, which in this case
        % means opening the file.
        function setupImpl(obj)
            obj.pFileID = fopen(obj.Filename,'wb');
            if obj.pFileID < 0
                error("Opening the file failed");
            end
        end

        % This System
        % object™ writes the input to the file.
        function stepImpl(obj,data)
            fwrite(obj.pFileID,data);
        end

        % Use release to close the file to prevent the
        % file handle from being left open.
        function releaseImpl(obj)
            fclose(obj.pFileID);
        end
    end
end
```

```
    end  
end
```

See Also

`matlab.System` `Constructor` | `releaseImpl` | `setupImpl` | `stepImpl`

Related Examples

- “Release System Object Resources” on page 33-30
- “Define Property Attributes” on page 33-32
- “Summary of Call Sequence” on page 33-59

Set Property Values at Construction Time

This example shows how to define a System object constructor and allow it to accept name-value property pairs as input.

Set Properties to Use Name-Value Pair Input

Define the System object constructor, which is a method that has the same name as the class (MyFile in this example). Within that method, you use the `setProperties` method to make all public properties available for input when the user constructs the object. `nargin` is a MATLAB function that determines the number of input arguments. `varargin` indicates all of the object's public properties.

```
methods
    function obj = MyFile(varargin)
        setProperties(obj,nargin,varargin{:});
    end
end
```

Complete Class Definition File with Constructor Setup

```
classdef MyFile < matlab.System
% MyFile write numbers to a file

    % These properties are nontunable. They cannot be changed
    % after the setup method has been called or while the
    % object is running.
    properties (Nontunable)
        Filename = "default.bin" % the name of the file to create
        Access = 'wb' % The file access character vector (write, binary)
    end

    % These properties are private. Customers can only access
    % these properties through methods on this object
    properties (Hidden,Access = private)
        pFileID; % The identifier of the file to open
    end

    methods
        % You call setProperties in the constructor to let
        % a user specify public properties of object as
        % name-value pairs.
        function obj = MyFile(varargin)
            setProperties(obj,nargin,varargin{:});
        end
    end
end
```

```
        end
    end

    methods (Access = protected)
        % In setup allocate any resources, which in this case is
        % opening the file.
        function setupImpl(obj)
            obj.pFileID = fopen(obj.Filename,obj.Access);
            if obj.pFileID < 0
                error("Opening the file failed");
            end
        end
    end

    % This System
    object™ writes the input to the file.
    function stepImpl(obj,data)
        fwrite(obj.pFileID,data);
    end

    % Use release to close the file to prevent the
    % file handle from being left open.
    function releaseImpl(obj)
        fclose(obj.pFileID);
    end
end
end
```

See Also

nargin | setProperties

Related Examples

- “Define Property Attributes” on page 33-32
- “Release System Object Resources” on page 33-30

Reset Algorithm and Release Resources

In this section...

“Reset Algorithm State” on page 33-30

“Release System Object Resources” on page 33-30

Reset Algorithm State

When a user calls `reset` on a System object, the internal `resetImpl` method is called. In this example, `pCount` is an internal counter property of the Counter System object. When a user calls `reset`, `pCount` resets to 0.

```
classdef Counter < matlab.System
% Counter System object that increments a counter

    properties (Access = private)
        pCount
    end

    methods (Access = protected)
        % Increment the counter and return
        % its value as an output
        function c = stepImpl(obj)
            obj.pCount = obj.pCount + 1;
            c = obj.pCount;
        end

        % Reset the counter to zero.
        function resetImpl(obj)
            obj.pCount = 0;
        end
    end
end
```

Release System Object Resources

When a user calls `release` on a System object, the internal `releaseImpl` method is called. This example shows how to implement the method that releases resources allocated and used by the System object. These resources include allocated memory and files used for reading or writing.

This method allows you to clear the axes on the Whiteboard figure window while keeping the figure open.

```
function releaseImpl(obj)
    cla(Whiteboard.getWhiteboard());
    hold on
end
```

For a complete definition of the `Whiteboard System` object, see “Create a Whiteboard System object” on page 33-39.

See Also

`releaseImpl` | `resetImpl`

More About

- “Summary of Call Sequence” on page 33-59
- “Initialize Properties and Setup One-Time Calculations” on page 33-25

Define Property Attributes

This example shows how to specify property attributes.

Property attributes, which add details to a property, provide a layer of control to your properties. In addition to the MATLAB property attributes, System objects can use these three additional attributes: `Nontunable`, `Logical`, and `PositiveInteger`. To specify multiple attributes, separate them with commas.

Specify Property as Nontunable

Use the `Nontunable` attribute for a property when the algorithm depends on the value being constant once data processing starts. Defining a property as nontunable may improve the efficiency of your algorithm by removing the need to check for or react to values that change. For code generation, defining a property as nontunable allows the memory associated with that property to be optimized. You should define all properties that affect the number of input or output ports as nontunable.

Users of the System object can only change nontunable properties before calling the object or after calling the `release` function. In this example, you define the `InitialValue` property as nontunable and set its value to 0.

```
properties (Nontunable)
    InitialValue = 0;
end
```

Specify Property as Logical

Logical properties have the value `true` or `false`. System object users can enter 1 or 0 or any value that can be converted to a logical. The value, however, displays as `true` or `false`. You can use sparse logical values, but they must be scalar values. In this example, the `Increment` property indicates whether to increase the counter. By default, `Increment` is a tunable property. The following restrictions apply to a property with the `Logical` attribute,

- Cannot also be `Dependent` or `PositiveInteger`
- Default value must be `true` or `false`. You cannot use 1 or 0 as a default value.

```
properties (Logical)
    Increment = true;
end
```

Specify Property as Positive Integer

In this example, the private property `MaxValue` is constrained to accept only real, positive integers. You cannot use sparse values. The following restriction applies to a property with the `PositiveInteger` attribute,

- Cannot also be `Dependent` or `Logical`

```
properties (PositiveInteger)
    MaxValue
end
```

Specify Property as DiscreteState

If your algorithm uses properties that hold state, you can assign those properties the `DiscreteState` attribute. Properties with this attribute display their state values via the `getDiscreteStateImpl` when users call `getDiscreteState`. The following restrictions apply to a property with the `DiscreteState` attribute,

- Numeric, logical, or `fi` value, but not a scaled double `fi` value
- Does not have any of these attributes: `Nontunable`, `Dependent`, `Abstract`, `Constant`.
- No default value
- Not publicly settable
- `GetAccess = Public` by default
- Value set only via the `setupImpl` when the `System` object is called. (See “Summary of Call Sequence” on page 33-59)

In this example, you define the `Count` property as a discrete state:

```
properties (DiscreteState)
    Count;
end
```

Complete Class Definition File with Property Attributes

```
classdef Counter < matlab.System
% Counter Increment a counter to a maximum value

    % These properties are nontunable. They cannot be changed
    % after the setup method has been called or while the
```

```
% object is running.
properties (Nontunable)
    % The initial value of the counter
    InitialValue = 0
end
properties (Nontunable, PositiveInteger)
    % The maximum value of the counter
    MaxValue = 3
end

properties (Logical)
    % Whether to increment the counter
    Increment = true
end

properties (DiscreteState)
    % Count state variable
    Count
end

methods (Access = protected)
    % Increment the counter and return its value
    % as an output

    function c = stepImpl(obj)
        if obj.Increment && (obj.Count < obj.MaxValue)
            obj.Count = obj.Count + 1;
        else
            disp(['Max count, ' num2str(obj.MaxValue) ' ,reached'])
        end
        c = obj.Count;
    end

    % Setup the Count state variable
    function setupImpl(obj)
        obj.Count = 0;
    end

    % Reset the counter to one.
    function resetImpl(obj)
        obj.Count = obj.InitialValue;
    end
end
```

end
end

See Also

More About

- “Class Attributes”
- “Property Attributes”
- “What You Cannot Change While Your System Is Running” on page 33-13
- “Summary of Call Sequence” on page 33-59

Hide Inactive Properties

To display only active System object properties, use the `isInactivePropertyImpl` method. This method specifies whether a property is inactive. An *inactive property* is a property that does not affect the System object because of the value of other properties. When you pass the `isInactiveProperty` method a property and the method returns `true`, then that property is inactive and does not display when the `disp` function is called.

Specify Inactive Property

This example uses the `isInactiveProperty` method to check the value a dependant property. For this System object, the `InitialValue` property is not relevant if the `UseRandomInitialValue` property is set to `true`. This `isInactiveProperty` method checks for that situation and if `UseRandomInitialValue` is `true`, returns `true` to hide the inactive `InitialValue` property.

```
methods (Access = protected)
    function flag = isInactivePropertyImpl(obj,propertyName)
        if strcmp(propertyName,'InitialValue')
            flag = obj.UseRandomInitialValue;
        else
            flag = false;
        end
    end
end
```

Complete Class Definition File with Inactive Properties Method

```
classdef Counter < matlab.System
    % Counter Increment a counter

    % These properties are nontunable. They cannot be changed
    % after the setup method has been called or when the
    % object is running.
    properties (Nontunable)
        % Allow the user to set the initial value
        UseRandomInitialValue = true
        InitialValue = 0
    end
```

```
% The private count variable, which is tunable by default
properties (Access = private)
    pCount
end

methods (Access = protected)
    % Increment the counter and return its value
    % as an output
    function c = stepImpl(obj)
        obj.pCount = obj.pCount + 1;
        c = obj.pCount;
    end

    % Reset the counter to either a random value or the initial
    % value.
    function resetImpl(obj)
        if obj.UseRandomInitialValue
            obj.pCount = rand();
        else
            obj.pCount = obj.InitialValue;
        end
    end
end

% This method controls visibility of the object's properties
function flag = isInactivePropertyImpl(obj,propertyName)
    if strcmp(propertyName,'InitialValue')
        flag = obj.UseRandomInitialValue;
    else
        flag = false;
    end
end
end
end
```

See Also

`isInactivePropertyImpl`

Limit Property Values to Finite List

When you want to create a System object property with a limited set of acceptable values, you use enumerations. To use enumerated data in a System object in MATLAB or Simulink, you refer to them in your System object class definition and define your enumerated class in a separate class definition file.

Enumerations can derive from any integer type smaller than or equal to an `int32`. For example,

```
classdef Bearing < uint8
    enumeration
        North (0)
        East (90)
        South (180)
        West (270)
    end
end
```

For a System object that is used in MATLAB only, see “Limit Property Values to Finite List” on page 33-38 and “Enumerations”.

For a System object that is used in a MATLAB System block in Simulink, see “Use Enumerated Data in Simulink Models” (Simulink).

To create an enumerated property, you need:

- A System object property set to the enumeration class.
- The associated enumeration class definition that defines all possible values for the property.

Enumerations can also derive from `Simulink.IntEnumType`. You use this type of enumeration to add attributes, such as custom headers, to the input or output of the MATLAB System block. See “Use Enumerated Data in Simulink Models” (Simulink).

This example defines a color enumeration property for a System object.

The definition of the enumeration class `ColorValues` is:

```
classdef ColorValues < int32
    enumeration
        blue (0)
```

```

        red (1)
        green (2)
    end
end

```

The `ColorValues` class inherits from `int32` for code generation compatibility. Enumeration values must be valid MATLAB identifiers.

The System object property called `Color` and sets the default value to `blue`. The `(1,1)` defines the size of the `Color` property:

```

properties
    Color (1, 1) ColorValues = ColorValues.blue
end

```

Create a Whiteboard System object

This example shows the class definition of a `Whiteboard System` object and how to use the object. Each time you run the whiteboard object, it draws a line on a whiteboard.

```

type('Whiteboard.m');

classdef Whiteboard < matlab.System
    % Whiteboard Draw lines on a figure window
    %

    properties(Nontunable)
        Color (1, 1) ColorValues = ColorValues.blue
    end

    methods (Access = protected)
        function stepImpl(obj)
            h = Whiteboard.getWhiteboard();
            plot(h, ...
                randn([2,1]), randn([2,1]), ...
                'Color',char(obj.Color));
        end

        function releaseImpl(obj)
            cla(Whiteboard.getWhiteboard());
            hold on
        end
    end
end

```

```
        methods (Static)
            function a = getWhiteboard()
                h = findobj('tag','whiteboard');
                if isempty(h)
                    h = figure('tag','whiteboard');
                    hold on
                end
                a = gca;
            end
        end
    end
end
```

Construct the System object.

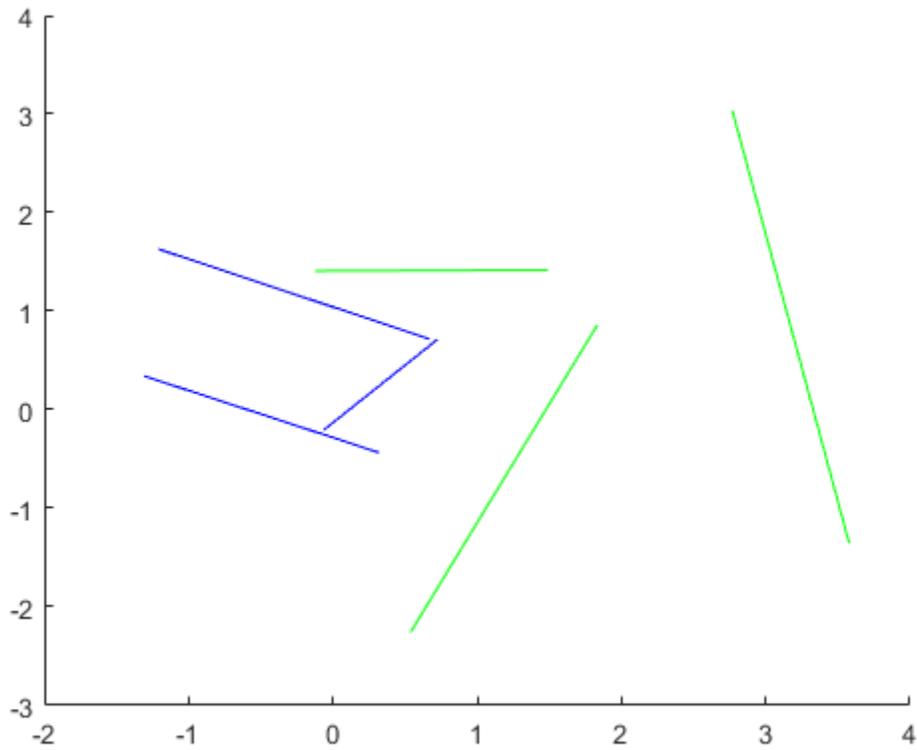
```
greenInk = Whiteboard;
blueInk = Whiteboard;
```

Change the color.

```
greenInk.Color = 'green';
blueInk.Color = 'blue';
```

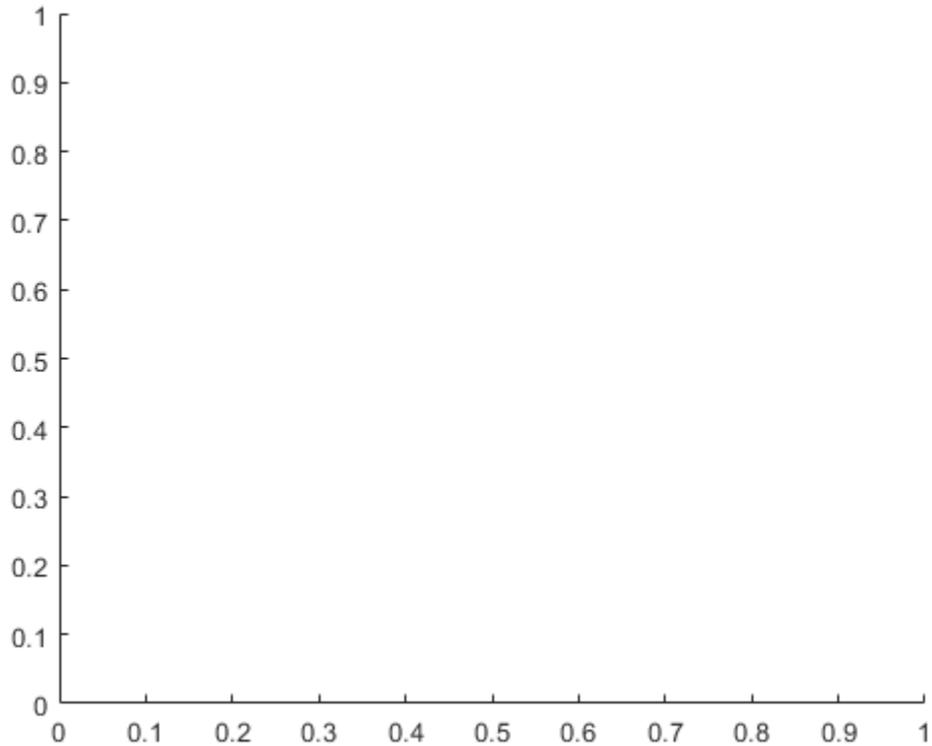
Draw a few lines.

```
for i=1:3
    greenInk();
    blueInk();
end
```



Clear the whiteboard

```
release(greenInk);  
release(blueInk);
```



See Also

Related Examples

- “Enumerations”
- “Code Generation for Enumerations” (MATLAB Coder)

Process Tuned Properties

This example shows how to specify the action to take when a tunable property value changes during simulation.

The `processTunedPropertiesImpl` method is useful for managing actions to prevent duplication. In many cases, changing one of multiple interdependent properties causes an action. With the `processTunedPropertiesImpl` method, you can control when that action is taken so it is not repeated unnecessarily.

Control When a Lookup Table Is Generated

This example of `processTunedPropertiesImpl` causes the `pLookupTable` to be regenerated when either the `NumNotes` or `MiddleC` property changes.

```
methods (Access = protected)
    function processTunedPropertiesImpl(obj)
        propChange = isChangedProperty(obj, 'NumNotes') || ...
            isChangedProperty(obj, 'MiddleC')
        if propChange
            obj.pLookupTable = obj.MiddleC * ...
                (1+log(1:obj.NumNotes)/log(12));
        end
    endend
```

Complete Class Definition File with Tuned Property Processing

```
classdef TuningFork < matlab.System
    % TuningFork Illustrate the processing of tuned parameters
    %

    properties
        MiddleC = 440
        NumNotes = 12
    end

    properties (Access = private)
        pLookupTable
    end

    methods (Access = protected)
        function resetImpl(obj)
            obj.MiddleC = 440;
            obj.pLookupTable = obj.MiddleC * ...
```

```
        (1+log(1:obj.NumNotes)/log(12));
    end

    function hz = stepImpl(obj,noteShift)
        % A noteShift value of 1 corresponds to obj.MiddleC
        hz = obj.pLookupTable(noteShift);
    end

    function processTunedPropertiesImpl(obj)
        propChange = isChangedProperty(obj,'NumNotes')||...
            isChangedProperty(obj,'MiddleC')
        if propChange
            obj.pLookupTable = obj.MiddleC *...
                (1+log(1:obj.NumNotes)/log(12));
        end
    end
end
```

See Also

processTunedPropertiesImpl

Define Composite System Objects

This example shows how to define System objects that include other System objects. Define a bandpass filter System object from separate highpass and lowpass filter System objects.

Store System Objects in Properties

To define a System object from other System objects, store those other objects in your class definition file as properties. In this example, the highpass and lowpass filters are the separate System objects defined in their own class-definition files.

```
properties (Access = private)
    % Properties that hold filter System objects
    pLowpass
    pHighpass
end
```

Complete Class Definition File of Bandpass Filter Composite System Object

```
classdef BandpassFIRFilter < matlab.System
    % Implements a bandpass filter using a cascade of eighth-order lowpass
    % and eighth-order highpass FIR filters.
```

```
    properties (Access = private)
        % Properties that hold filter System objects
        pLowpass
        pHighpass
    end

    methods (Access = protected)
        function setupImpl(obj)
            % Setup composite object from constituent objects
            obj.pLowpass = LowpassFIRFilter;
            obj.pHighpass = HighpassFIRFilter;
        end

        function yHigh = stepImpl(obj,u)
            yLow = obj.pLowpass(u);
            yHigh = obj.pHighpass(yLow);
        end

        function resetImpl(obj)
            reset(obj.pLowpass);
        end
    end
end
```

```
        reset(obj.pHighpass);
    end
end
end
```

Class Definition File for Lowpass FIR Component of Bandpass Filter

```
classdef LowpassFIRFilter < matlab.System
% Implements eighth-order lowpass FIR filter with 0.6pi cutoff

    properties (Nontunable)
        % Filter coefficients
        Numerator = [0.006, -0.0133, -0.05, 0.26, 0.6, 0.26, -0.05, -0.0133, 0.006];
    end

    properties (DiscreteState)
        State
    end

    methods (Access = protected)
        function setupImpl(obj)
            obj.State = zeros(length(obj.Numerator)-1,1);
        end

        function y = stepImpl(obj,u)
            [y,obj.State] = filter(obj.Numerator,1,u,obj.State);
        end

        function resetImpl(obj)
            obj.State = zeros(length(obj.Numerator)-1,1);
        end
    end
end
```

Class Definition File for Highpass FIR Component of Bandpass Filter

```
classdef HighpassFIRFilter < matlab.System
% Implements eighth-order highpass FIR filter with 0.4pi cutoff

    properties (Nontunable)
        % Filter coefficients
        Numerator = [0.006, 0.0133, -0.05, -0.26, 0.6, -0.26, -0.05, 0.0133, 0.006];
    end

    properties (DiscreteState)
```

```
    State
end

methods (Access = protected)
    function setupImpl(obj)
        obj.State = zeros(length(obj.Numerator)-1,1);
    end

    function y = stepImpl(obj,u)
        [y,obj.State] = filter(obj.Numerator,1,u,obj.State);
    end

    function resetImpl(obj)
        obj.State = zeros(length(obj.Numerator)-1,1);
    end
end
end
```

See Also

nargin

Define Finite Source Objects

This example shows how to define a System object that performs a specific number of steps or specific number of reads from a file.

In this section...

“Use the FiniteSource Class and Specify End of the Source” on page 33-48

“Complete Class Definition File with Finite Source” on page 33-48

Use the FiniteSource Class and Specify End of the Source

- 1 Subclass from finite source class.

```
classdef RunTwice < matlab.System & ...  
    matlab.system.mixin.FiniteSource
```

- 2 Specify the end of the source with the `isDoneImpl` method. In this example, the source has two iterations.

```
methods (Access = protected)  
    function bDone = isDoneImpl(obj)  
        bDone = obj.NumSteps==2  
    end
```

Complete Class Definition File with Finite Source

```
classdef RunTwice < matlab.System & ...  
    matlab.system.mixin.FiniteSource  
    % RunTwice System object that runs exactly two times  
    %  
    properties (Access = private)  
        NumSteps  
    end  
  
    methods (Access = protected)  
        function resetImpl(obj)  
            obj.NumSteps = 0;  
        end  
  
        function y = stepImpl(obj)  
            if ~obj.isDone()
```

```
        obj.NumSteps = obj.NumSteps + 1;
        y = obj.NumSteps;
    else
        y = 0;
    end
end

function bDone = isDoneImpl(obj)
    bDone = obj.NumSteps==2;
end
end
end
```

See Also

matlab.system.mixin.FiniteSource

More About

- “Subclassing Multiple Classes”
- “Using ~ as an Input Argument in Method Definitions” on page 33-66

Save and Load System Object

This example shows how to load and save a System object.

Save System Object and Child Object

Define a `saveObjectImpl` method to specify that more than just public properties should be saved when the user saves a System object. Within this method, use the default `saveObjectImpl@matlab.System` to save public properties to the struct, `s`. Use the `saveObject` method to save child objects. Save protected and dependent properties, and finally, if the object has been called and not released, save the object state.

```
methods (Access = protected)
function s = saveObjectImpl(obj)
    s = saveObjectImpl@matlab.System(obj);
    s.child = matlab.System.saveObject(obj.child);
    s.protectedprop = obj.protectedprop;
    s.pdependentprop = obj.pdependentprop;
    if isLocked(obj)
        s.state = obj.state;
    end
end
end
```

Load System Object and Child Object

Define a `loadObjectImpl` method to load a previously saved System object. Within this method, use the `loadObject` to load the child System object, load protected and private properties, load the state if the object was called and not released, and use `loadObjectImpl` from the base class to load public properties.

```
methods (Access = protected)
function loadObjectImpl(obj,s,isInUse)
    obj.child = matlab.System.loadObject(s.child);

    obj.protectedprop = s.protectedprop;
    obj.pdependentprop = s.pdependentprop;

    if isInUse
        obj.state = s.state;
    end
end
```

```

        loadObjectImpl@matlab.System(obj,s,isInUse);
    end
end

```

Complete Class Definition Files with Save and Load

The Counter class definition file sets up an object with a count property. This counter is used in the MySaveLoader class definition file to count the number of child objects.

```

classdef Counter < matlab.System
    properties (DiscreteState)
        Count
    end
    methods (Access=protected)
        function setupImpl(obj, ~)
            obj.Count = 0;
        end
        function y = stepImpl(obj, u)
            if u > 0
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end
    end
end

classdef MySaveLoader < matlab.System

    properties (Access = private)
        child
        pdependentprop = 1
    end

    properties (Access = protected)
        protectedprop = rand;
    end

    properties (DiscreteState = true)
        state
    end

    properties (Dependent)
        dependentprop
    end
end

```

```
methods
function obj = MySaveLoader(varargin)
    obj@matlab.System();
    setProperties(obj,nargin,varargin{:});
end

function set.dependentprop(obj, value)
    obj.pdependentprop = min(value, 5);
end

function value = get.dependentprop(obj)
    value = obj.pdependentprop;
end
end

methods (Access = protected)
function setupImpl(obj)
    obj.state = 42;
    obj.child = Counter;
end
function out = stepImpl(obj,in)
    obj.state = in + obj.state + obj.protectedprop + ...
        obj.pdependentprop;
    out = obj.child(obj.state);
end
end

% Serialization
methods (Access = protected)
function s = saveObjectImpl(obj)
    % Call the base class method
    s = saveObjectImpl@matlab.System(obj);

    % Save the child System objects
    s.child = matlab.System.saveObject(obj.child);

    % Save the protected & private properties
    s.protectedprop = obj.protectedprop;
    s.pdependentprop = obj.pdependentprop;

    % Save the state only if object called and not released
    if isLocked(obj)
        s.state = obj.state;
    end
end
```

```
    end
end

function loadObjectImpl(obj,s,isInUse)
    % Load child System objects
    obj.child = matlab.System.loadObject(s.child);

    % Load protected and private properties
    obj.protectedprop = s.protectedprop;
    obj.pdependentprop = s.pdependentprop;

    % Load the state only if object is in use
    if isInUse
        obj.state = s.state;
    end

    % Call base class method to load public properties
    loadObjectImpl@matlab.System(obj,s,isInUse);
end
end
end
```

See Also

loadObjectImpl | saveObjectImpl

Define System Object Information

This example shows how to define information to display for a System object.

Define System Object Info

You can define your own `info` method to display specific information for your System object. The default `infoImpl` method returns an empty struct. This `infoImpl` method returns detailed information when `info` is called using `info(x, 'details')` or only count information if it is called using `info(x)`.

```
methods (Access = protected)
    function s = infoImpl(obj,varargin)
        if nargin>1 && strcmp('details',varargin(1))
            s = struct('Name','Counter',
                'Properties', struct('CurrentCount',obj.Count, ...
                'Threshold',obj.Threshold));
        else
            s = struct('Count',obj.Count);
        end
    end
end
```

Complete Class Definition File with InfoImpl

```
classdef Counter < matlab.System
    % Counter Count values above a threshold

    properties
        Threshold = 1
    end

    properties (DiscreteState)
        Count
    end

    methods (Access = protected)
        function setupImpl(obj)
            obj.Count = 0;
        end

        function resetImpl(obj)
            obj.Count = 0;
        end
    end
end
```

```
function y = stepImpl(obj,u)
    if (u > obj.Threshold)
        obj.Count = obj.Count + 1;
    end
    y = obj.Count;
end

function s = infoImpl(obj,varargin)
    if nargin>1 && strcmp('details',varargin(1))
        s = struct('Name','Counter',...
            'Properties', struct('CurrentCount', ...
                obj.Count, 'Threshold',obj.Threshold));
    else
        s = struct('Count',obj.Count);
    end
end
end
end
end
```

See Also

infoImpl

Handle Input Specification Changes

This example shows how to control the input specifications for a System object. You can control what happens when an input specification changes.

You can also restrict whether the input complexity, data type, or size can change while the object is in use. Whichever aspects you restrict cannot change until after the user calls release.

React to Input Specification Changes

To modify the System object algorithm or properties when the input changes size, data type, or complexity, implement the `processInputSpecificationChangeImpl` method. Specify actions to take when the input specification changes between calls to the System object.

In this example, `processInputSpecificationChangeImpl` changes the `isComplex` property when either input is complex.

```
properties(Access = private, Logical)
    isComplex = false;
end

methods (Access = protected)
    function processInputSpecificationChangeImpl(obj,input1,input2)
        if(isreal(input1) && isreal(input2))
            obj.isComplex = false;
        else
            obj.isComplex = true;
        end
    end
end
```

Restrict Input Specification Changes

To specify that the input complexity, data type, and size cannot change while the System object is in use, implement the `isInputComplexityMutableImpl`, `isInputDataTypeMutableImpl`, and `isInputSizeMutableImpl` methods to return false. If you want to restrict only some aspects of the System object input, you can include only one or two of these methods.

```

methods (Access = protected)
    function flag = isInputComplexityMutableImpl(~,~)
        flag = false;
    end
    function flag = isInputSizeDataTypeImpl(~,~)
        flag = false;
    end
    function flag = isInputSizeMutableImpl(~,~)
        flag = false;
    end
end

```

Complete Class Definition File

This CounterSystem object restricts all three aspects of the input specification.

```

classdef Counter < matlab.System
    %Counter Count values above a threshold

    properties
        Threshold = 1
    end

    properties (DiscreteState)
        Count
    end

    methods
        function obj = Counter(varargin)
            setProperties(obj,nargin,varargin{:});
        end
    end

    methods (Access=protected)
        function resetImpl(obj)
            obj.Count = 0;
        end

        function y = stepImpl(obj, u1)
            if (any(u1 >= obj.Threshold))
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end
    end
end

```

```
function flag = isInputComplexityMutableImpl(~,~)
    flag = false;
end

function flag = isInputDataTypeMutableImpl(~,~)
    flag = false;
end

function flag = isInputSizeMutableImpl(~,~)
    flag = false;
end
end
end
```

See Also

isInputSizeMutableImpl

Related Examples

- “What You Cannot Change While Your System Is Running” on page 33-13

Summary of Call Sequence

In this section...
“Setup Call Sequence” on page 33-59
“Running the Object or Step Call Sequence” on page 33-60
“Reset Method Call Sequence” on page 33-62
“Release Method Call Sequence” on page 33-62

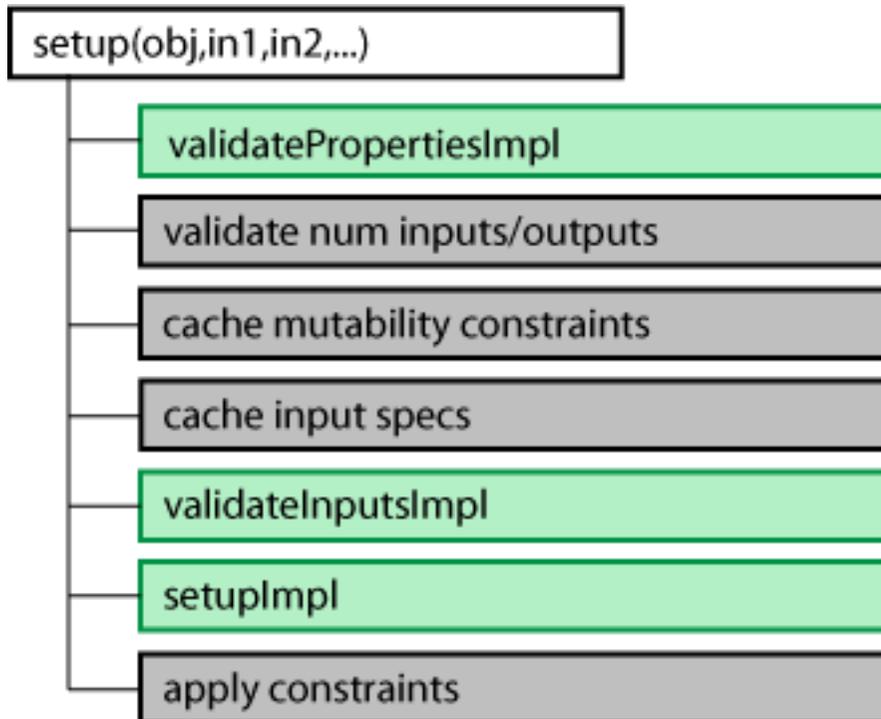
The diagrams show an abstract view of which actions are performed when you call a System object. The background color of each action indicates the type of call.

- Grey background — Internal actions
- Green background — Author-implemented method
- White background — User-accessible functions

If you want a more detailed call sequence, see “Detailed Call Sequence” on page 33-63.

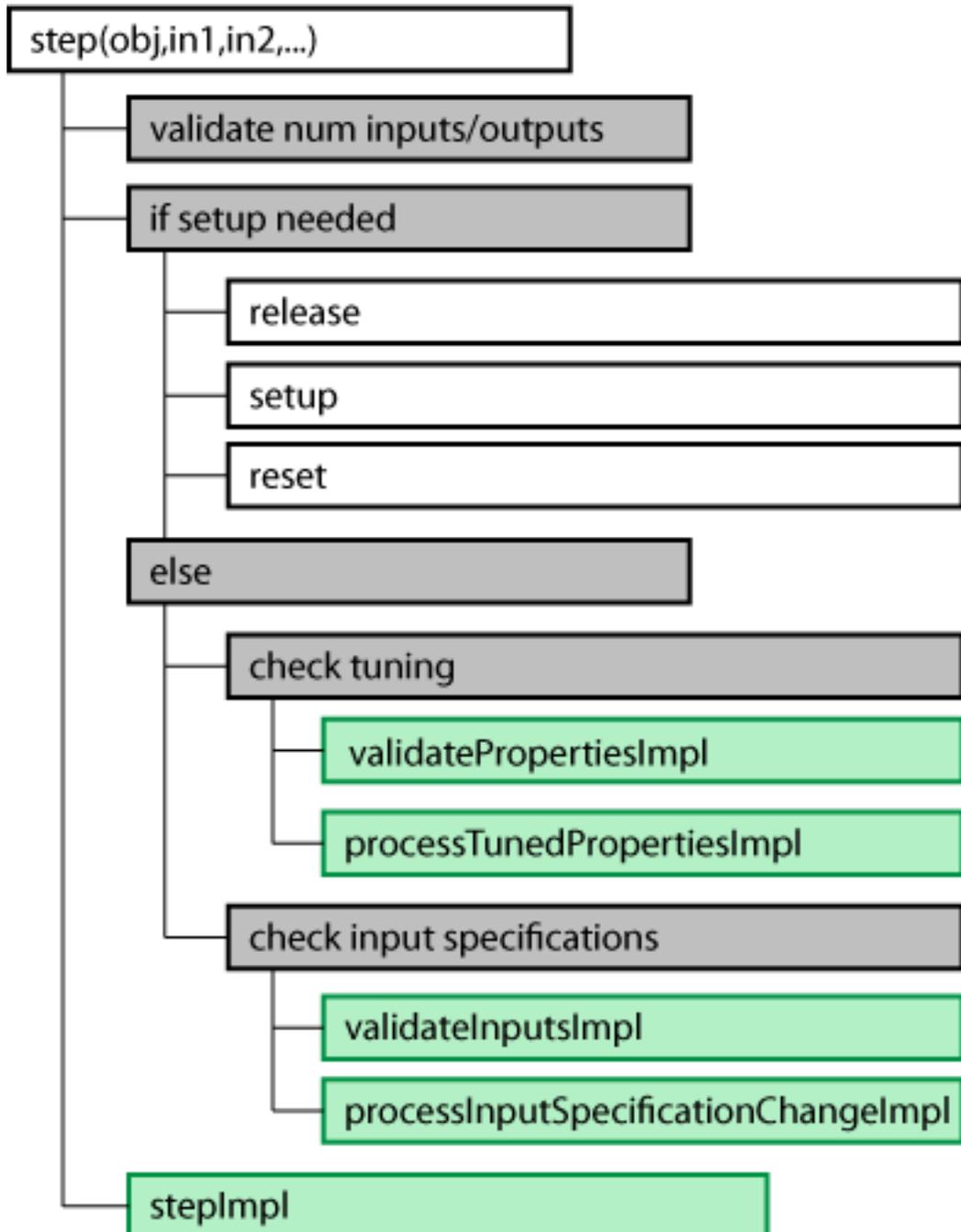
Setup Call Sequence

This hierarchy shows the actions performed when you call the `setup` function.



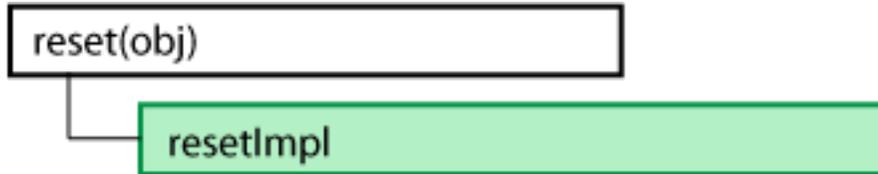
Running the Object or Step Call Sequence

This hierarchy shows the actions performed when you call the `step` function.



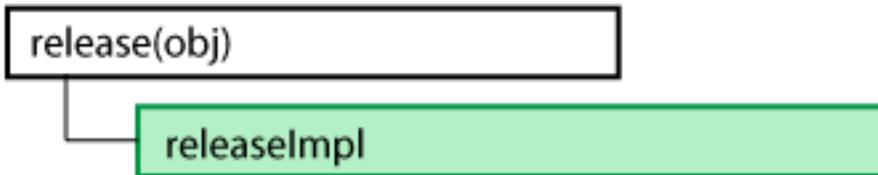
Reset Method Call Sequence

This hierarchy shows the actions performed when you call the `reset` function.



Release Method Call Sequence

This hierarchy shows the actions performed when you call the `release` function.



See Also

`releaseImpl` | `resetImpl` | `setupImpl` | `stepImpl`

Related Examples

- “Release System Object Resources” on page 33-30
- “Reset Algorithm State” on page 33-30
- “Set Property Values at Construction Time” on page 33-28
- “Define Basic System Objects” on page 33-15

Detailed Call Sequence

In this section...

“setup Call Sequence” on page 33-63

“Running the Object or step Call Sequence” on page 33-64

“reset Call Sequence” on page 33-64

“release Call Sequence” on page 33-64

The call sequence diagrams show the order in which internal methods are called when you run the specified method. If your System object does not overwrite a specified method, the default implementation of that method is used.

If you want a more abstract view of the method calls, see “Summary of Call Sequence” on page 33-59.

setup Call Sequence

When you run a System object for the first time, `setup` is called to perform one-time set up tasks. This sequence of methods is called:

- 1 If the System object is not in use, `release` on page 33-64
- 2 `validatePropertiesImpl`
- 3 `isDiscreteStateSpecificationMutableImpl`
- 4 `isInputDataTypeMutableImpl`
- 5 `isInputComplexityMutableImpl`
- 6 `isInputSizeMutableImpl`
- 7 `isTunablePropertyDataTypeMutableImpl`
- 8 `validateInputsImpl`
- 9 If the System object uses the `matlab.system.mixin.Nondirect` mixin, call `isInputDirectFeedthroughImpl`
- 10 `setupImpl`

Running the Object or step Call Sequence

When you run a System object in MATLAB, either by calling the object as a function or calling `step`, this sequence of methods is called:

- 1 If the System object is not in use (object was just created or was released),
 - a `release` on page 33-62
 - b `setup` on page 33-63
 - c `reset` on page 33-64

Else, if the object is in use (object was called and `release` was not called)

- a If tunable properties have changed
 - i `validatePropertiesImpl`
 - ii `processTunedPropertiesImpl`
- b If the input size, data type, or complexity has changed
 - i `validateInputsImpl`
 - ii `processInputSpecificationChangeImpl`

reset Call Sequence

When `reset` is called, these actions are performed.

- 1 If the object is in use (object was called and not released), call `resetImpl`

release Call Sequence

When `release` is called, these actions are performed.

- 1 If the object is in use (object was called and not released), call `releaseImpl`

See Also

`releaseImpl` | `resetImpl` | `setupImpl` | `stepImpl`

Related Examples

- “Release System Object Resources” on page 33-30
- “Reset Algorithm State” on page 33-30
- “Set Property Values at Construction Time” on page 33-28
- “Define Basic System Objects” on page 33-15

Tips for Defining System Objects

A System object is a specialized MATLAB object that is optimized for iterative processing. Use System objects when you need to run an object multiple times or process data in a loop. When defining your own System object, use the following suggestions to help your System object run more quickly.

General

- Define all one-time calculations in the `setupImpl` method and cache the results in a private property. Use the `stepImpl` method for repeated calculations.
- Specify Boolean values using `true` or `false` instead of `1` or `0`, respectively.
- If the variables in a method do not need to retain their values between calls, use local scope for those variables in that method.

Inputs and Outputs

- Some methods use the `stepImpl` algorithm inputs as their inputs, such as `setupImpl`, `updateImpl`, `validateInputsImpl`, `isInputDirectFeedThroughImpl`, and `processInputSizeChangeImpl`. The inputs must match the order of inputs to `stepImpl`, but do not need to match the number of inputs. If your implementation does not require any of the inputs to the System object, you can leave them all off.
- For the `getNumInputsImpl` and `getNumOutputsImpl` methods, if you set the return argument from an object property, that object property must have the `Nontunable` attribute.

Using ~ as an Input Argument in Method Definitions

All methods, except static methods, expect the System object handle as the first input argument. You can use any name for your System object handle. The code inserted by the MATLAB Editor menu uses `obj`.

In many examples, instead of passing in the object handle, `~` is used to indicate that the object handle is not used in the function. Using `~` instead of an object handle prevents warnings about unused variables.

Properties

- For properties that do not change, define them in as `Nontunable` properties. `Tunable` properties have slower access times than `Nontunable` properties
- Whenever possible, use the `protected` or `private` attribute instead of the `public` attribute for a property. Some `public` properties have slower access times than `protected` and `private` properties.
- If properties are accessed more than once in the `stepImpl` method, cache those properties as local variables inside the method. A typical example of multiple property access is a loop. Iterative calculations using cached local variables run faster than calculations that must access the properties of an object. When the calculations for the method complete, you can save the local cached results back to the properties of that System object. Copy frequently used tunable properties into private properties. This best practice also applies to the `updateImpl` and `outputImpl` methods.

For example, in this code `k` is accessed multiple times in each loop iteration, but is saved to the object property only once.

```
function y = stepImpl(obj,x)
    k = obj.MyProp;
    for p=1:100
        y = k * x;
        k = k + 0.1;
    end
    obj.MyProp = k;
end
```

- Default values of properties are shared across all instances of an object. Two instances of a class can access the same default value if that property has not been overwritten by either instance.

Text Comparisons

Do not use character vector comparisons or character vector-based switch statements in the `stepImpl` method. Instead, create a method handle in `setupImpl`. This handle points to a method in the same class definition file. Use that handle in a loop in `stepImpl`.

This example shows how to use method handles and cached local variables in a loop to implement an efficient object. In `setupImpl`, choose `myMethod1` or `myMethod2` based on a character vector comparison and assign the method handle to the `pMethodHandle`

property. Because there is a loop in `stepImpl`, assign the `pMethodHandle` property to a local method handle, `myFun`, and then use `myFun` inside the loop.

```
classdef MyClass < matlab.System
function setupImpl(obj)
    if strcmp(obj.Method, 'Method1')
        obj.pMethodHandle = @myMethod1;
    else
        obj.pMethodHandle = @myMethod2;
    end
end
function y = stepImpl(obj,x)
    myFun = obj.pMethodHandle;
    for p=1:1000
        y = myFun(obj,x)
    end
end
function y = myMethod1(x)
    y = x+1;
end
function y = myMethod2(x)
    y = x-1;
end
end
```

Simulink

For System objects being included in Simulink, add the `StrictDefaults` attribute. This attribute sets all the `MutableImpl` methods to return false by default.

Code Generation

For information about System objects and code generation, see “System Objects in MATLAB Code Generation” (MATLAB Coder).

Insert System Object Code Using MATLAB Editor

In this section...

“Define System Objects with Code Insertion” on page 33-69

“Create a Temperature Enumeration” on page 33-73

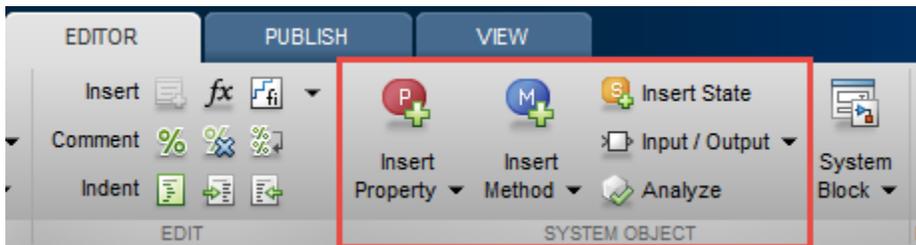
“Create Custom Property for Freezing Point” on page 33-75

“Add Method to Validate Inputs” on page 33-76

Define System Objects with Code Insertion

You can define System objects from the MATLAB Editor using code insertion options. When you select these options, the MATLAB Editor adds predefined properties, methods, states, inputs, or outputs to your System object. Use these tools to create and modify System objects faster, and to increase accuracy by reducing typing errors.

To access the System object editing options, create a new System object, or open an existing one.



To add predefined code to your System object, select the code from the appropriate menu. For example, when you click **Insert Property > Numeric**, the MATLAB Editor adds the following code:

```
properties(Nontunable)
    Property
end
```

The MATLAB Editor inserts the new property with the default name `Property`, which you can rename. If you have an existing properties group with the `Nontunable` attribute, the MATLAB Editor inserts the new property into that group. If you do not have a property group, the MATLAB Editor creates one with the correct attribute.

Insert Options

Properties	Properties of the System object: Numeric, Logical, Enumeration, Positive Integer, Tunable Numeric, Private, Protected, and Custom. When you select Enumeration or Custom Properties, a separate dialog box opens to guide you in creating these properties.
-------------------	---

Methods	<p>Methods commonly used in System object definitions. The MATLAB Editor creates only the method structure. You specify the actions of that method.</p> <p>The Insert Method menu organizes methods by categories, such as Algorithm, Inputs and Outputs, and Properties and States. When you select a method from the menu, the MATLAB Editor inserts the method template in your System object code. In this example, selecting Insert Method > Release resources inserts the following code:</p> <pre>function releaseImpl(obj) % Release resources, such as file handles end</pre> <p>If a method from the Insert Method menu is present in the System object code, that method is shown shaded on the Insert Method menu:</p>
----------------	--

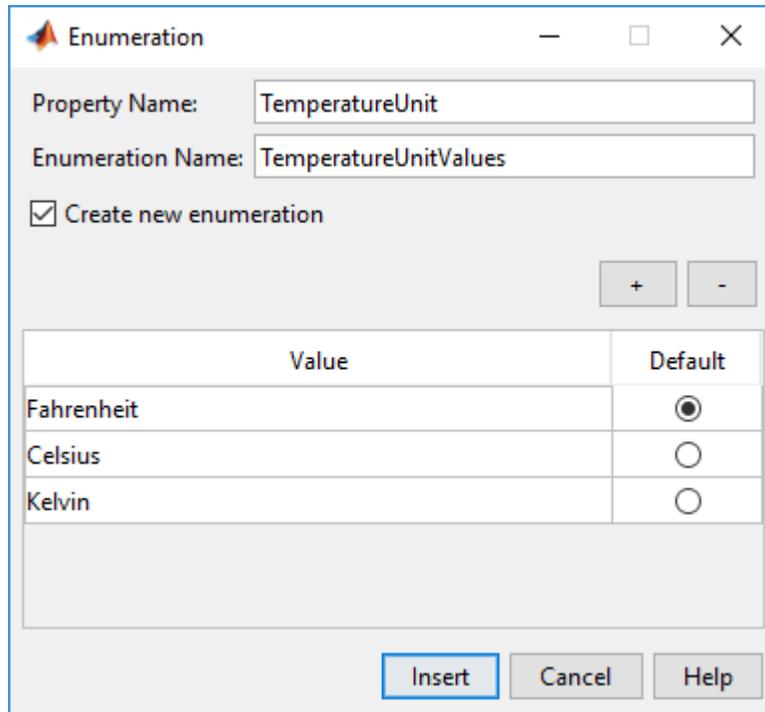
	<p>ALGORITHM</p> <ul style="list-style-type: none">  Implement algorithm stepImpl  Perform one-time calculations setupImpl  Initialize / reset states resetImpl  Release resources releaseImpl <p>CONSTRUCTOR TOP ▲</p> <ul style="list-style-type: none">  Set properties from name-value pairs System object constructor <p>PROPERTIES AND STATES TOP ▲</p> <ul style="list-style-type: none">  Validate related property values validatePropertiesImpl <p>ations if tun</p>
States	Properties containing the DiscreteState attribute.

Inputs / Outputs	<p>Inputs, outputs, and related methods, such as Validate inputs and Disallow input size changes.</p> <p>When you select an input or output, the MATLAB Editor inserts the specified code in the <code>stepImpl</code> method. In this example, selecting Insert > Input causes the MATLAB Editor to insert the required input variable <code>u2</code>. The MATLAB Editor determines the variable name, but you can change it after it is inserted.</p> <pre>function y = stepImpl(obj,u,u2) % Implement algorithm. Calculate y as a function of % input u and discrete states. y = u; end</pre>
-------------------------	---

Create a Temperature Enumeration

- 1 Open a new or existing System object.
- 2 In the MATLAB Editor, select **Insert Property > Enumeration**.
- 3 In the **Enumeration** dialog box, enter:
 - a **Property Name** with `TemperatureUnit`.
 - b **Enumeration Name** with `TemperatureUnitValues`.
- 4 Select the **Create new enumeration** checkbox.
- 5 Remove the existing enumeration values with the - (minus) button.
- 6 Add three an enumeration values with the + (plus) button and the following values:
 - `Fahrenheit`
 - `Celsius`
 - `Kelvin`
- 7 Select `Fahrenheit` as the default value by clicking **Default**.

The dialog box now looks as shown:



- 8 To create this enumeration and the associated class, click **Insert**.
- 9 In the MATLAB Editor, an additional class file with the enumeration definition is created. Save the enumeration class definition file as `TemperatureUnitValues.m`.

```
classdef TemperatureUnitValues < int32
    enumeration
        Fahrenheit (0)
        Celsius (1)
        Kelvin (2)
    end
end
```

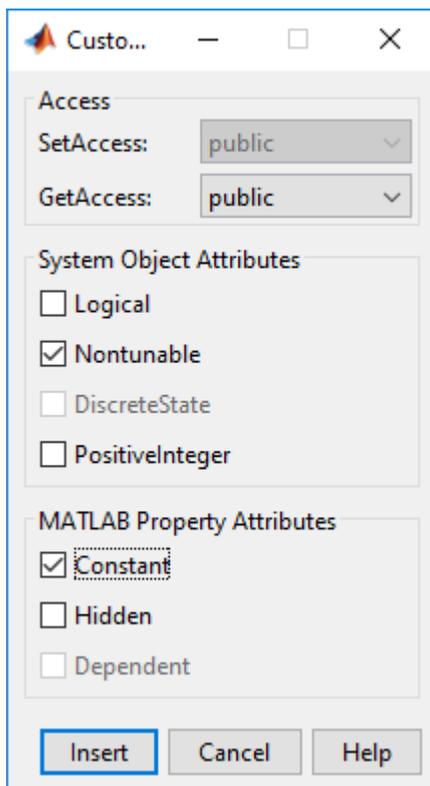
In the System object class definition, the following code was added:

```
properties(Nontunable)
    TemperatureUnit (1, 1) TemperatureUnitValues = TemperatureUnitValues.Fahrenheit
end
```

For more information on enumerations, see “Limit Property Values to Finite List” on page 33-38.

Create Custom Property for Freezing Point

- 1 Open a new or existing System object.
- 2 In the MATLAB Editor, select **Insert Property > Custom Property**.
- 3 In the Custom Property dialog box, under **System Object Attributes**, select **Nontunable**. Under **MATLAB Property Attributes**, select **Constant**. Leave **GetAccess** as **public**. **SetAccess** is grayed out because properties of type constant cannot be set using System object methods.



- 4 Click **Insert** and the following code is inserted into the System object definition:


```
properties(Nontunable, Constant)
    Property
end
```
- 5 Replace `Property` with your property.

```
properties(Nontunable, Constant)
    FreezingPointFahrenheit = 32;
end
```

Add Method to Validate Inputs

- 1 Open a new or existing System object.
- 2 In the MATLAB Editor, select **Insert Method > Validate inputs**.

The MATLAB Editor inserts this code into the System object:

```
function validateInputsImpl(obj,u)
    % Validate inputs to the step method at initialization
end
```

See Also

Related Examples

- “Analyze System Object Code” on page 33-77

Analyze System Object Code

In this section...
“View and Navigate System object Code” on page 33-77
“Example: Go to StepImpl Method Using Analyzer” on page 33-77

View and Navigate System object Code

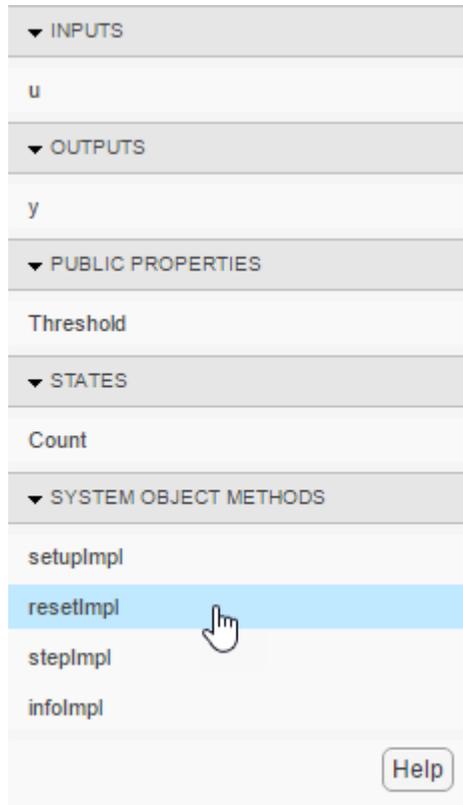
View and navigate System object code using the Analyzer.

The Analyzer displays all elements in your System object code.

- Navigate to a specific input, output, property, state, or method by clicking the name of that element.
- Expand or collapse element sections with the arrow buttons.
- Identify access levels for properties and custom methods with the + (public), # (protected), and - (private) symbols.

Example: Go to StepImpl Method Using Analyzer

- 1 Open an existing System object.
- 2 Select **Analyze**.
- 3 Click **resetImpl**.



The cursor in the MATLAB Editor window jumps to the `resetImpl` method.

```
10     end
11
12     methods (Access = protected)
13     function setupImpl(obj)
14         obj.Count = 0;
15     end
16
17     function resetImpl(obj)
18         obj.Count = 0;
19     end
20
21     function y = stepImpl(obj,u)
22         if (u > obj.Threshold)
23             obj.Count = obj.Count + 1;
24         end
```

See Also

Related Examples

- “Insert System Object Code Using MATLAB Editor” on page 33-69

Use Global Variables in System Objects

Global variables are variables that you can access in other MATLAB functions or Simulink blocks.

System Object Global Variables in MATLAB

For System objects that are used only in MATLAB, you define global variables in System object class definition files in the same way that you define global variables in other MATLAB code (see “Global Variables” on page 20-13).

System Object Global Variables in Simulink

For System objects that are used in the MATLAB System block in Simulink, you also define global variables as you do in MATLAB. However, to use global variables in Simulink, you must declare global variables in the `stepImpl`, `updateImpl`, or `outputImpl` method if you have declared them in methods called by `stepImpl`, `updateImpl`, or `outputImpl`, respectively.

You set up and use global variables for the MATLAB System block in the same way as you do for the MATLAB Function block (see “Data Stores” (Simulink) and “Share Data Globally” (Simulink)). Like the MATLAB Function block, you must also use variable name matching with a Data Store Memory block to use global variables in Simulink.

For example, this class definition file defines a System object that increments the first row of a matrix by 1 at each time step. You must include `getGlobalNamesImpl` if the class file is P-coded.

```
classdef GlobalSysObjMatrix < matlab.System
    methods (Access = protected)
        function y = stepImpl(obj)
            global B;
            B(1,:) = B(1,:)+1;
            y = B;
        end

        % Include getGlobalNamesImpl only if the class file is P-coded.
        function globalNames = getGlobalNamesImpl(~)
            globalNames = {"B"};
        end
end
```

```
end  
end
```

This model includes the `GlobalSysObjMatrix` object in a MATLAB System block and the associated Data Store Memory block.

